Become more productive
with the best Python IDE

Modern Python Development With **PyCharm**

Pedro Kroger

Version 1.1

Contents

1 | Introduction

Why Should I Use PyCharm?

I know how productive we can be with our favorite editor. I've been using Emacs for more than fifteen years and, at one point, I used it to read my email, browse the web, and make coffee. (Haven't we all?) But today I find that I'm more productive using the right tool for the right job.

Powerful editors like Emacs and Vim are fantastic. I find it puzzling, for instance, that some editors and IDEs perform basic operations such as search and replace awkwardly, whereas Vim and Emacs get it just right.

And yet a search for terms like "Vim as a Python IDE" or "Emacs as a Python IDE" returns hundreds of thousands of links, which shows that people want features such as smart completion, code navigation, and project support. In fact, one of the most popular articles on my website explains how to configure Emacs as a Python IDE, even if it's old and hopelessly out-of-date.

PyCharm is the most popular IDE for Python, and it's packed with great features such as unsurpassed code completion and inspection, an advanced debugger, and support for web programming and frameworks such as Django and Flask.

In this book, I'm using PyCharm 4.0.4 Professional Edition.

Conventions Used in This Book

To make things concise, shortcuts are written with the first letter of keyboard modifiers (or the funny Mac symbols) followed by the actual keys in lower case. For instance, *C-a* means Control and the "a" key. The keys are presented for both OS X and Windows/Linux in the format (mac, win). For example, the shortcut *(⌘-S-⌥-t, S-A-C-t)* means to press the keys ⌘, Shift, Option, and t at the same time on the Mac, and Shift, Alt, Control, and t on Windows and Linux. When only one shortcut is presented, it will be the same on all operating systems.

PyCharm uses the words Preferences and Settings interchangeably. For consistency, I use only Settings in this book, as in *Settings → Editor → Live Templates*.

The following typographical conventions are used in this book:

Italics Action and commands, menu items, shortcuts, configuration paths.

Quotation marks Strings and elements on the screen. When the item mentioned is short and unambiguous, the quotes may be dropped to make the text cleaner. Therefore, the checkbox named "click to show preview" will have quotes whereas the Configure button won't.

Constant width File names, code.

Light green Links to external pages and sections inside this document.

Screenshots

The screenshots were taken on a retina computer, so the quality should be pretty good. However, depending on your system, you may need to zoom in to be able to see some details better. Also, some screenshots have been

lightly edited to keep the focus on the features being discussed. For instance, a long menu can be shortened by removing unrelated items. If things look different on your screen, it may be due to this or you may be using a different PyCharm version.

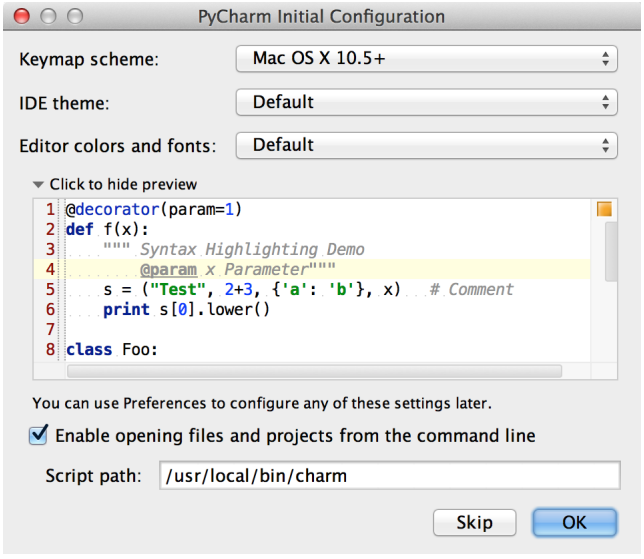
Acknowledgments

This book would not be possible without the encouragement of many people. I'd like to thank the nice JetBrains folks working on PyCharm, especially Dmitry Filippov. Finally, I'd like to acknowledge Mara for her patience, love, and support.

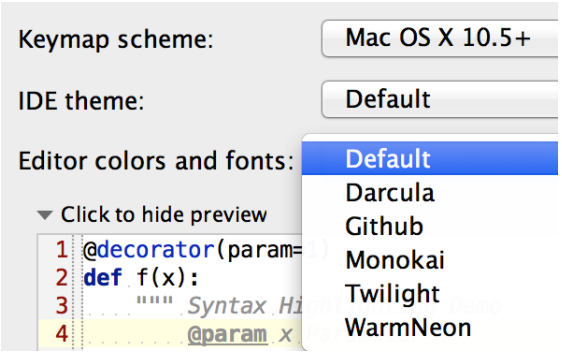
2 | Getting Started

Initial Configuration

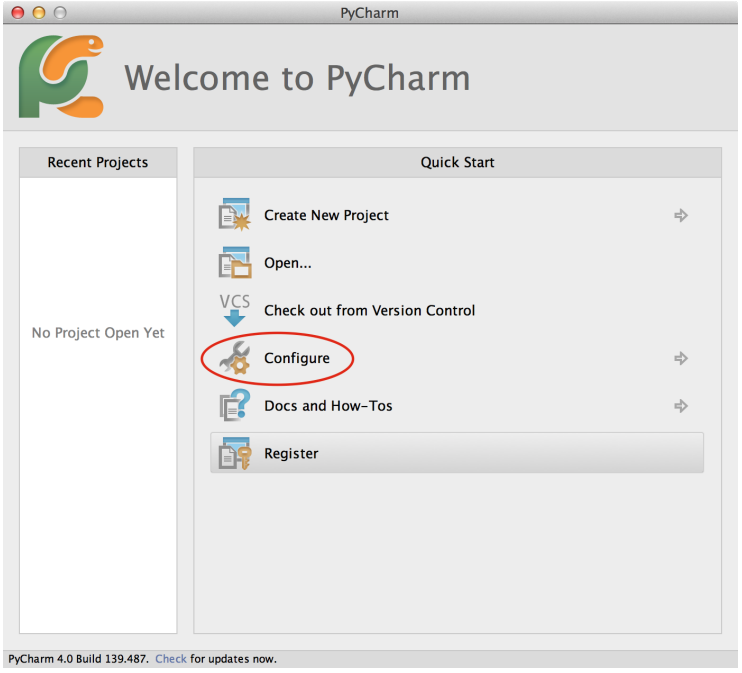
When you start PyCharm for the first time, it asks what keymap and theme you want to use. I use the default Mac OS X keymap and customize the shortcuts I want (see *Shortcuts*). You can click on "Click to show preview" to see how the theme will look.



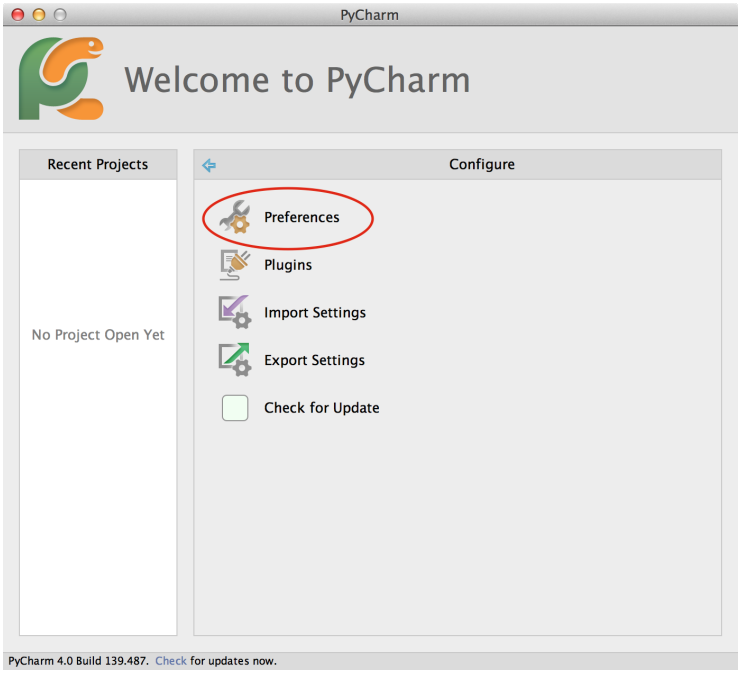
We can also preview the color schemes by selecting one in “Editor colors and fonts.”



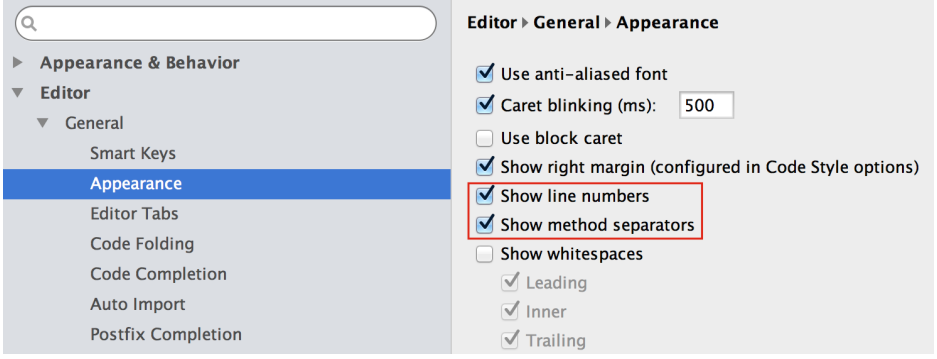
If this is the first time you are using PyCharm, and you don’t have a configuration to import, you may want to click on Configure in the welcome screen to set up some basic settings.



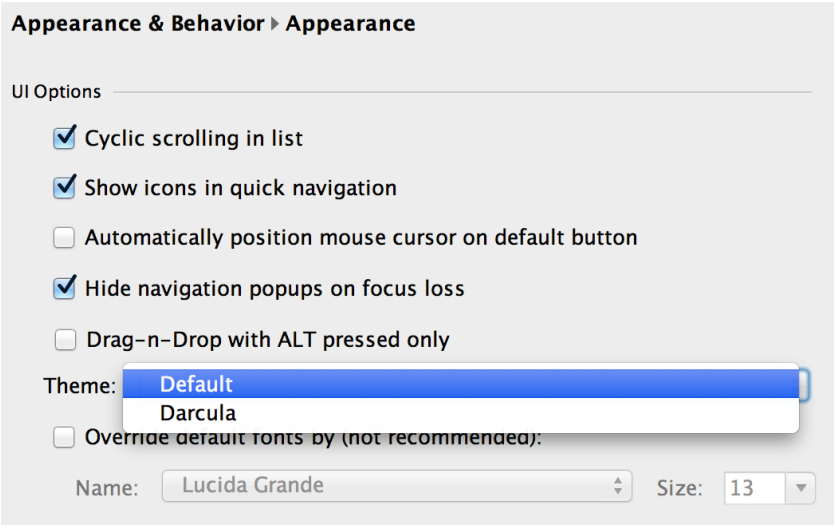
The next screen allows us to set preferences, install plugins, and import and export settings. We want to click on Preferences:



I like to show the line numbers and method separators (*Settings → Editor → General → Appearance*).

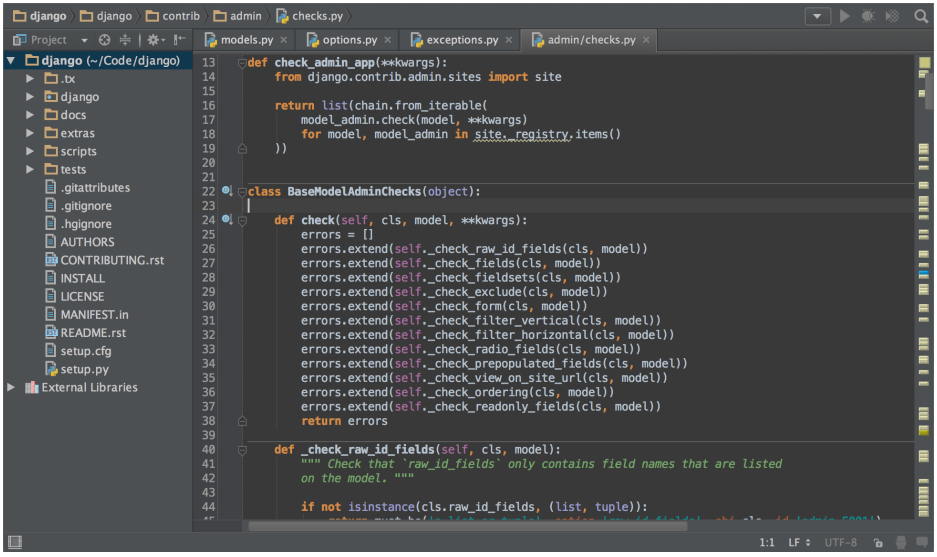


We have two ways to change the way PyCharm looks. One is by changing the whole IDE theme. It controls how things such as the toolbar, dialogs, and sidebar will look. The other is by changing the editor color scheme. To set the IDE theme, we go to *Settings → Appearance & Behavior → Appearance* and choose the appropriate theme. On the Mac, there are only two themes: Default and Darcula. We can't customize them; they are hardcoded in PyCharm.

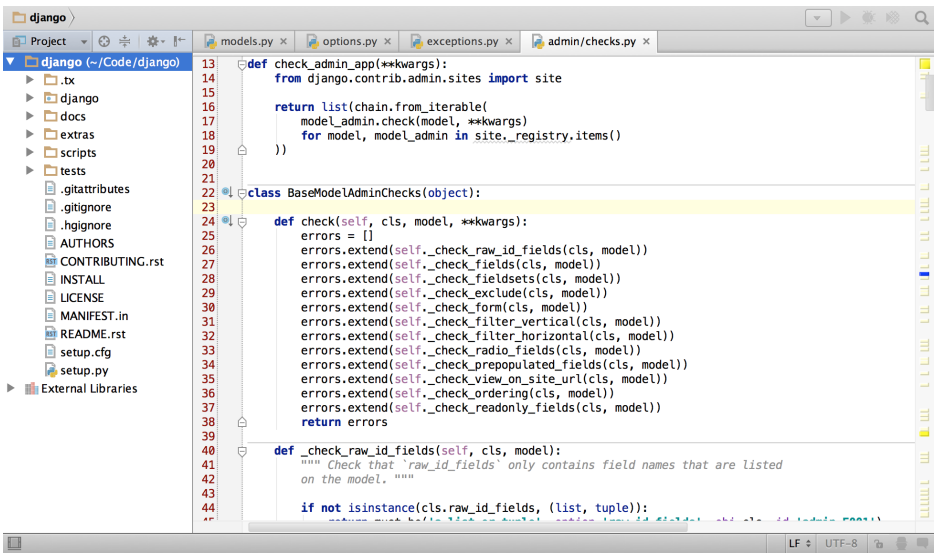


Darcula is a nice dark theme, whereas Default has the standard grayish look and a white background for the editor. I'll use the standard theme in this book because black on white tends to look better on different devices, sizes, and resolutions (for instance, dark images don't look good on the Kindle).

The Darcula look and feel:

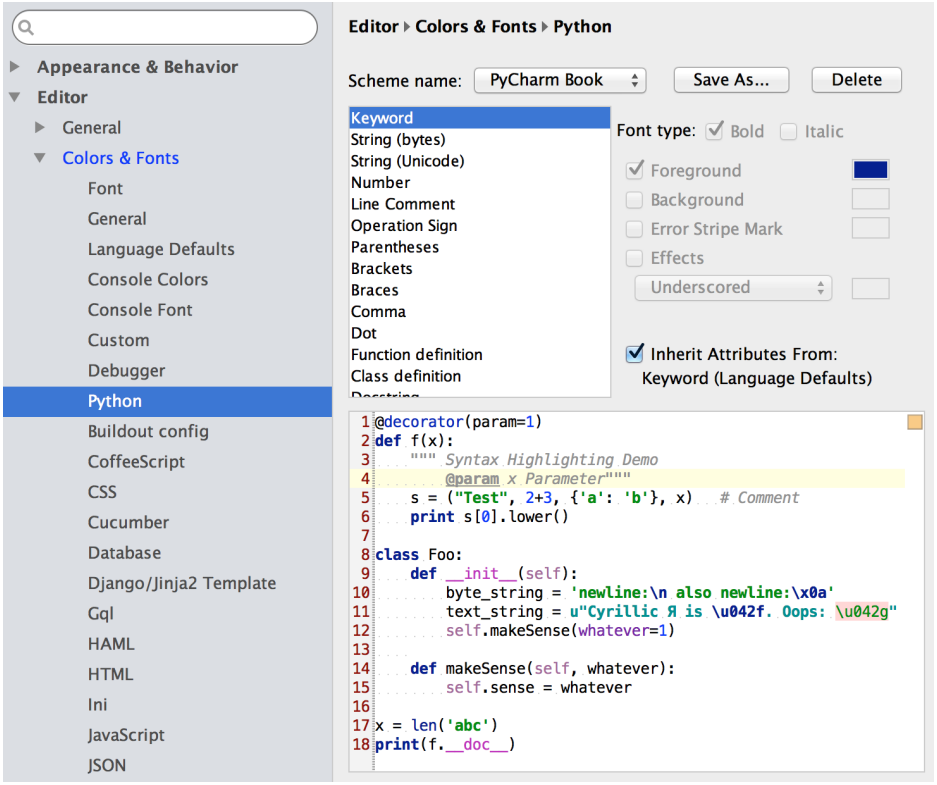


The Default look and feel:

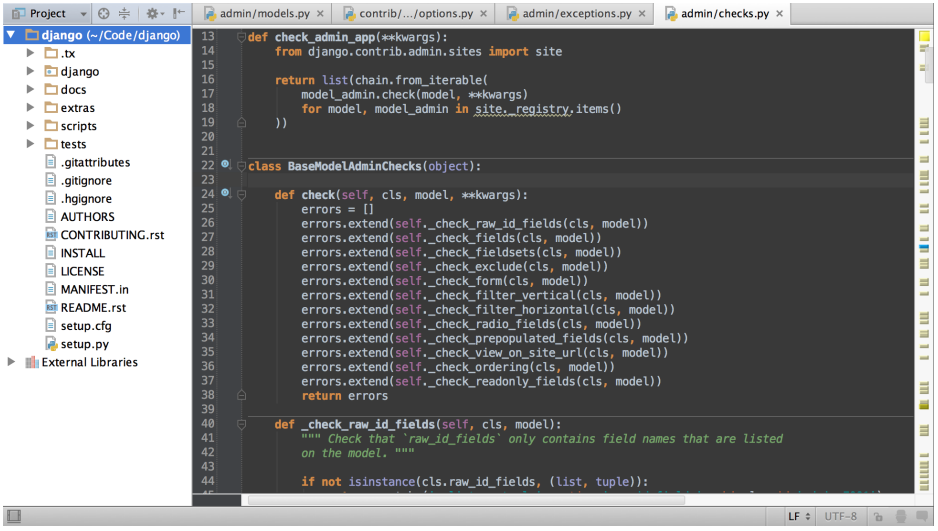


To set the editor color scheme, we go to *Settings → Editor → Colors & Fonts*

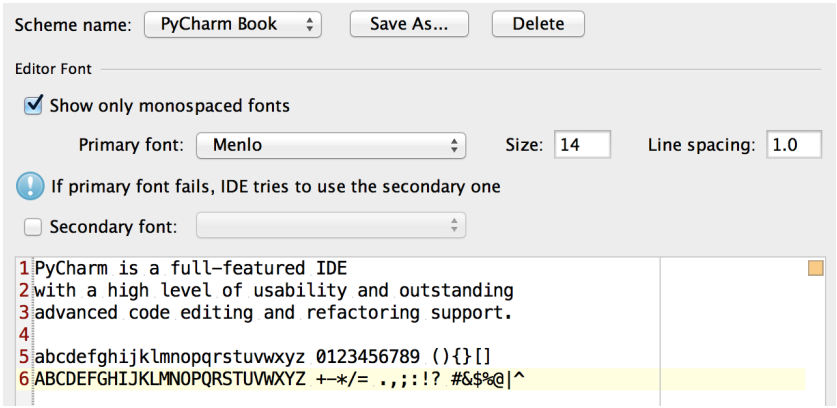
and pick a scheme. Unlike IDE themes, we can use the built-in configuration to make very specific changes to a scheme. We can control how different programming elements such as keywords and strings look. My advice is not to be afraid to change the colors to something you like. You can export your settings in case you need to reinstall PyCharm or use it in another computer. Also, there are dozens of themes available on the Internet such as IDEA color and pycharm-themes.



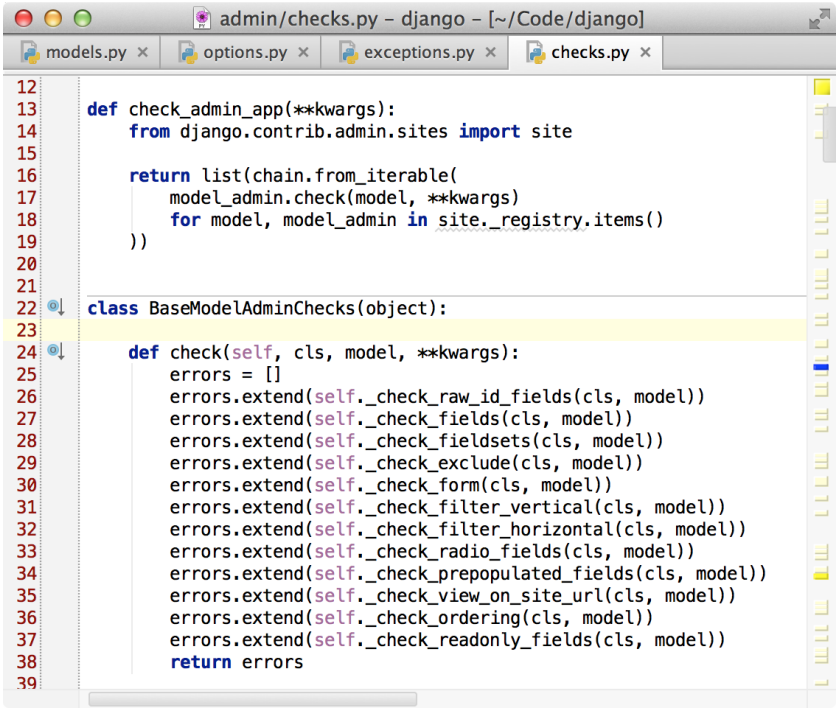
IDE themes and color schemes are independent of each other. We can have a dark IDE such as like Darcula with a light editor scheme or vice versa. The following is the Default theme with the Darcula color scheme (Darcula is the name of both an IDE theme and a color scheme):



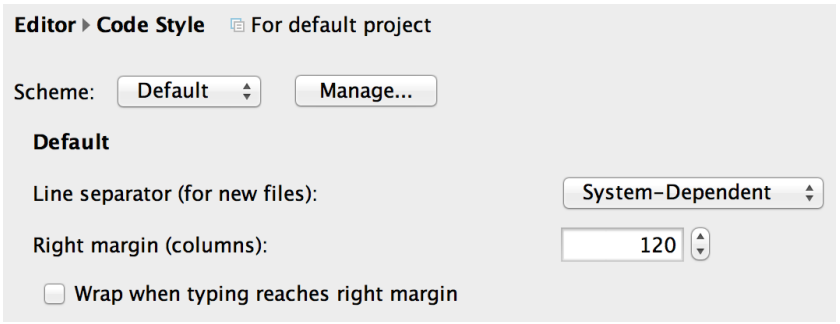
The font type and size is another thing you may want to change (*Settings → Editor → Colors & Fonts → Font*).



PyCharm's default interface is very clean. By default it doesn't show the Toolbar and Tool buttons (in previous versions they were shown by default), and you can unhide them in the *View* menu. You can also hide the Status and Navigation bars if you want to have an even cleaner interface.

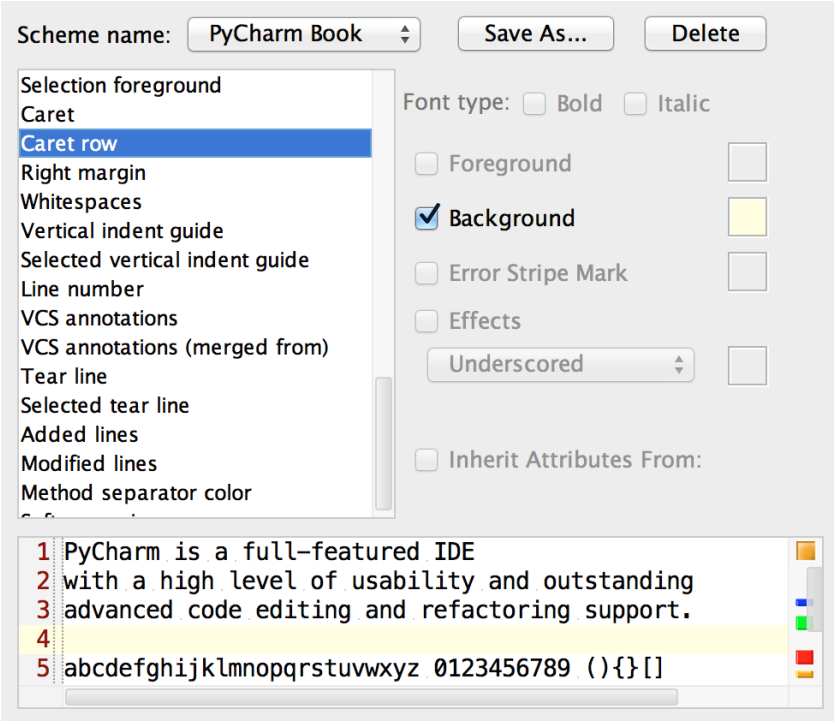


PyCharm uses 120 characters per line instead of the recommended 79. You can change that in *Settings* → *Editor* → *Code Style*.



Another thing you may want to change is the color that highlights the current line in *Settings* → *Editor* → *Colors & Fonts* on the impossible-to-guess

Caret row:

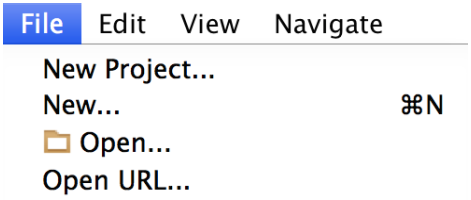


These are the basic configurations before starting to work. The next step is to tell PyCharm what Python interpreter you want to use. Because the choice of interpreters will probably be coupled with a specific project (that is, we can have Project A using Python 2.7 and Project B using Python 3.3), it makes sense to see how to deal with new and existing projects. We'll also see how to isolate Python projects by creating separate environments with `virtualenv`.

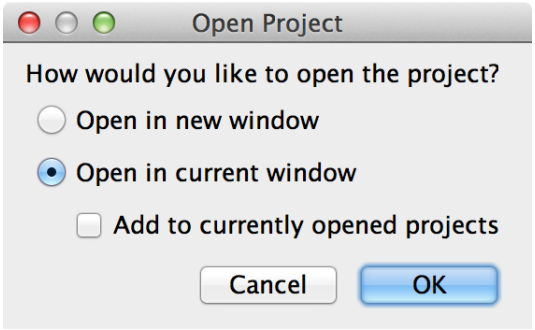
Working with Projects

As with other IDEs, PyCharm is based on projects; everything happens in the scope of a project. However, unlike other IDEs, project creation and management in PyCharm couldn't be easier. In PyCharm, a project is a directory with source code and a directory named `.idea`. PyCharm stores all necessary metadata in this directory, and you never need to worry about its existence. Therefore, from our perspective, a project is just a directory with Python code.

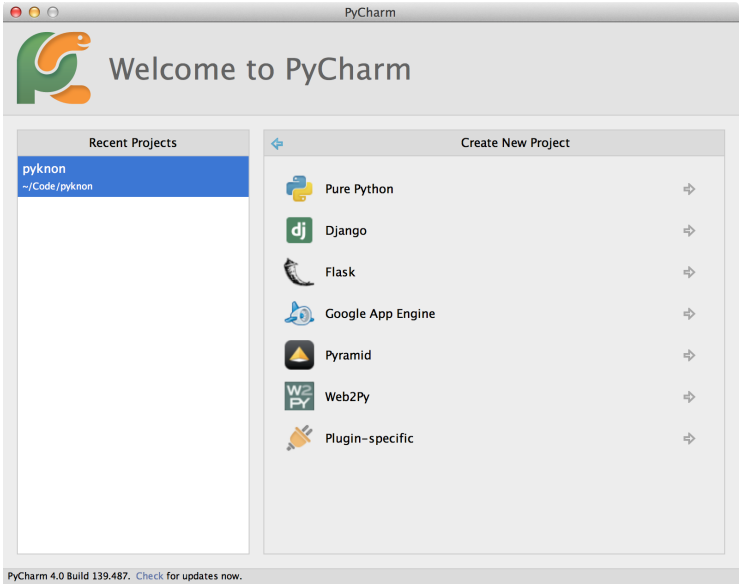
The PyCharm manual recommends putting the `.idea` directory under version control, with the exception of the `workspace.xml` file, which should be ignored by the version control system. This file contains your individual workspace preferences. We can explicitly create a new project (*File → New Project*) or open an existing directory as a project (*File → Open*). When we open an existing directory, PyCharm will automatically create the `.idea` directory if it doesn't exist, turning the directory into a project. There are no default keyboard shortcuts for these commands, but we can create our own if we want (see *Shortcuts*).



If we try to open a directory (for instance, `project2`) while a project (for example, `project1`) is already opened, PyCharm will ask if we want to open it in a new window or in the current window. If we keep "Add to currently opened projects" unchecked, it will close `project1` before opening `project2`. If we check "Add to currently opened projects," it will include `project2` *inside* `project1` (which is probably what we don't want most of the time). We can always remove a project included in another project by selecting it and going to *Edit → Remove From Project View*.

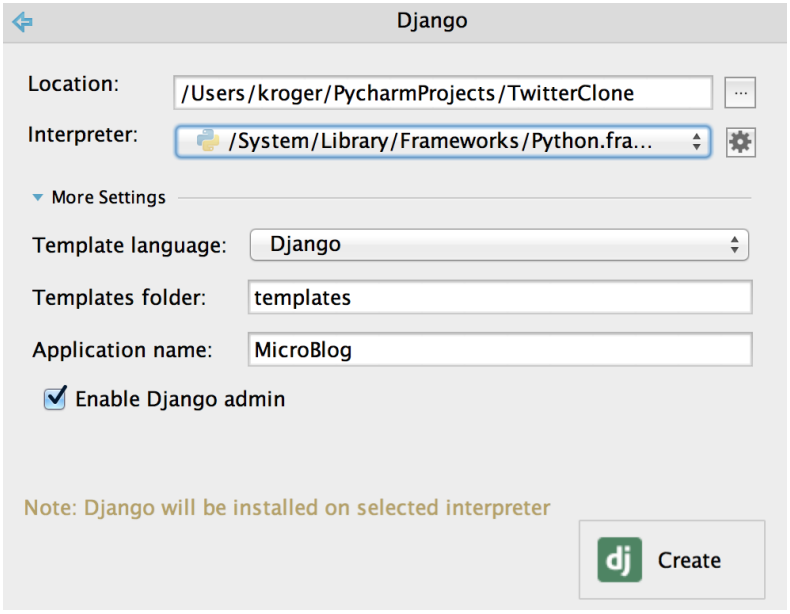


We create a new project in PyCharm in *File → New Project*. We can assign a project name and location and select a project type and interpreter. PyCharm has templates for many project types such as Django, Google App Engine, Pyramid, Flask, and Web2Py.

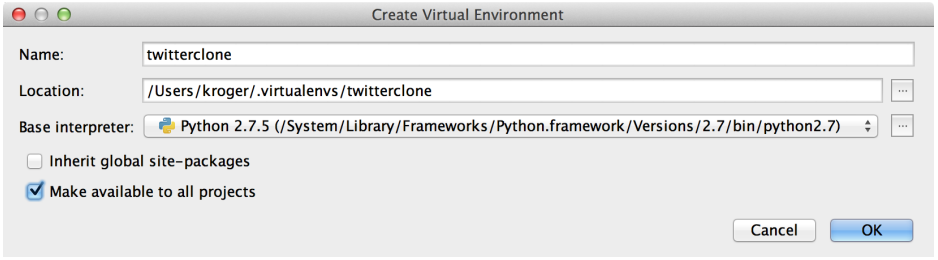


In the following example, we'll create a new Django project. PyCharm will automatically install Django (it works similarly for Flask and Pyramid).

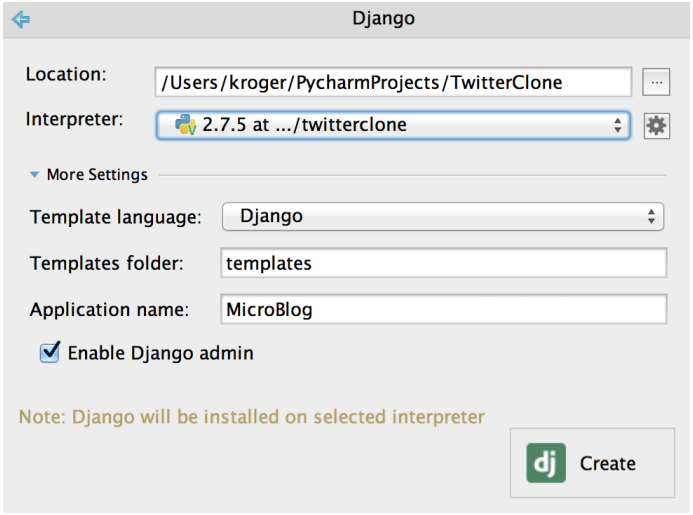
- Select the project type and add the project location, template folder, application name and select the template language.



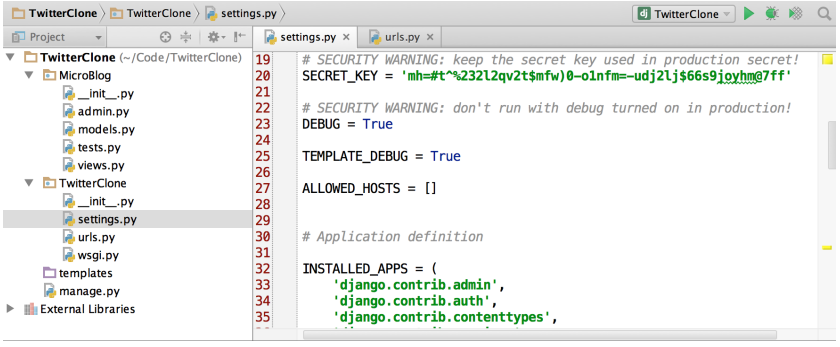
- Create a new `virtualenv` by clicking on the "settings" icon on the right. Set the environment's name and location (PyCharm tries to guess the default location) and base Python interpreter:



- Select the new environment on the Interpreter list and click on the Create button.

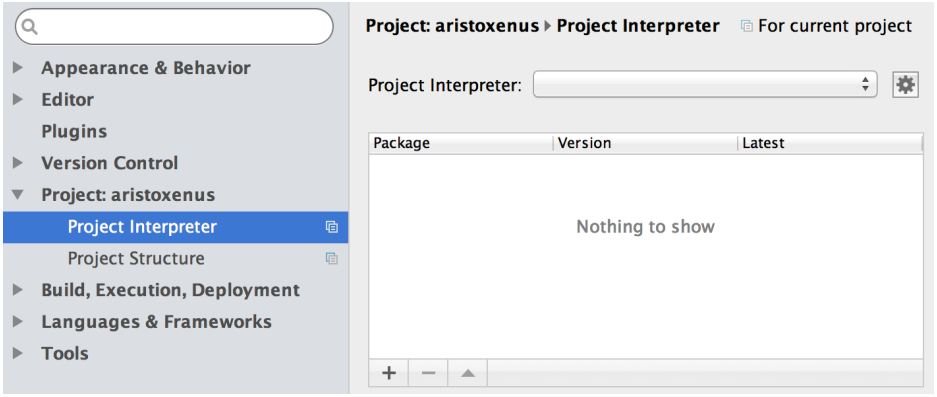


PyCharm will create boilerplate code for projects automatically. We could achieve the same effect by using the framework's tool, such as Django's `django-manager.py` or Pyramid's `pcreate`, but it's super quick to do it from the IDE itself. Conversely, if you are using custom templates, I'd recommend creating projects on the command line and opening them on PyCharm afterward.

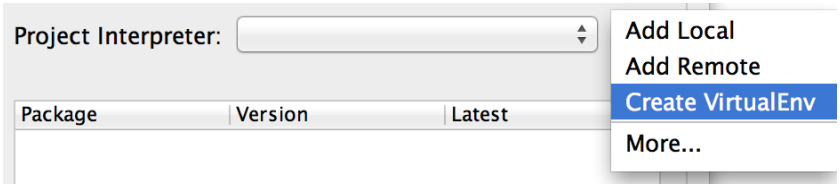


Virtualenv and Packages

We need to tell PyCharm what Python interpreter we want to use because we can have a different interpreter for each project. PyCharm will use this information to index all available libraries and for code completion, code analysis, and other features. A good practice is to have one virtual environment for each project. To set up the Python interpreter for the current project, go to *Settings* → *Project: <project name>* → *Project Interpreter* and click on the + button to use an existing `virtualenv`, or click on the "settings" icon on the right to create a new environment.

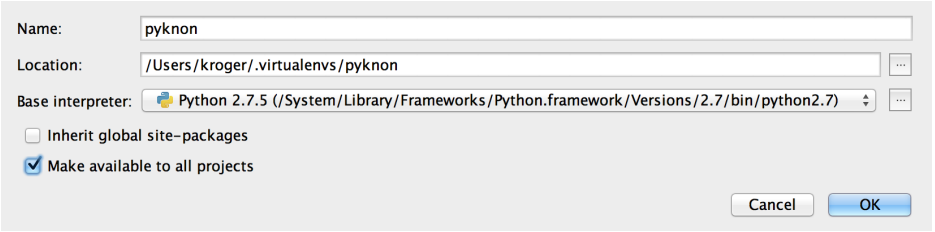


We can create either a new environment or add a local or remote one:

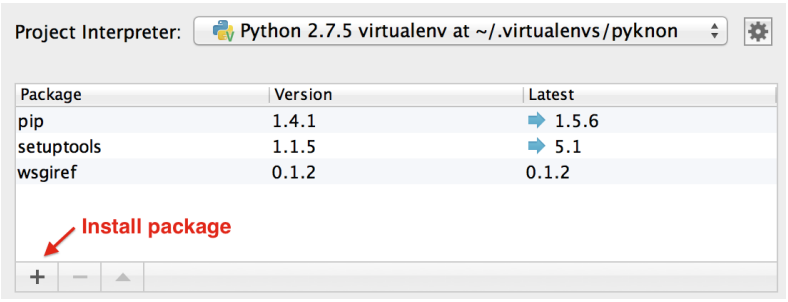


Creating new virtual environments is straightforward and identical when we create a new project (see *Working with Projects*); we can give it a name and

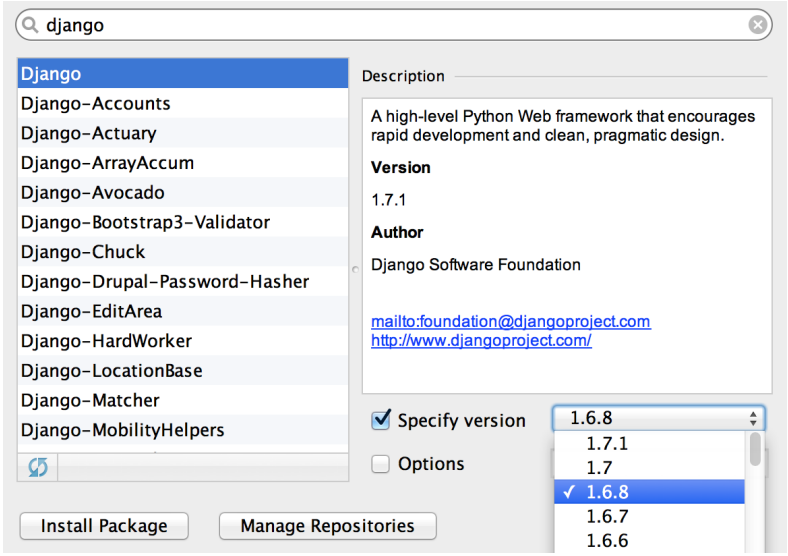
location, choose a base interpreter, and make it available to all projects.



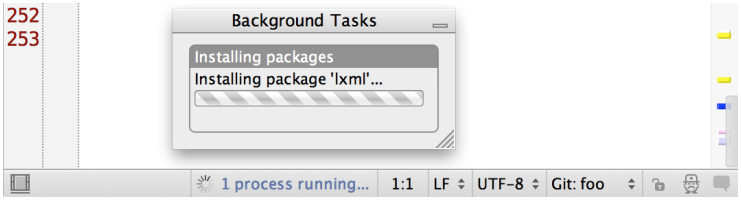
We can quickly search, read the description, and install Python packages from PyCharm. This is nothing we couldn't do in the terminal, but it's very convenient to be able to search for packages without leaving the IDE. As expected, the packages will be installed in the selected interpreter's path.



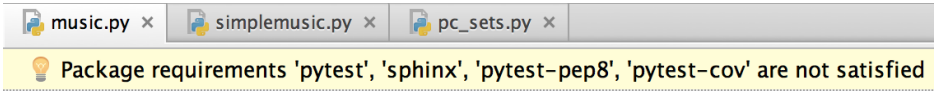
We can see a package's description and select a specific version to install.



Some packages may take a while to install, especially if they need to be compiled. PyCharm runs the installation in background and we can see what is happening by clicking in the status bar:

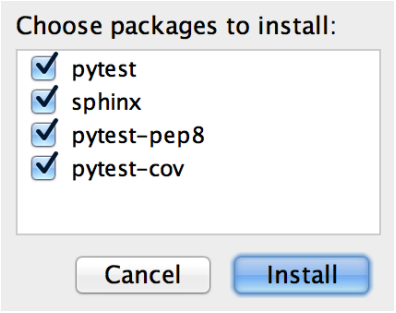


PyCharm will recognize `requirement.txt` files and offer to install the packages if they are not installed.



When we click on "Install requirements," all packages will be already selected,

so we just need to click on the Install button:



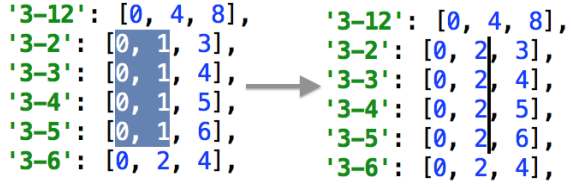
3 | Editing and Navigating

Editing

PyCharm has powerful features for code editing. In this section, we will see the most important features, but you can see all of them in the manual.

Column Selection and Multiple Carets

As with many modern editors, PyCharm has support for column selection and multiple carets. Because column selection is implemented as a mode, we need to toggle it by going to *Edit → Column Selection Mode (⌘-S-8, S-A-Insert)*. If column mode is switched on, any selection made with either the keyboard or mouse will be a rectangular grid.



Once the grid is selected, we can use most commands that work on selected

text, such as editing in-place, and search and replace. We can select a column of text quickly without switching to column selection mode by pressing *Alt* and selecting the text by dragging the mouse pointer.

We can also have multiple, independent carets. We create a caret with *Shift-Alt-Click* on the position of the new caret. Once the new caret has been created, we can perform most editing actions. Multiple carets are particularly good when we want to perform multiple operations on blocks of codes that have a similar structure. In the following example, I created four carets to edit the `class` of both `div` and `label`. I'd need to search and replace twice to accomplish the same thing.

A quick way to create multiple carets is by searching. We can select multiple occurrences of a word under the caret by going to *Edit → Find → Add Selection for Next Occurrence (C-g)*. In the following example, the caret started on the first `note` variable (on the second line). I selected the next `notes` by typing *C-g* five times.

```
def transposition_startswith(self, note_start):
    note = self._make_note(note_start)
    return self.transposition(note - self.items[0])

def inversion(self, index=0):
    initial_octave = self.items[0].octave
    return NoteSeq([x.inversion(index, initial_octave) if isinstance(x, Note)
                    else x for x in self.items])

def inversion_startswith(self, note_start):
    note = self._make_note(note_start)
    inv = self.transposition_startswith(Note(0, note.octave)).inversion()
    return inv.transposition_startswith(note)
```

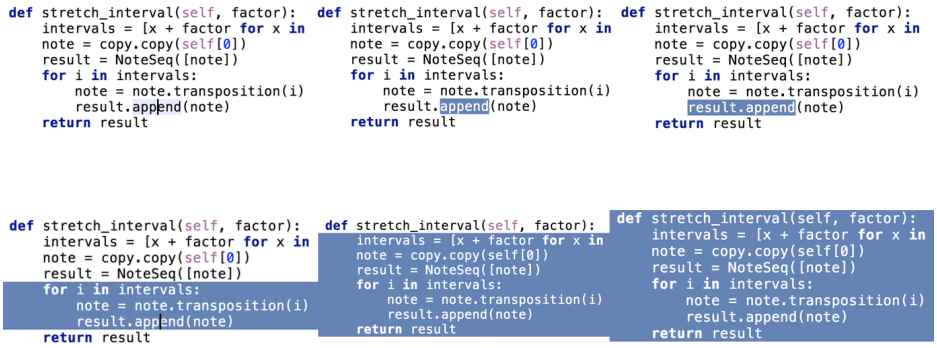
After that, we have five carets ready to be used:

```
def transposition_startswith(self, note_start):
    note = self._make_note(note_start)
    return self.transposition(note - self.items[0])

def inversion(self, index=0):
    initial_octave = self.items[0].octave
    return NoteSeq([x.inversion(index, initial_octave) if isinstance(x, Note)
                    else x for x in self.items])

def inversion_startswith(self, note_start):
    note = self._make_note(note_start)
    inv = self.transposition_startswith(Note(0, note.octave)).inversion()
    return inv.transposition_startswith(note)
```

Instead of selecting matches one by one, we can select all occurrences at once by going to *Edit → Find → Select All Occurrences (S-C-g)*.

Select Words

A quick way to select (and unselect) expressions by scope is by using *Select Word at Caret (⌥-↑, C-w)* and *Unselect Word at Caret (⌥-↓, C-w)*. When called once, *Select Word at Caret* will select the next symbol under the caret. Successive calls will extend the selection scope. The following example shows the result of calling *Select Word at Caret* successively.



It's a much faster way to select blocks of code than using the regular *Shift* and arrow keys.

Line Editing

We can delete a whole line with *(⌘-Delete, C-y)*, duplicate it with *Edit → Duplicate Line (⌘-d, C-d)*, and move it up or down with *Code → Move Line Up (S-⌥-↑, S-A-↑)* and *Code → Move Line Down (S-⌥-↓, S-A-↓)*, respectively. More interestingly, we can move an entire code block up or down with *Code → Move Statement Up (⌘-S-↑, S-C-↑)* and *Code → Move Statement Down (⌘-S-↓, S-C-↓)*. It works with blocks of code from a single line to entire functions, methods, or classes. In the following example, I move the code block starting with `for` up a line:

```
def stretch_interval(self, factor):
    intervals = [x + factor for x in self.intervals()]
    note = copy.copy(self[0])
    result = NoteSeq([note])
    for i in intervals:
        note = note.transposition(i)
        result.append(note)
    return result
```

```
def stretch_interval(self, factor):
    intervals = [x + factor for x in self.intervals()]
    note = copy.copy(self[0])
    for i in intervals:
        note = note.transposition(i)
        result.append(note)
    result = NoteSeq([note])
    return result
```

The action *Edit → Join Lines (S-C-j)* must be one of the biggest time-savers in PyCharm. I cannot tell you how much time I spent joining lines manually before I discovered this action. The beauty is that *Join Lines* is smart and knows when characters need to be removed or added to make the code valid.

```
numbers = [1,
           2,
           3,
           4]
```

```
numbers = [1, 2, 3, 4]
```

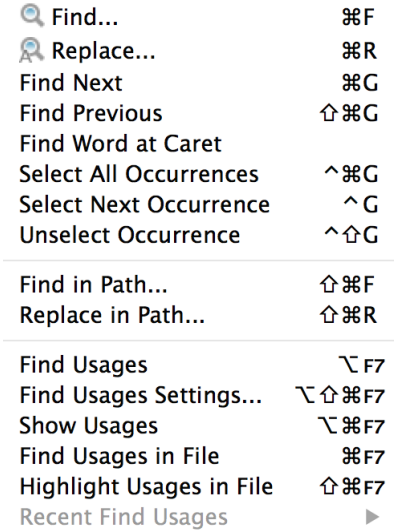
Join Lines also works with string literals:

```
name = "foo" \
       "bar" \
       "bla"
```

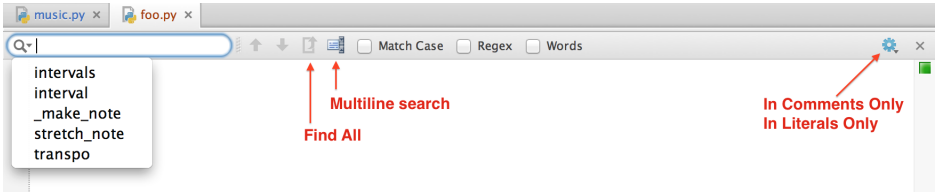
```
name = "foobarbla"
```

Find Code

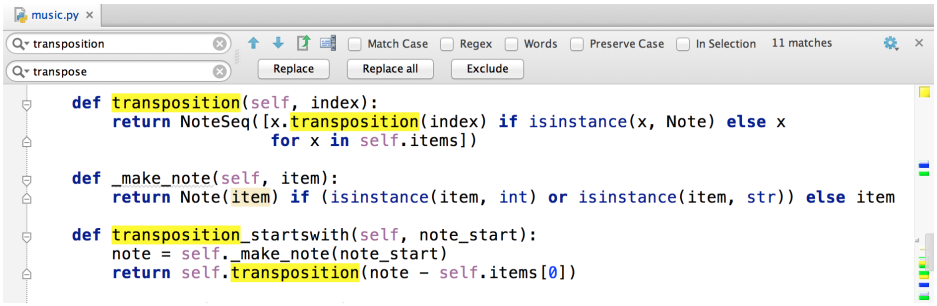
PyCharm has many options that help you find things in the source code. The most basic are, as expected, find (*Edit → Find → Find… (⌘-f, C-f)*) and replace (*Edit → Find → Replace… (⌘-r, C-r)*).



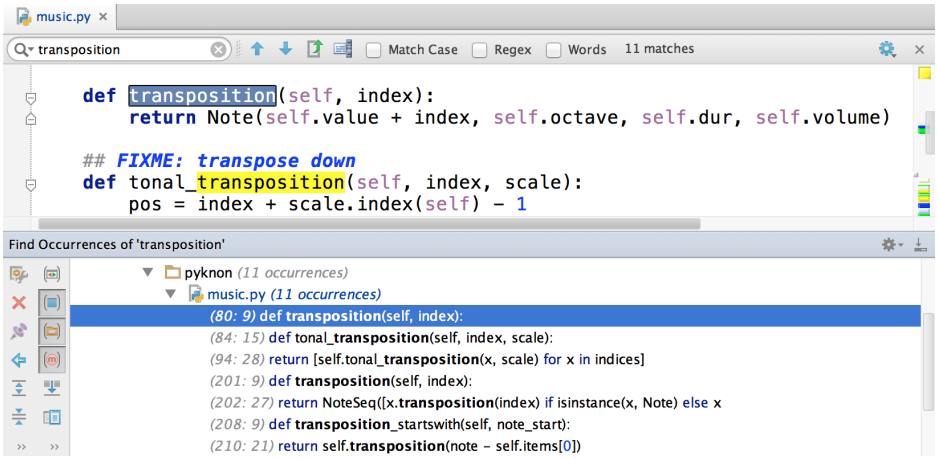
While searching, we can include past searches by clicking on the search icon on the left, activate multiline search, activate case sensitive search, use regular expressions, search for words, or search only in comments or literals.



The *Replace* action has similar options. In addition, we can use *Preserve Case* to preserve the case of the replacement's first letter (for example, if we replace `NoteSeq` with `notesequence`, the result will be `Notesequence`) and replace items in the selected text. We can also replace all items at once (*Replace all*) or one by one (*Replace*) while skipping the items we don't want to replace (*Exclude*).



We can also see all matches at once by clicking on the *Find All (⌥-F7, A-F7)* button:

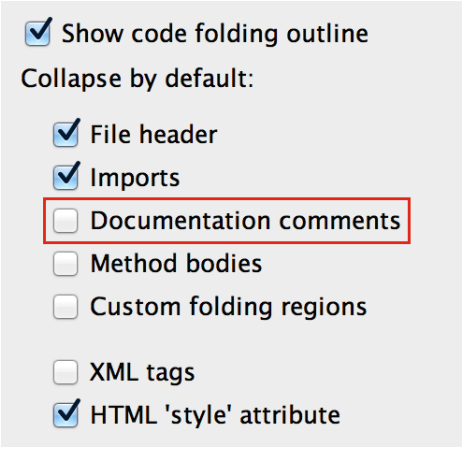


PyCharm keeps the last item searched in case we need it. We can go to the next or previous match with *Edit → Find → Find Next / Move to Next Occurrence (⌘-g, F3)* and *Edit → Find → Find Previous / Move to Previous Occurrence (⌘-S-g, S-F3)*, respectively, even if we dismissed the find/replace dialog.

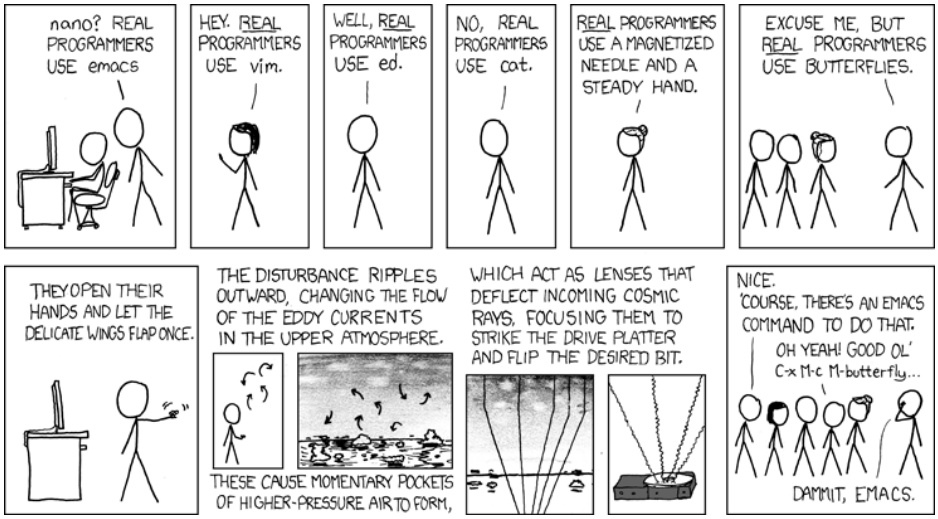
One of my favorites features is the ability to see where a function or method is being called (*Edit → Find → Find Usages*). In the section *Refactoring*, we can see how it can be used for refactoring.

Code Folding and Regions

In PyCharm, we can fold blocks such as classes, methods, and functions, and we can also create our own foldable regions. PyCharm will fold some items such as file headers and imports by default, and we can change this behavior in *Settings* → *Editor* → *General* → *Code Folding*. I like to have documentation strings folded by default as well:



Unless you have lived in the woods for the past twenty years and used butterflies to write programs, you know how code folding works—you click on the - (minus) button to fold a code block and click on the + (plus) button to expand it.



© Randall Munroe, in xkcd, Real Programmers

```
def inversion_startswith(self, note_start):...

def harmonize(self, interval=3, s  e=3):
    return [NoteSeq(note.harmonize(self, interval, size)) for note in self]
```

We can also fold or expand a code block by going to *Code → Folding → Collapse (⌘-, C-,-)* and *Code → Folding → Expand (⌘-+, C-+)*, respectively. We can fold or expand all code blocks by going to *Code → Folding → Collapse All (⌘-S–, S-C–)* and *Code → Folding → Expand All (⌘-S-+, S-C-+)* and fold and expand the docstrings by going to *Code → Folding → Collapse doc comments* and *Code → Folding → Expand doc comments*.

Finally, we can fold an arbitrary code selection by going to *Code → Folding → Fold Selection / Remove Region (⌘-., C-.)*.

```
            elif thing.type == 'trackName':
                event = MIDIEvent()
                event.type = "TrackName"
                event.time = thing.time * TICKSPERBEAT
                event.trackName = thing.trackName
                event.ord = 0
                self.MIDIEventList.append(event)
```

After that, PyCharm will create a region for the code selected, and it will be available as a regular code block:

```
            elif thing.type == 'trackName':
                event = MIDIEvent()
                event.type = "TrackName"
                event.time = thing.time * TICKSPERBEAT
                event.trackName = thing.trackName
                event.ord = 0
                self.MIDIEventList.append(event)
```

When some code is folded, we can hover the mouse over the ellipses to see the content of the folded region without needing to unfold it:

```
            elif thing.type == 'programChange':
...
                event = MIDIEvent()
                event.type = "ProgramChange"
                event.time = thing.time * TICKSPERBEAT
                event.programNumber = thing.programNumber
                event.channel = thing.channel
                event.ord = 1
                self.MIDIEventList.append(event)
            elif thing.type == 'SysEx':
...
```

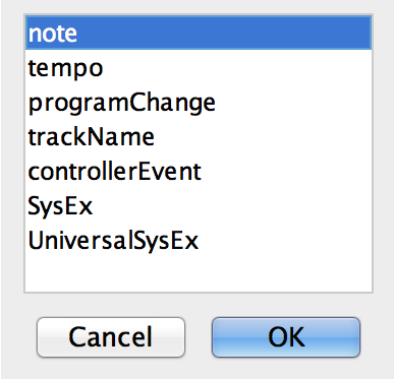
Another way to create custom regions is by surrounding a code block with `#region <description>` and `#endregion`:

```
        #region controllerEvent
        elif thing.type == 'controllerEvent':
            event = MIDIEvent()
            event.type = "ControllerEvent"
            event.time = thing.time * TICKSPERBEAT
            event.eventType = thing.eventType
            event.channel = thing.channel
            event.paramerter1 = thing.parameter1
            event.ord = 1
            self.MIDIEventList.append(event)
        #endregion
```

This will create a foldable region, and it will show the region description when we fold it:

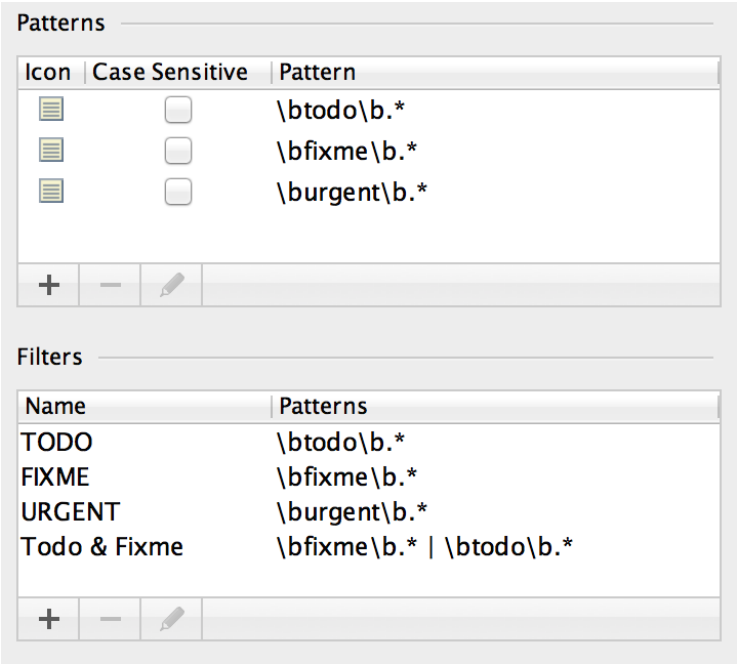
```
    for thing in self.eventList:
        note

        tempo

        programChange

        trackName

        controllerEvent
```

We can navigate to a custom region in a file by going to *Navigate → Custom Folding…* (*⌘-⌥-, A-C-.*) and choosing the region description. Sadly, there's no way to list all regions in a project.

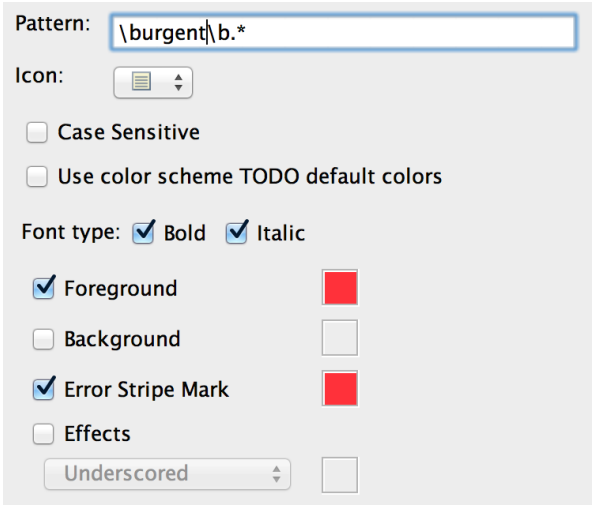


TODO Items

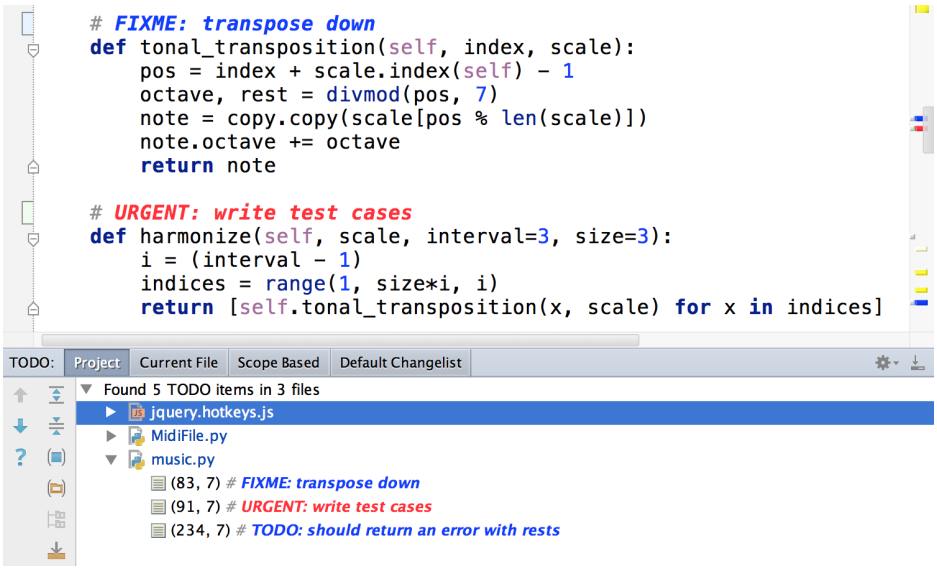
Some programmers like to add TODO items as comments in the source code. PyCharm has support to quickly find and filter these items. We can define our TODO keywords in *Settings* → *Editor* → *TODO*. In the following example, I defined the keywords `todo`, `fixme`, and `feature`. We can also define filters to show only the items we are interested in.



When we create a new filter, we can define the pattern and the color scheme. In the following example, I created a red color scheme that is different from the default blue one:

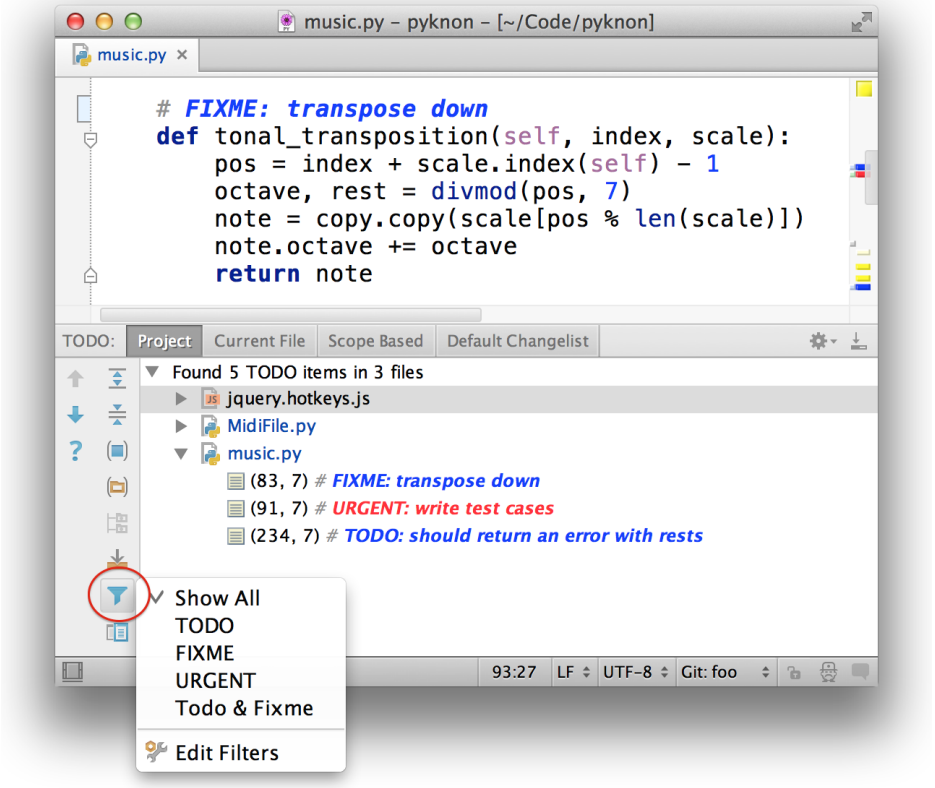


We can see the TODOs by going to *View → Tool Windows → TODO (⌘-6, A-6)* and select whether we want to see the TODOs for the whole project, for the current file, or in a particular scope.




```
    # FIXME: transpose down
    def tonal_transposition(self, index, scale):
        pos = index + scale.index(self) - 1
        octave, rest = divmod(pos, 7)
        note = copy.copy(scale[pos % len(scale)])
        note.octave += octave
        return note

    # URGENT: write test cases
    def harmonize(self, scale, interval=3, size=3):
        i = (interval - 1)
        indices = range(1, size*i, i)
        return [self.tonal_transposition(x, scale) for x in indices]
```

We can filter the TODOs by clicking on the filter tool. We can click on the TODO item to jump to that place in the source code.



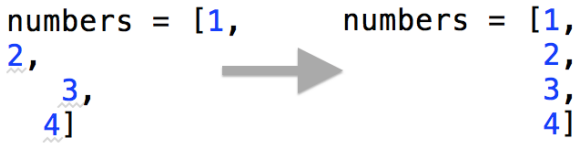
Tidying Up the Code

The action *Edit → Fill Paragraph* will redistribute the line breaks in the current paragraph to fill the maximum line width (120 by default in PyCharm, see section *Initial Configuration*). It works with comments, strings, and text files. In the following example, we can see the comments before and after being filled.

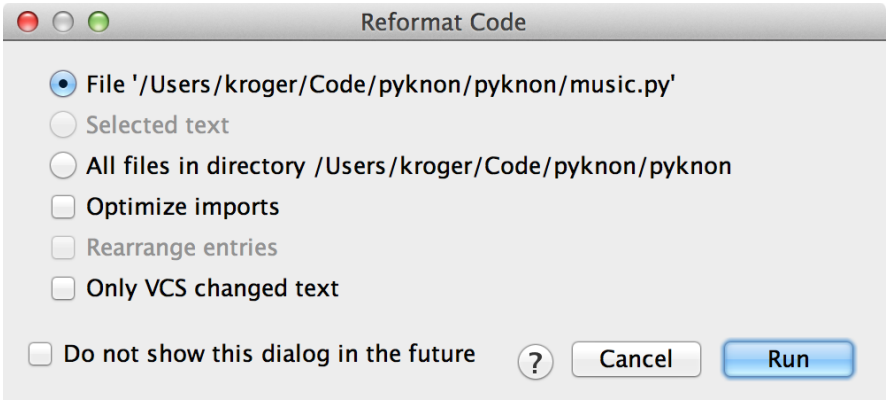
```
def __radd__(self, other):
    if isinstance(other, NoteSeq):
        # This should
        # never be called because the other NoteSeq should
        # handle the concatenation,
        # but it's here for completness sake
        return NoteSeq(other.items + self.items)
```

```
def __radd__(self, other):
    if isinstance(other, NoteSeq):
        # This should never be called because the other NoteSeq should
        # handle the concatenation, but it's here for completness sake
        return NoteSeq(other.items + self.items)
```

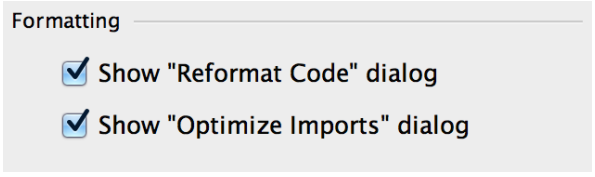
I don't need to tell you how important indentation is in Python. PyCharm makes it really easy to deal with indentation; we select the code and type *Tab* to indent and *Shift-Tab* to unindent one level. Some people like spaces; others like tabs. We can fix badly indented code quickly by going to *Edit* → *Convert Indents* → *To Spaces* or *Edit* → *Convert Indents* → *To Tabs* and we can *Code* → *Auto-Indent Lines (⌥-C-i, A-C-i)*:



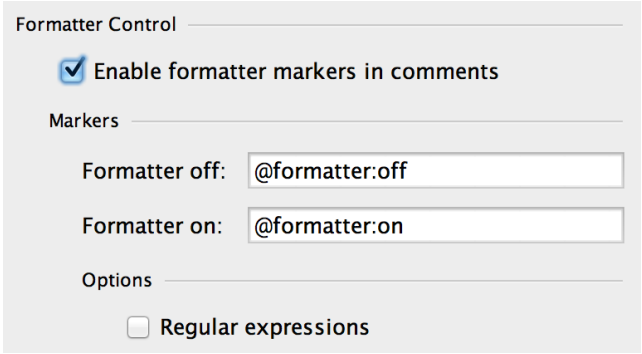
To be honest, I don't use *Auto-Indent Lines* much because PyCharm has something much more powerful. The action *Code* → *Reformat Code…* *(⌘-⌥-l, A-C-l)* will fix the code to match the current style. When we call this action, it will open a dialog asking for the scope where the formatting will be done. We can choose the current file, the selected text, all files in the project, and even the code that has changed since the last commit.



If you click the option "Do not show this dialog in the future," the reformat code dialog will not be shown ever again. This option can be restored in *Settings → Editor → General*.



Sometimes, it is necessary to format the code in a nonstandard way. A classical example is the list of url patterns in Django `urls.py`. It looks too weird if we use the default indentation. We can tell PyCharm to leave the formatting of a block of code alone. We need to make sure that "Enable formatter markers in comments" is selected in *Settings → Editor → Code Style*:



Then we surround the code block with `@formatter:off` and `@formatter:on`, inside comments:

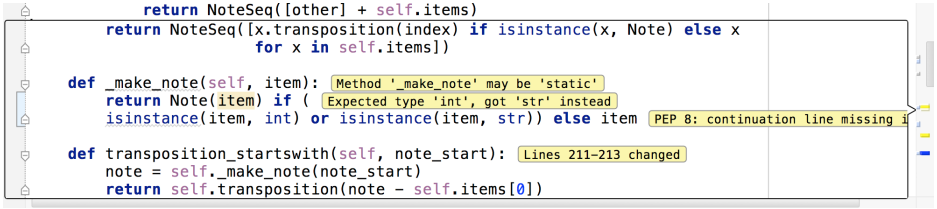
```
# @formatter:off
urlpatterns = patterns('',
    url(r'^$', 'analysis.views.home', name='home'),
    url(r'^login/$', 'analysis.views.login_user', name='login_user'),
    url(r'^range/$', 'analysis.views.show_range', name='range'),
    url(r'^dashboard/$', 'analysis.views.dashboard', name='dashboard'),
    url(r'^admin/', include(admin.site.urls)),
)
# @formatter:on
```

Lens Mode

On the right, with the scrollbar, we have the validation sidebar. It shows warning and error strips:

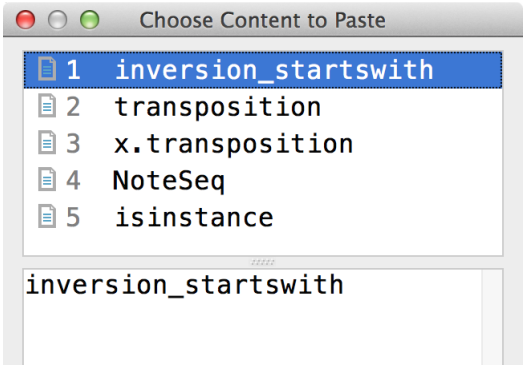
```
        return NoteSeq([x.transposition(index) if isinstance(x, Note)
                        for x in self.items])

    def _make_note(self, item):
        return Note(item) if (isinstance(item, int) or isinstance(ite

    def transposition_startswith(self, note_start):
        note = self._make_note(note_start)
        return self.transposition(note - self.items[0])
```

When we hover the mouse over the warning and error strips, PyCharm shows a fragment of the code annotated with the appropriate message. It's very useful to see problems in the code without having to scroll back and forth.



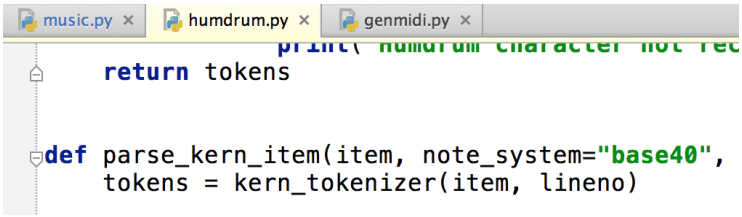
Miscellaneous

As with any modern editor, PyCharm has the usual copy and paste commands. In addition, it keeps a history of copied items that we can access at *Edit → Paste from History… (⌘-S-v, S-C-v)*. It opens a window where we can choose the item to paste by either typing or double clicking the number on the left. PyCharm keeps five items in the clipboard, although we can change this in *Settings → Editor → General*.

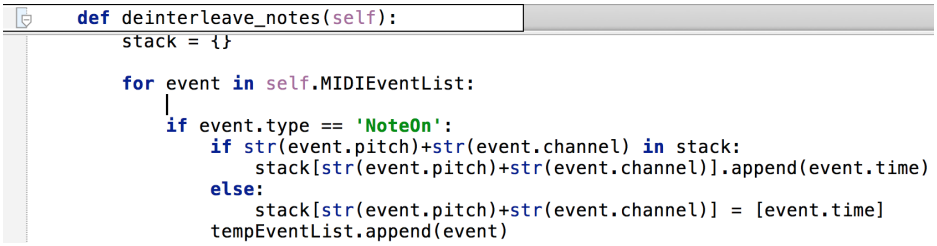


Although PyCharm is a project-oriented IDE, we can open files outside the current project by going to *File → Open…*. It will show the tab in a different

color, to emphasize that the file is not part of the project. As usual, we can change the color in *Settings → Appearance & Behavior → File Colors*.



Sometimes, we are focused deeply in fixing a class, method, or function whose name and signature is not visible on the screen. The action *View → Context Info (S-C-q, A-q)* shows this information painlessly, without making us scroll to the beginning of the declaration:

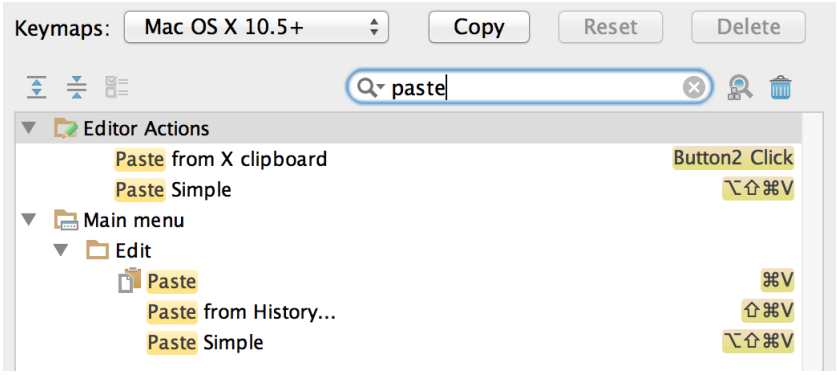


Shortcuts

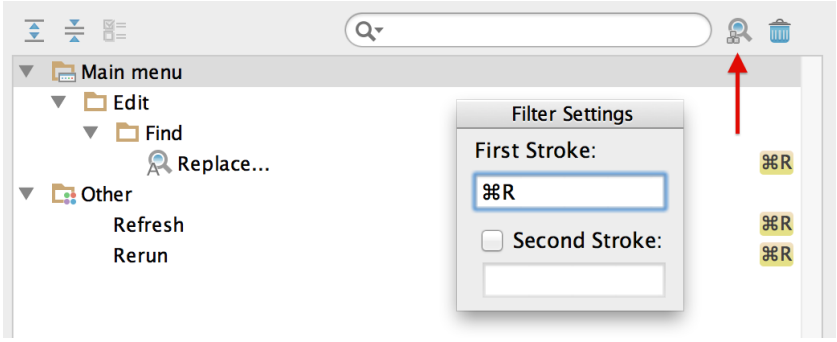
PyCharm is no Vim, but you can do a lot by using only the keyboard. We can assign shortcuts to hundreds of actions in *Settings → Appearance & Behavior → Keymap*. I change the default keymap heavily to suit my preferences, and I recommend you do the same. Check the official Mac and the Windows/Linux reference cards for the default shortcuts.

To add a new shortcut, we go to *Settings → Appearance & Behavior → Keymap*

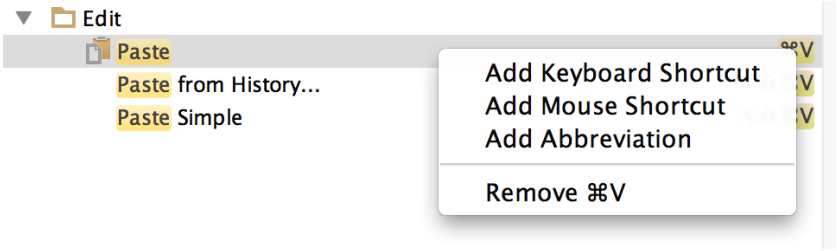
and select the desired action. The search box is very handy for narrowing down the actions. In the following example, I will assign the shortcut ⌘-*y* to the *Paste* command.



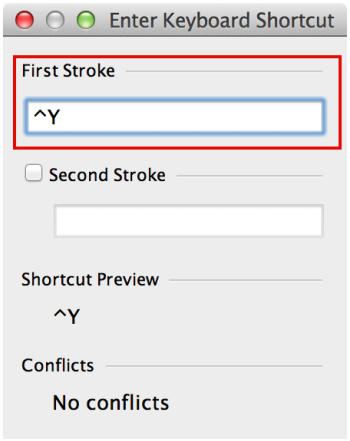
We can also find an action by searching for a shortcut:



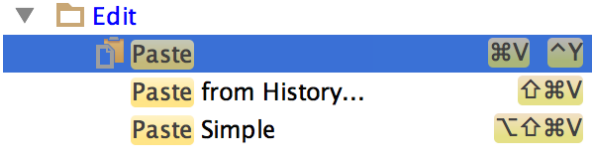
When we double-click an action, we can add a keyboard or mouse shortcut, or an abbreviation.



To enter a new shortcut, we type the keystroke on the input widget and PyCharm will recognize it. We can also enter emacs-like shortcuts with two strokes such as *C-c C-n*.



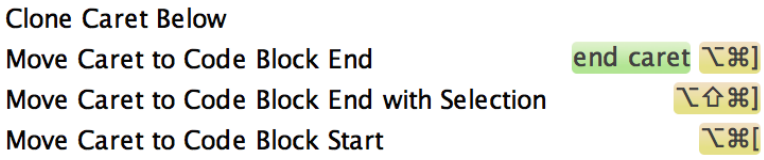
Now, we can run the action *Paste* with either ⌘-*v* or *C-y*:



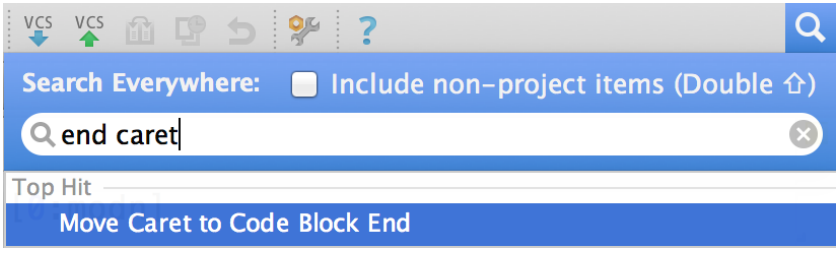
We can also define mouse shortcuts such as *Control-Click*. PyCharm has shortcuts that use a middle button, which some laptops lack. On the Mac,

we can use something like BetterTouchTool to emulate the middle button, but I prefer to remap the PyCharm keys that use the middle button.

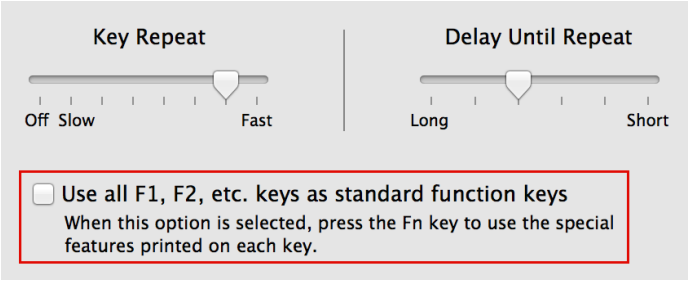
In PyCharm , abbreviations are metadata that are attached to a keyboard or mouse shortcut. These metadata can be used in the Search Everywhere feature (see *Search Everywhere*). It's a useful feature if you find yourself searching for an action but can't remember its name or it has a name similar to other actions. For example, we can define an abbreviation named "end caret" for the action "Move Caret to Code Block End."



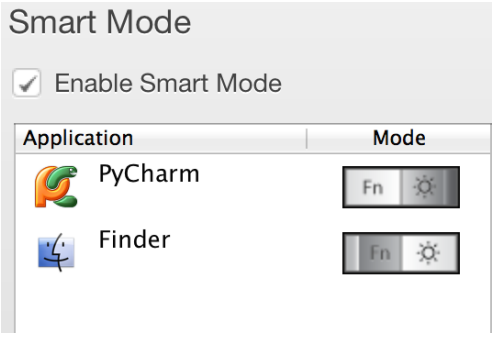
And now we can search for "end caret" on Search Everywhere.



PyCharm assigns lots of actions to function keys such as *F1* and *F2*. As you probably know, the top keys on the Mac work as media keys by default and as function keys by pressing the *Fn* key. We can swap this behavior in *System Preferences → Keyboard*.



I like to be able to control the sound volume and change the brightness without having to reach for the *Fn* key, but I also like to use the function keys while programming without reaching for the *Fn* key (I'm lazy; what can I say?). I use Palua to switch between the media and function keys. You can switch with a global key or you can configure Palua to switch automatically when using a specific application. I use the media keys for all applications (the default) and configure Palua to use the Function keys for Xcode and PyCharm.



On Windows and Linux the way the Function keys work will depend on the computer model you have. On Windows, you can use AutoHotKey to remap your keys; the command #IfWinActive applies shortcuts to specific applications. On Linux, you can use AutoKey, xbindkeys, or good ol' XModmap, depending on your needs.

Finding Your Way in the Source Code

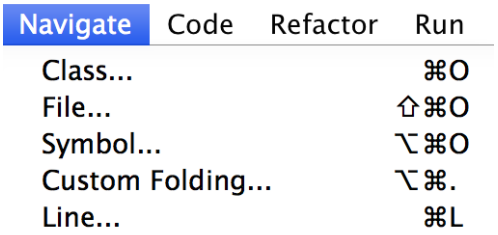
PyCharm's power starts when you master its navigation commands. It allows you to go quickly to classes, methods, functions, files, and variables. To jump to the declaration of a symbol under the cursor, go to *Navigate → Declaration (⌘-b, C-b)* or *(⌘-Click, C-Click)*. You can navigate back and forward with *Navigate → Back (⌘-[, A-C-Left)* and *Navigate → Forward (⌘-], A-C-Right)*, respectively. For example, in the following example, we want to see the definition for `MIDIFile`.

```
self.number_tracks = number_tracks
self.midi_data = MIDIFile(number_tracks)
```

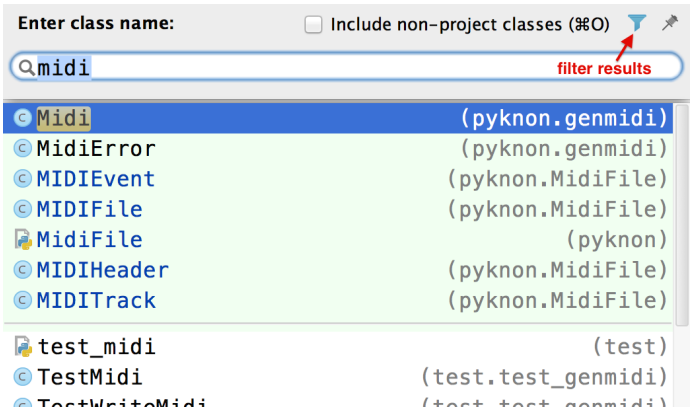
PyCharm jumps to its declaration even if it's defined in a file in a separate package. After reading its definition, we can go back to the original source code with *Navigate → Back*. This allows us to get acquainted with a new code base very quickly.

```
class MIDIFile:
    """Class that represents a full, well-formed MIDI pattern...."""

    def __init__(self, numTracks, removeDuplicates=True,  deinterleave=True):
        """..."""
        self.header = MIDIHeader(numTracks)

        self.tracks = list()
        self.numTracks = numTracks
        self.closed = False

        for i in range(0,numTracks):
            self.tracks.append(MIDITrack(removeDuplicates,  deinterleave))
```

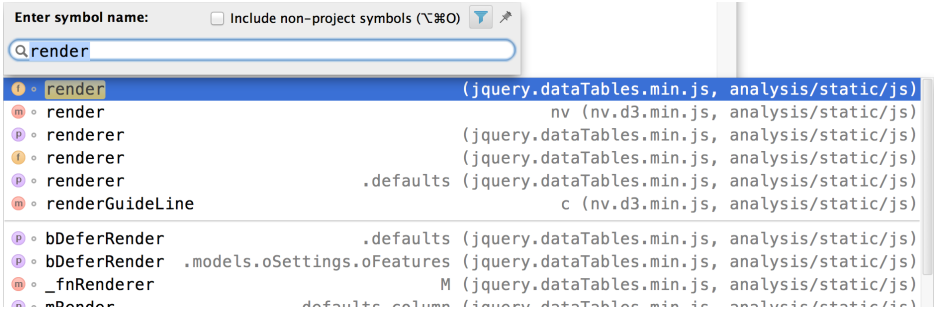
The mechanism to navigate to a class, file, or symbol is very similar, although every type of navigation has a different command and shortcut. To navigate to a class, file, or symbol, we go to *Navigate → Class... (⌘-o, C-n)*, *Navigate → File... (⌘-S-o, S-C-n)*, and *Navigate → Symbol... (⌘-⌥-o, S-A-C-n)*, respectively.



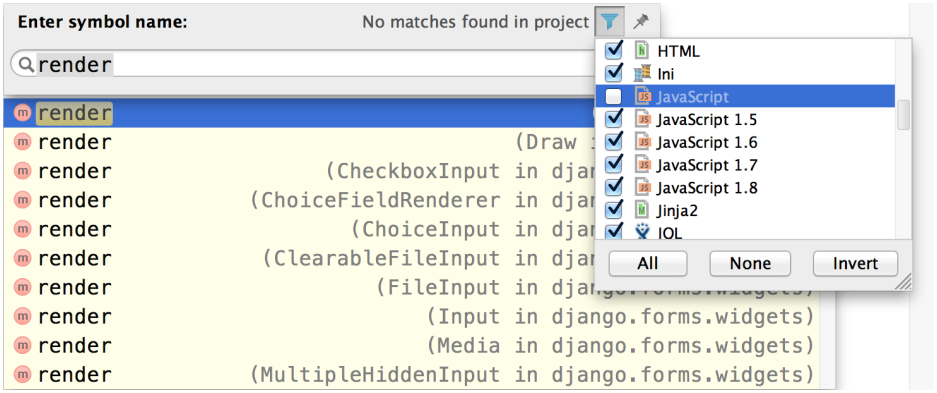
When using these navigation actions, we are presented with a dialog where we can type a substring and choose to include items outside the project and filter the results. For instance, if we search for the substring "midi" in the *Navigate → Class…* action, PyCharm will show all classes with this string.



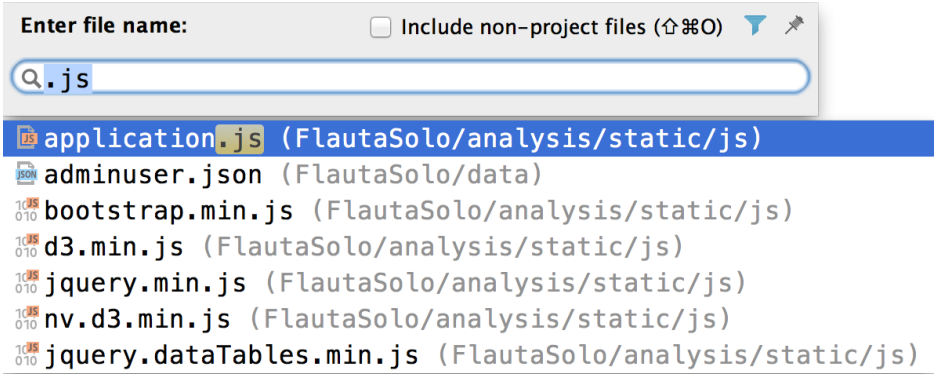
In large projects, it may be useful to filter the types of files we are searching. We can do this by clicking on the funnel icon and clicking on the types of files we want to remove from the search. In the following example, we are trying to find the Python method `render`, but there are too many related results in JavaScript:



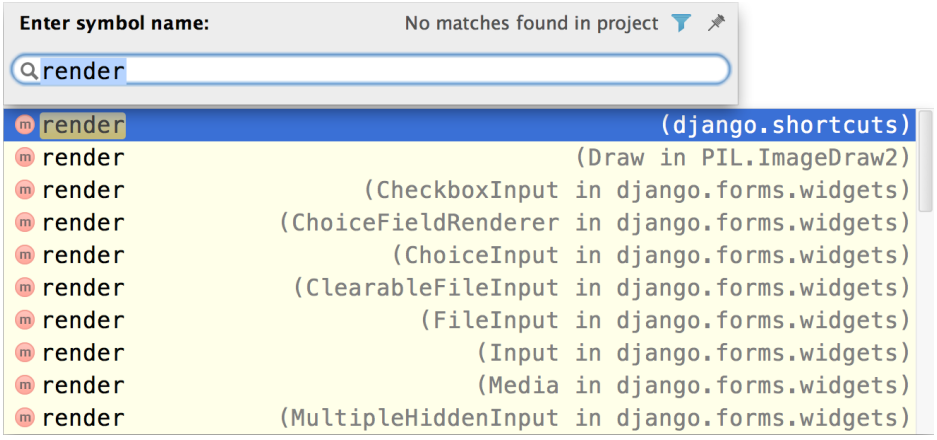
After we filter JavaScript from the search, it's easier to find the Python method we want:



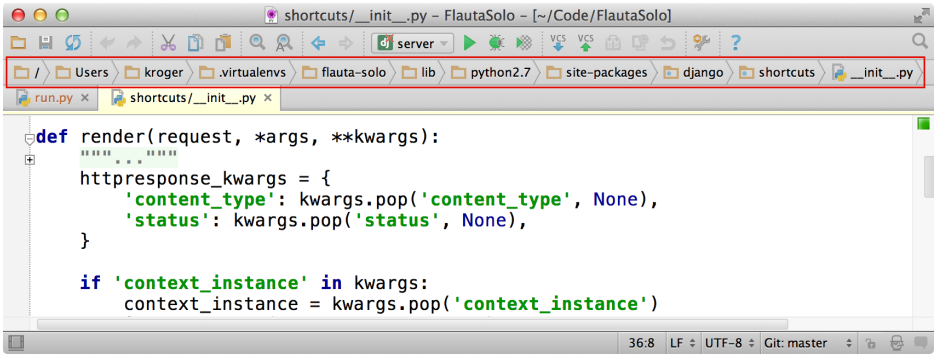
When navigating to files, we can enter part of the filename, including the file extension. For instance, if we want to open a JavaScript file in a Django project but don't remember its name (it happens to me all the time), we can search for `.js`.



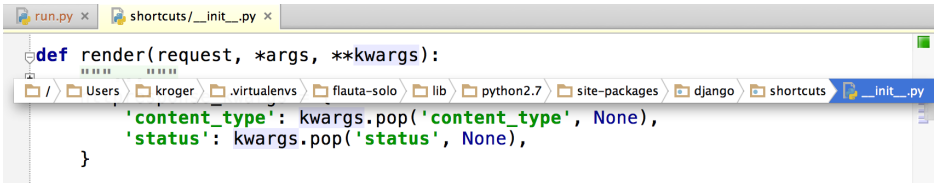
These features allow us to navigate the source code very quickly. Suppose we are working on a Django project and want to see how the function `render` is implemented. We go to *Navigate* → *Symbol...*, type "render," and pick the first option (we can see it's defined in `django.shortcuts`).



The Navigation Bar is useful to quickly see where a file is located in the directory structure (in our example, `django.shortcuts` is defined on `django/django/shortcuts.py`).



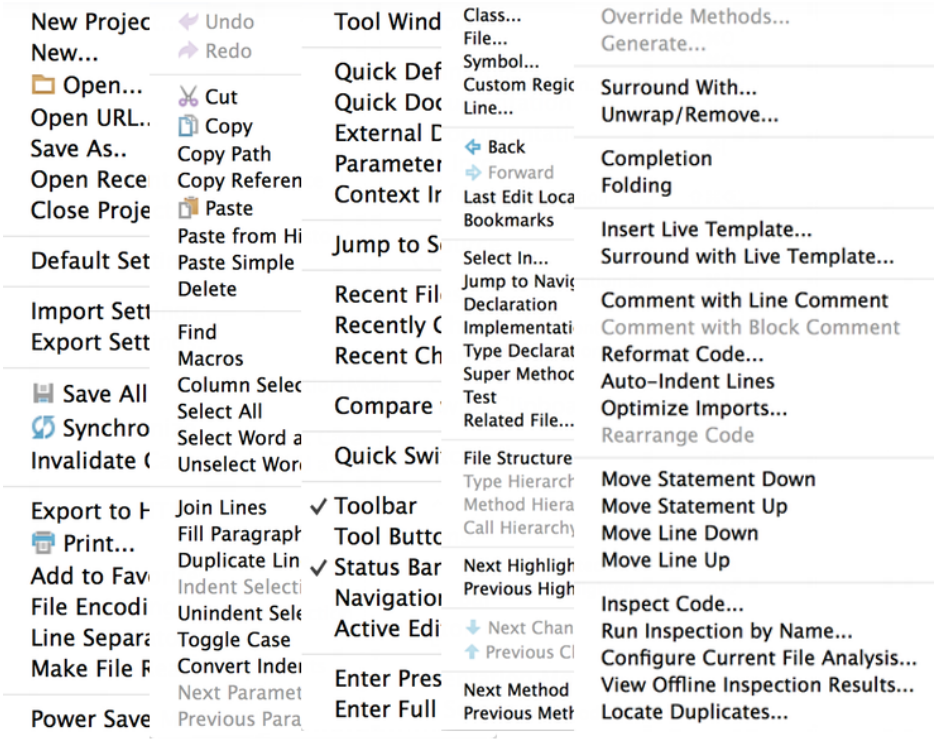
We can show the Navigation Bar quickly with *Navigate → Jump to the Navigation Bar (⌘-↑, A-Home)*, if it's hidden.



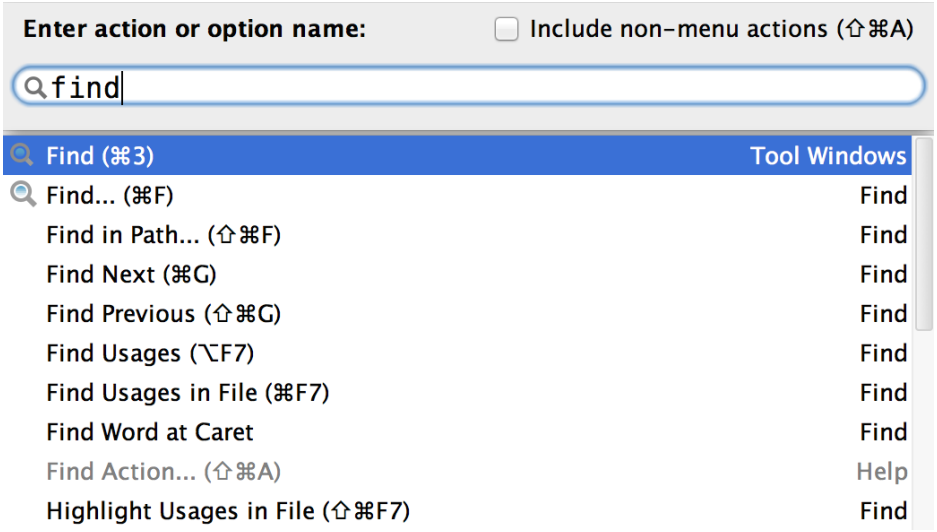
Starting with PyCharm 3.1, there's a new feature called Search Everywhere that combines all these navigation commands in one. See section *Search Everywhere* for more details.

Finding Commands

PyCharm has loads of commands and options. Although this is what makes it so powerful and useful, it's easy to feel overwhelmed with so many commands.



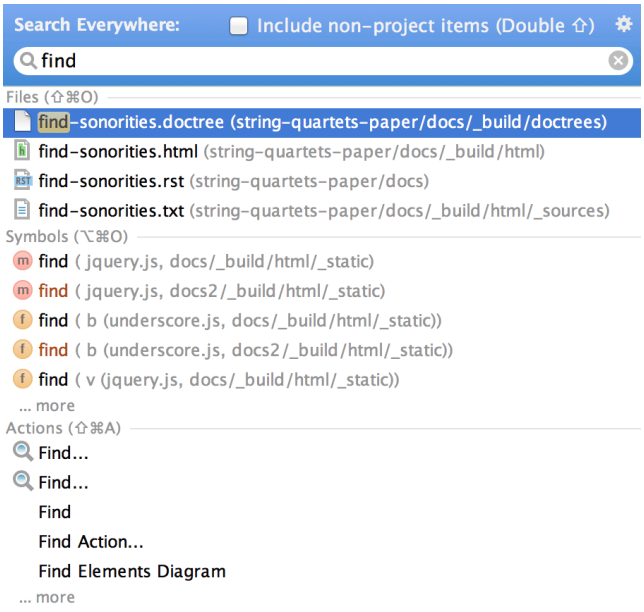
We can search any PyCharm command, including the ones not available from the menu, with *Help → Find Action (⌘-S-a, S-C-a)*. We can invoke the desired command by clicking on it or typing *Enter* when the desired action is selected. In the following example, we can see the actions that include the string "find."



If we type the shortcut again *(⌘-S-a, S-C-a)*, the option "Include non-menu actions" will be selected. Therefore, if we want to search some action and include non-menu actions, we can just type the shortcut twice.

Search Everywhere

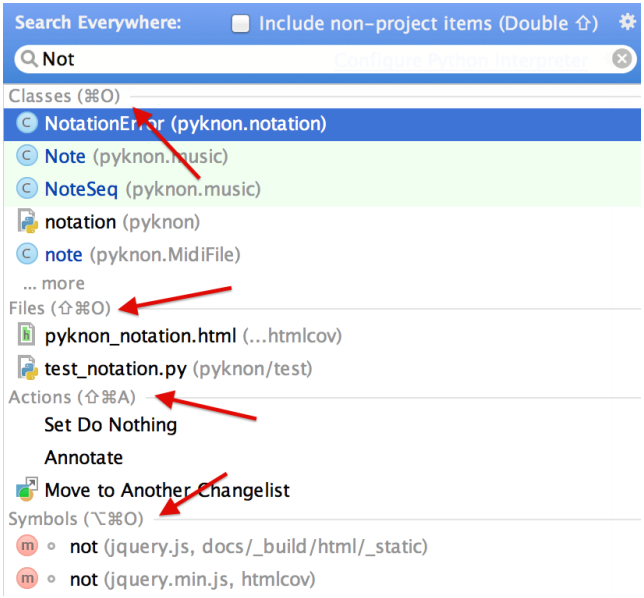
As we have seen, PyCharm has separate commands to search files, classes, functions, symbols, and actions. Search everywhere combines these commands into one.



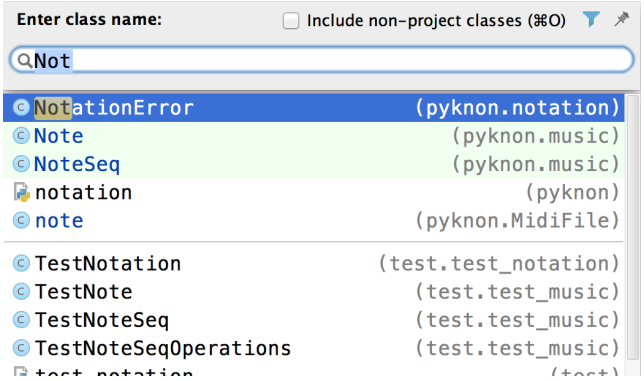
We can open Search Everywhere by clicking on the search icon on the upper-right corner or by pressing *Shift* twice (that is, *Shift-Shift*).

As with Find Action, we can type *Shift-Shift* again to select the "Include non-project items" option.

We can switch from Search Everywhere to the specific navigation command by typing the corresponding shortcut. In the following example, we have multiple results (Classes, Files, Actions, and Symbols) while searching for "Not."



If we want to see the other options for Classes, we can type the regular shortcut (⌘-*o, C-n*) and PyCharm will open the dialog, enter the string we are using, and show the completion in one step:

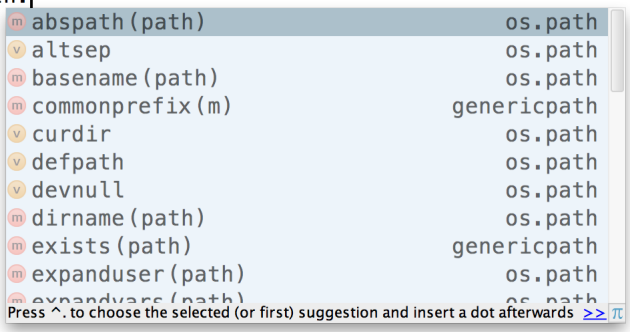


That's all there is to it. I find Search Everywhere a great complement for the other search commands.

Code Completion

Completion in PyCharm is top notch. By default, PyCharm will suggest things while you are typing:

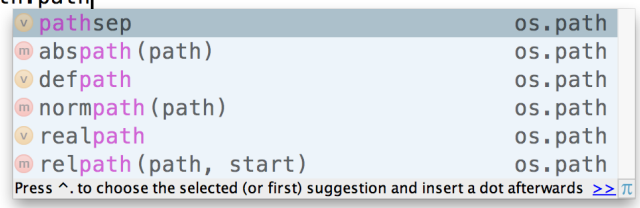
```
import os

os.path.
```



We can disable completion while typing by selecting *File → Power Save Mode* because it can be a little power hungry. We can still use completion by calling it explicitly with *C-Space*; PyCharm will just not suggest things while we type.

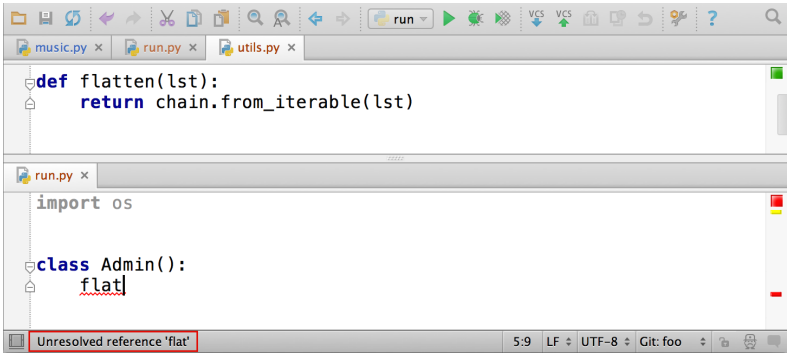
When completing, we can narrow the suggestion list by typing a substring:

```
import os

os.path.path
```

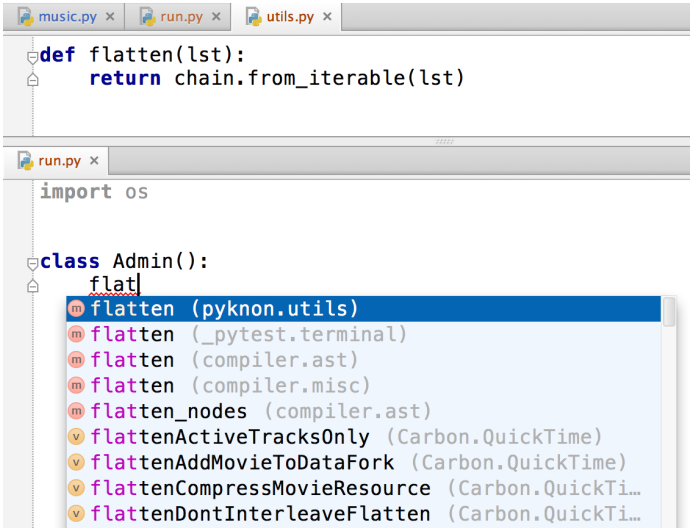


We can type only the first letters of `CammelCaseClasses` or `function_names_with_underscores` and PyCharm will complete it successfully. For example, if we want to autocomplete the `PendingDeprecationWarning` exception, we can just type `PDW`:

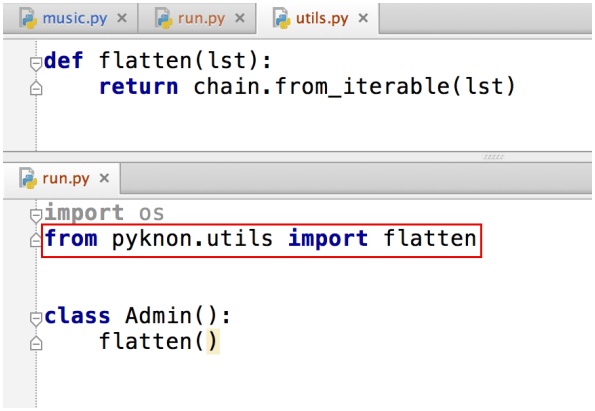
If we type *C-Space* once, PyCharm will try to list the most relevant items, but if it doesn't find any, it will show a "Unresolved reference" message in the status bar:



If we type *C-Space* again (that is, if we type it twice), PyCharm will list every related name it knows. This can be overwhelming, but notice that it is listing the function `flatten` from the `utils.py` file that has not been imported.



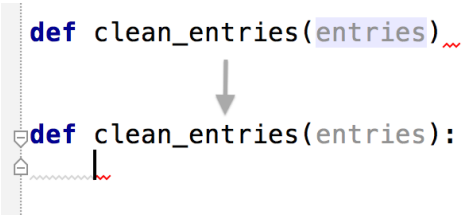
If we complete the completion by typing *Enter*, PyCharm will import `flatten` automatically.



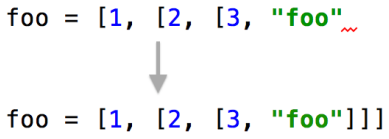
Although PyCharm is smart about completion, we sometimes just want to complete the name of a local variable in one of the opened files. This feature has been available in Emacs and Vim for a long time, and now it's available in PyCharm as well. The manual calls it Hippie Completion , but the actual

command name (that is, the name we will find in *Help → Find Action…*) is "Cyclic Expand Word" (⌥-/, *A-/*) and it works even inside docstrings.

A useful action is *Complete Current Statement* (⌘-*S-Enter, S-C-Enter*). It will insert missing closing parentheses, braces, quotes, and punctuation. For instance, if we type a new function and stop typing right before the closing parenthesis and type the *Complete Current Statement* shortcut, it will insert the closing parenthesis and the ending colon, and insert a new line because we are defining a new function (it works similarly for methods).



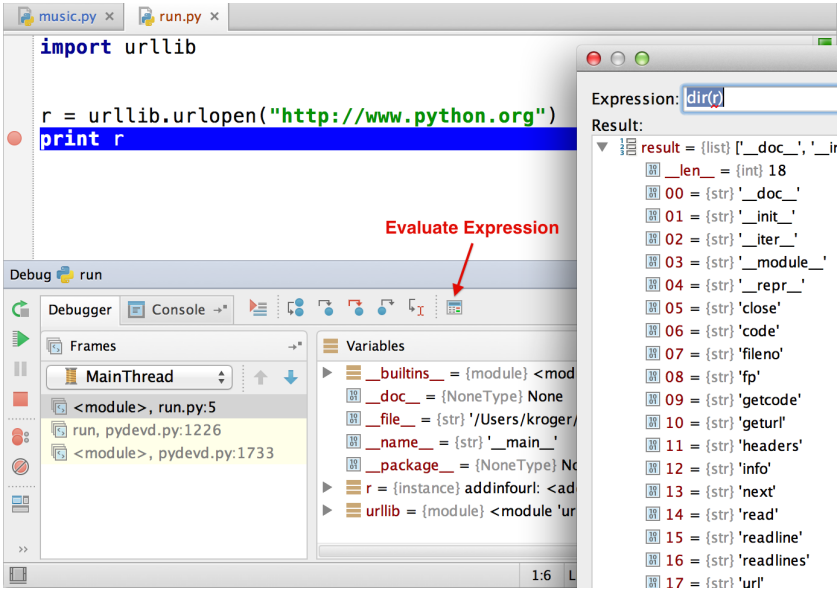
It also works for other Python objects and statements, such as lists and strings:



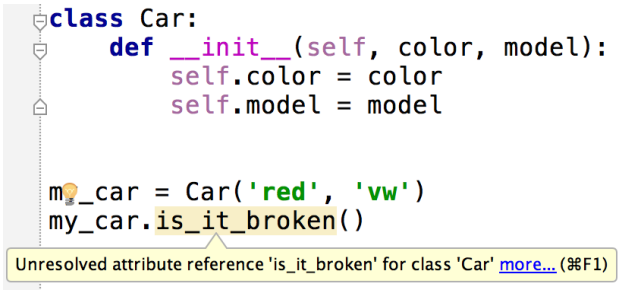
Type Hinting

Completion may not work in some cases when a module doesn't have type hints. If you can't change the module, a possible workaround is to set a breakpoint in PyCharm and evaluate an expression at the breakpoint by clicking on the last icon in the debug toolbar (⌥-*F8, A-F8*). For instance, in the following example, we can't autocomplete the methods in the variable `r` because `urllib.request` doesn't have type hints. In this case, we can evaluate

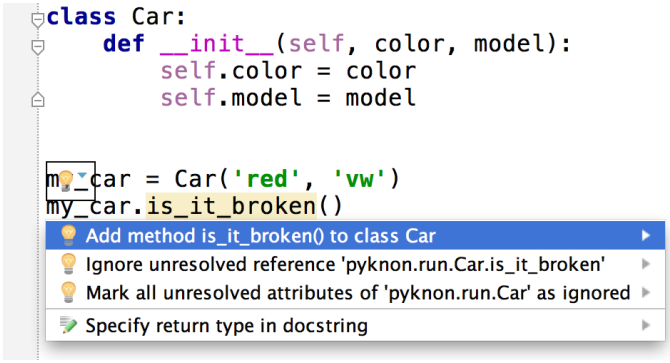
the expression `dir(r)` to see the available methods.



It's useful to add type hinting in your own code. In the following example, PyCharm is smart enough to know that the method `is_it_broken` does not exist:



And, once again, it's smart enough to offer to fix it, either by adding a method or by ignoring it:



If we decide to add a new method, PyCharm will create the method for us in the right place and with the right number of arguments:

```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model

    def is_it_broken(self):
        pass


my_car = Car('red', 'vw')
my_car.is_it_broken()
```

However, in some cases, PyCharm won't know the object type without some extra help. It will not complain if we use an undefined method in `Car` (`send_to_dealership`), where the object `car` is passed as an argument, because PyCharm has no way to know the type of the variable `car`:

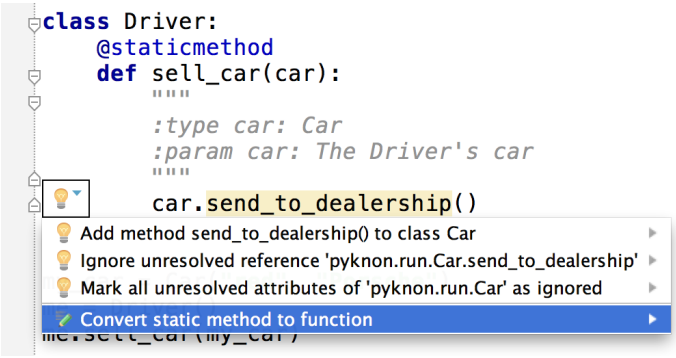
```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model


class Driver:
    @staticmethod
    def sell_car(car):
        car.send_to_dealership()


my_car = Car("red", "Porsche")
me = Driver()
me.sell_car(my_car)
```

If we add a type annotation with `:type`, PyCharm will know the argument's type instantly and will emit a warning:

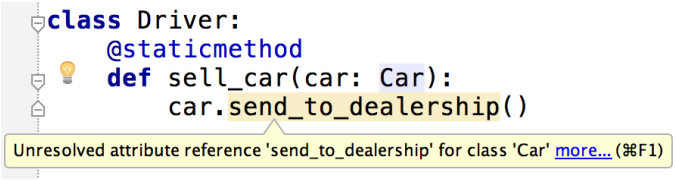
```
class Driver:
    @staticmethod
    def sell_car(car):
        """
        :type car: Car
        :param car: The Driver's car
        """
        car.send_to_dealership()
```

Unresolved attribute reference 'send_to_dealership' for class 'Car' more... (⌘F1)

And, as before, PyCharm will offer to fix the problem, either by creating the missing method or by ignoring the warning:



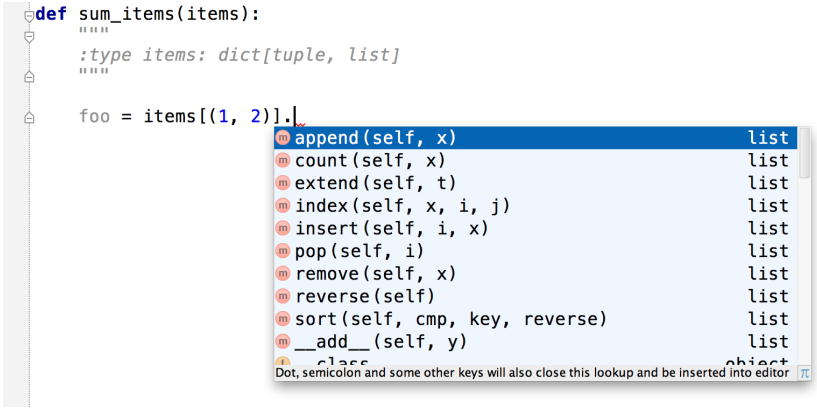
We can also accomplish the same thing by using function annotations (Python 3 only, see PEP 3107):



It's possible to give information about the parameters of a function, return values, local variables, and attributes. Because the syntax for types is not defined by any standard, PyCharm uses its own notation. For instance, we can use `list of str` to indicate a list of strings or `dic[tuple, list]` to notate a dictionary that has tuples as keys and lists as values.

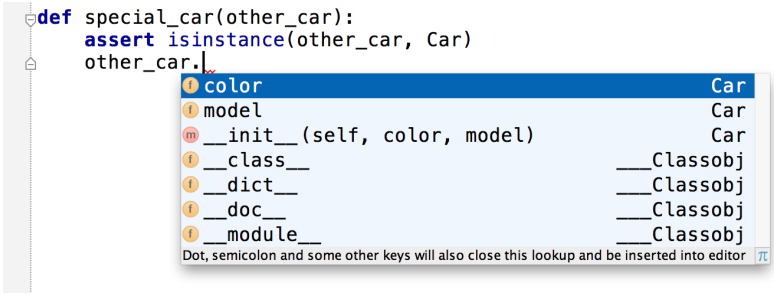
By adding type annotations, PyCharm is able to figure out that `items` is a list of lists and offer the correct completion:

Similarly, it knows that in the following example, `items` is a dictionary with tuples as keys and lists as values:



Check the manual for more information.

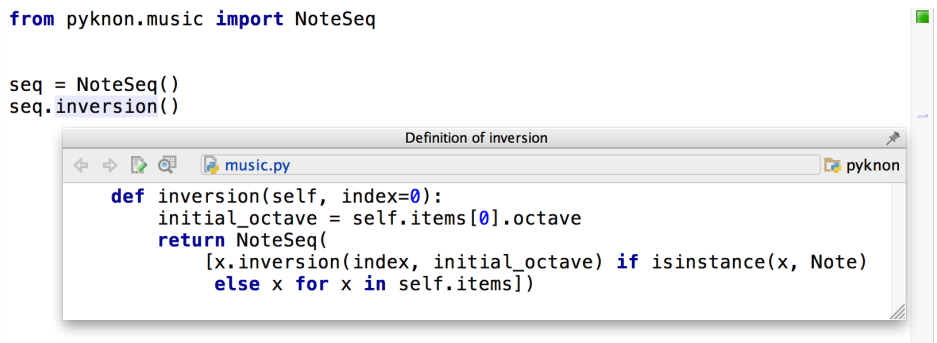
A third way to define the type of a variable is by using `isinstance`. This will force PyCharm to recognize the variable's type.



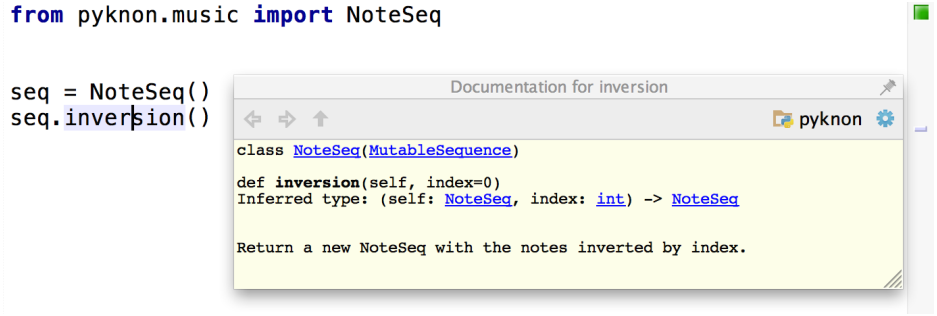
Documentation

PyCharm has four ways to access documentation: Quick Definition, Quick Documentation, External Documentation, and Parameter Info.

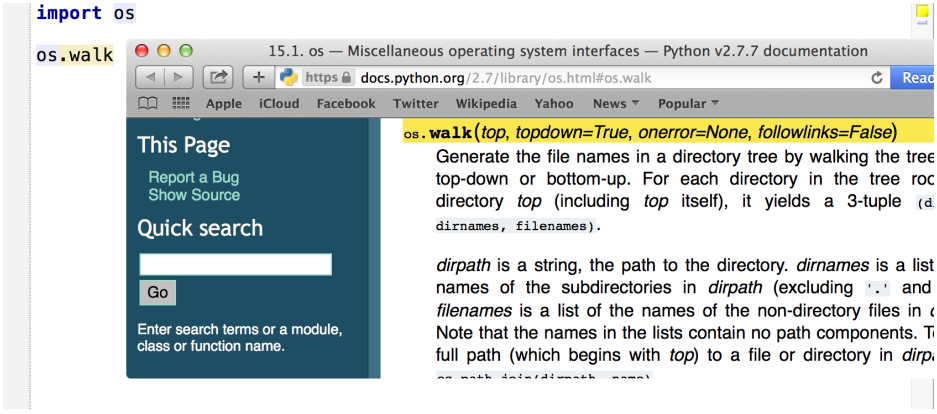
Quick definition, *View → Quick Definition (⌥-Space, S-C-i)*, will show the whole definition of a symbol (class, method, function, etc.), including its documentation. It's useful when we just want to take a quick look at the definition without jumping to it.



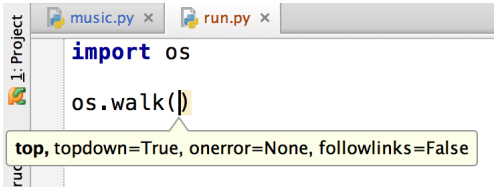
Quick documentation, *View → Quick Documentation (C-j, C-q)*, will show the symbol's documentation and signature:



The action *View → External Documentation (S-F1, S-F1)* opens the documentation in the default browser:

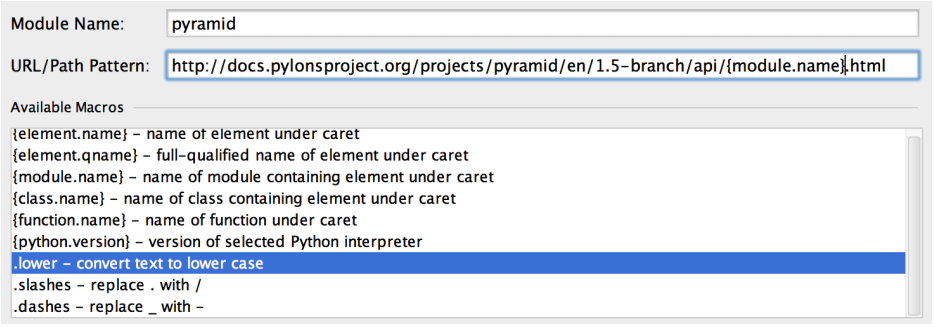


While *View → Parameter Info (⌘-p, C-p)* shows the parameter information for a function or method:



External documentation works out of the box with Python (of course),

PyQt4, PySide, gtk, wx, numpy, scipy, and kivy. We can add more documentation in *Settings* → *Tools* → *Python External Documentation* by adding a module name and a path to the documentation. We can use macros such as `{element.name}` and `{module.name}` in the path. For instance, this is how we would configure PyCharm to open Pyramid's documentation:

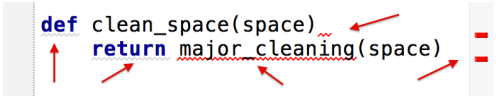


Unfortunately, there's a bug that prevents the macros from being expanded, but I hope it's going to be fixed soon.

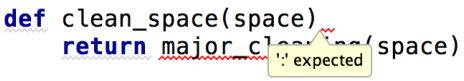
Code Quality

Code Inspection

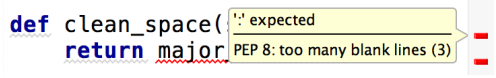
Code inspection is a powerful static analysis for Python, Django, Javascript, CoffeScript, HTML, CSS, and other languages. It will show issues in the code classified by severity, from simple typos to critical errors. Unlike other IDES that shove tons of popups in our face, PyCharm is very gentle about letting us know that something may not be right. In the following example, PEP 8 warnings are underlined in light gray, whereas major errors are underlined in red.



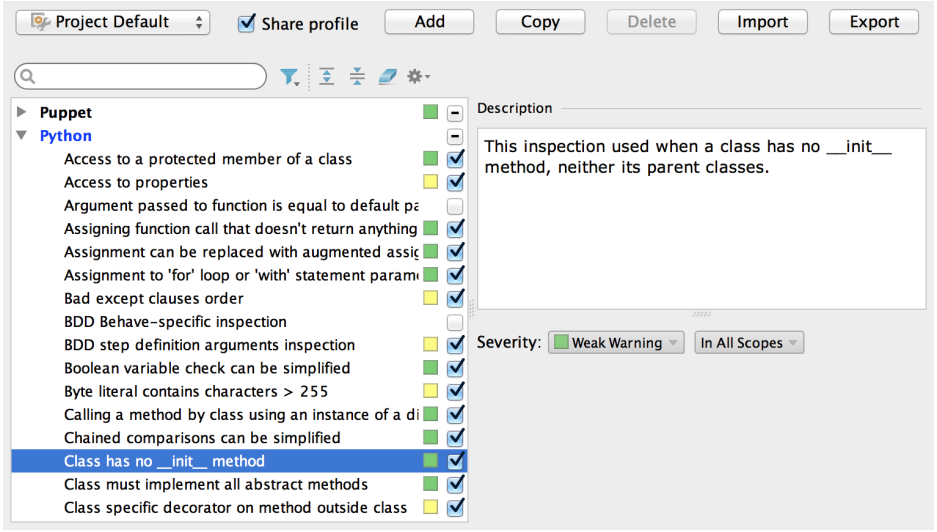
We can get more information by hovering the cursor over the underlined place:



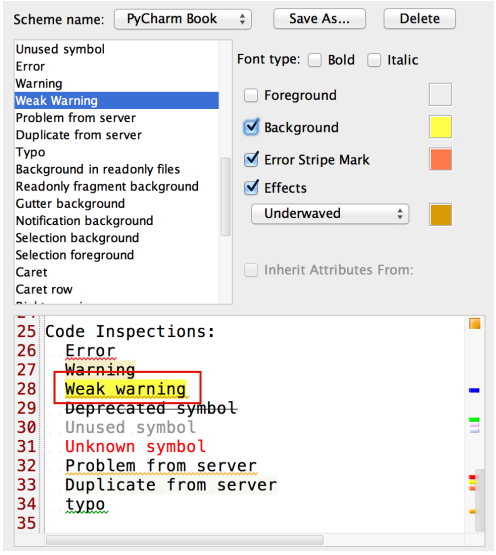
or over the validation sidebar:



If our code style is different than PyCharm's and we don't want to see a warning message, we can enable or disable an inspection in *Settings → Editor → Inspections*. We can also create different inspection profiles. For instance, we could have a tolerant profile with many inspections disabled for prototyping or demonstrations (ofter, when teaching, we want to show things that don't work) and a stricter one for actual development.



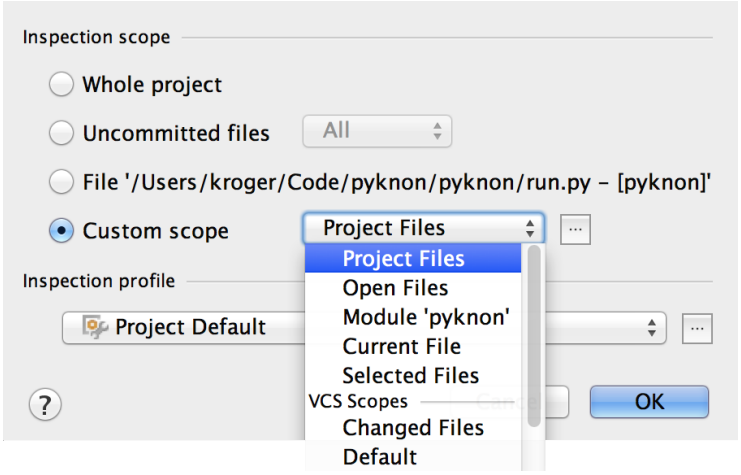
The severity color can be changed in *Settings → Editor → Colors & Fonts → General*. In the following example, I added an excessive yellow background and changed the default stripe mark and underline colors for weak warnings.



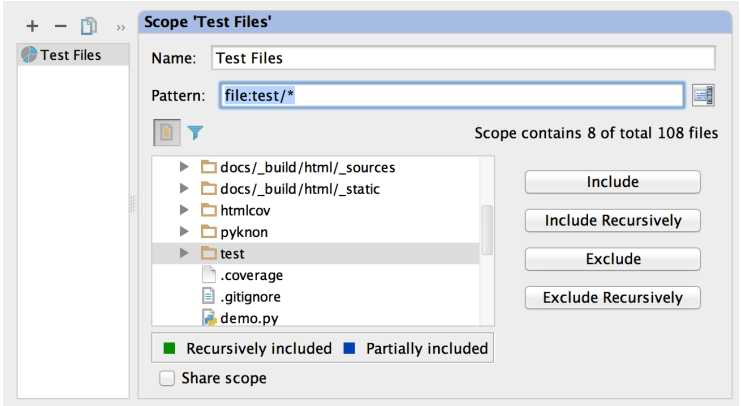
And this is the result:

```
for item in range(5):
    for item in range(20, 25):
        print "Inner", item
    print "Outer", item
```

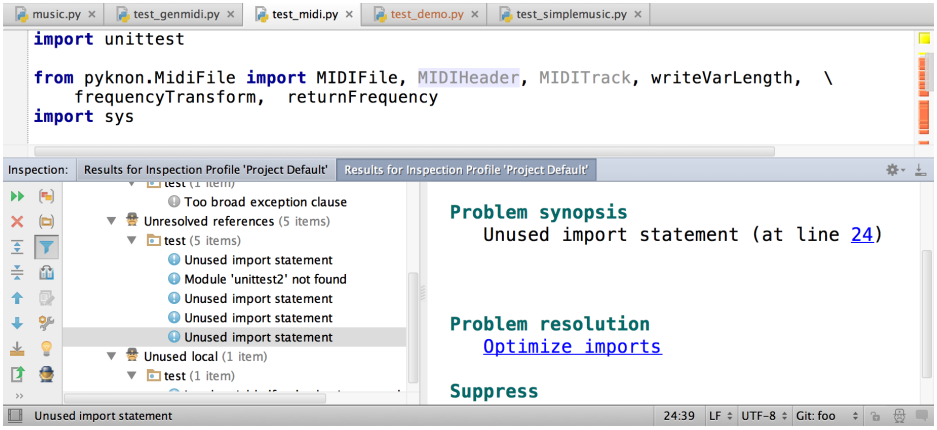
We can inspect the whole code at once in *Code → Inspect Code*. PyCharm will ask for the inspection scope (the whole project, uncommitted files, the current file, the current module, and so on) and will show a list with the problems in the selected scope.



It's very useful to be able to create custom scopes. In the following example, I create a scope named `Test Files` that, as the name suggests, only applies to files in the `test` directory:

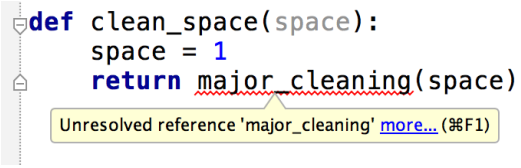


After inspecting the code, PyCharm will show a list with all the problems it found, and it will suggest possible solutions.

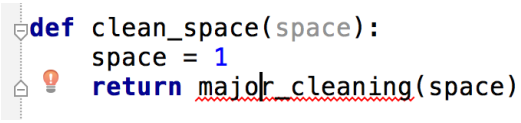


Intention Actions

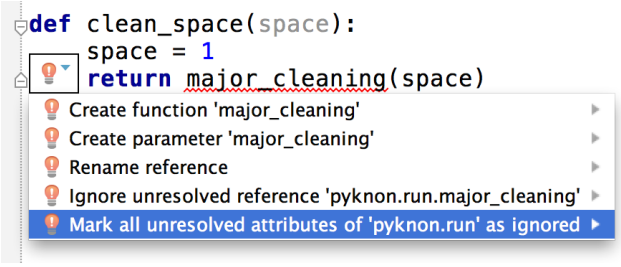
When the code inspection detects a problem, PyCharm will often try to offer solutions, the so-called intention actions. In the following example, the function `major_cleaning` is undefined:



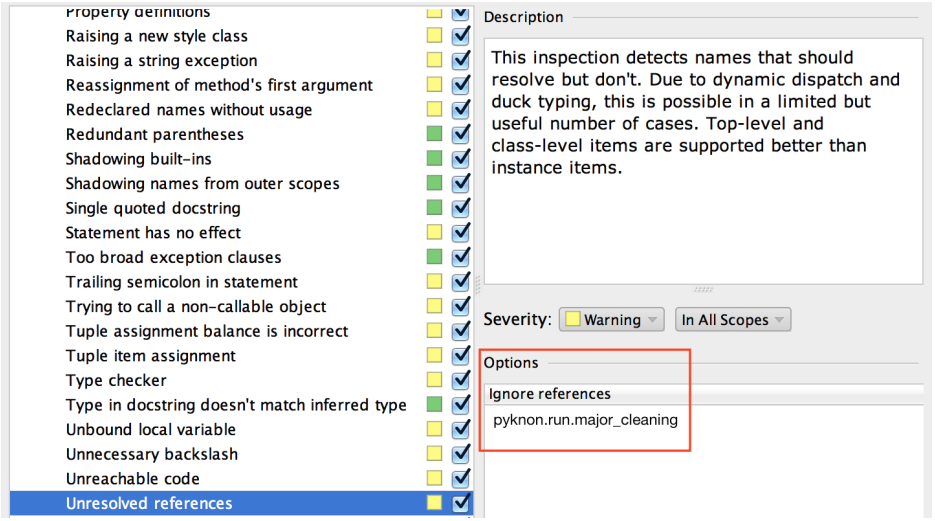
If we move the cursor anywhere in the underlined area (in this case, where `major_cleaning` is), PyCharm will show an intention action icon (the manual lists all types of intention action icons).



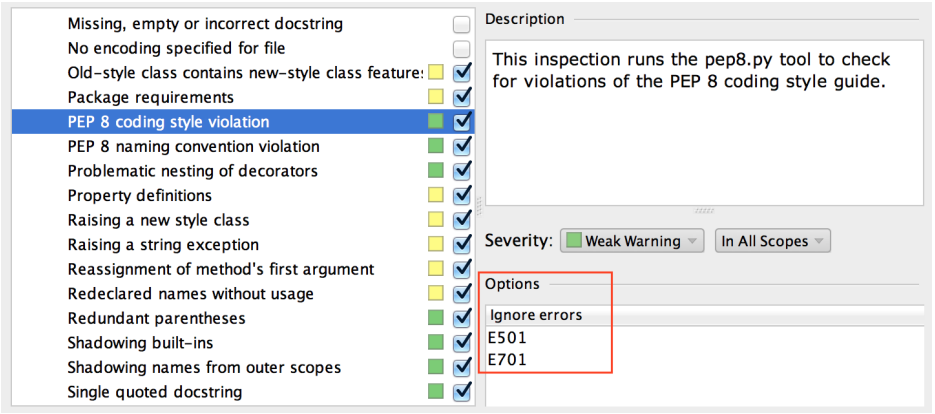
By clicking on the icon or using a shortcut (⌥-*Enter, A-Enter*), we can see the possible solutions. The solutions for the next example are self-explanatory: assume `major_cleaning` is a function and create it, assume it is an argument for `clean_space` and create it, assume we mistyped it and need to rename it, or just ignore it.



PyCharm will list ignored items (such as unresolved references or package requirements) in the related inspection. For instance, if we ignored the unresolved reference for `major_cleaning` by mistake in the previous example, we would go to *Settings* → *Editor* → *Inspections*, look for "Unresolved references" in the Python section, and delete the reference in the "Ignore references" list.



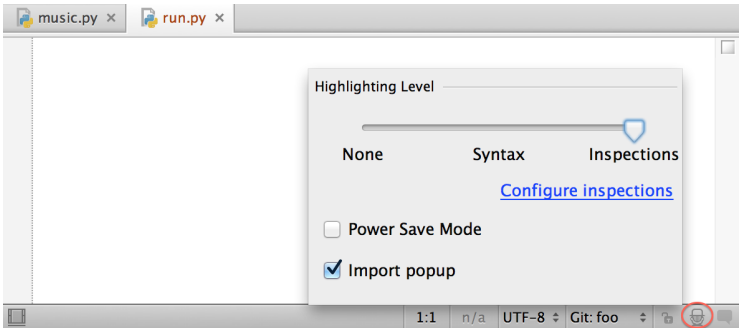
PyCharm uses 120 characters per line (see section *Initial Configuration*). We can change that or we can ignore any PEP 8 violation in *Settings* → *Editor* → *Inspections* by selecting "PEP 8 coding style violation." We can enter the error code by hand or by ignoring the intention action, as we've seen before.



We can find a full list of PEP 8 error codes here.

Disabling Inspections and Intentions

As we've seen, we can disable some inspections in *Settings → Editor → Inspections*. Also, we can click the Hector icon on the status bar (see more about the Hector icon and the status bar in the manual) to temporarily reduce the amount of inspection annotation:



With the highlighting level at "inspections," PyCharm will show, as usual, problems in our code. In the following example, `major_feature` is undefined:

```
def clean_space(space):
    return major_feature(space)
```

If we change the highlighting level to none, the annotations will disappear, but PyCharm still may show some intention icons:

```
def clean_space(space):
    return major_feature(space)
```

We can ask PyCharm not to show the intention icons by setting the option `SHOW_INTENTION_BULB` to false in the `editor.xml` file in the IDE Settings folder (check the manual on *Reference → Project and IDE Settings* to see where the

IDE Settings folder is located in your platform). We need to restart PyCharm for this to work (it seems this feature was removed from PyCharm 4.)

```
<option name="SHOW_INTENTION_BULB" value="false" />
<option name="IS_CARET_BLINKING" value="false" />
<option name="CARET_BLINKING_PERIOD" value="500" />
```

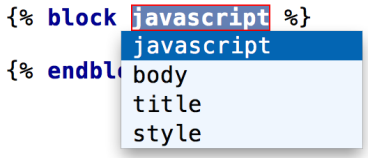
Code Snippets

As with many programming editors, PyCharm has code snippets where we can type a word and have it expand into some code. Many programmers love code snippets because they can help to write code faster and correctly. In PyCharm, code snippets are defined using the Velocity Template Language (VTL), an open-source template engine developed by the Apache project.

PyCharm has two kinds of code snippets: live templates and file templates.

Live Templates

Live templates expand when we type a word. For instance, when we type "block" and *TAB* on a Django template file, it will expand to a block call and will list the known block names, so we can autocomplete.

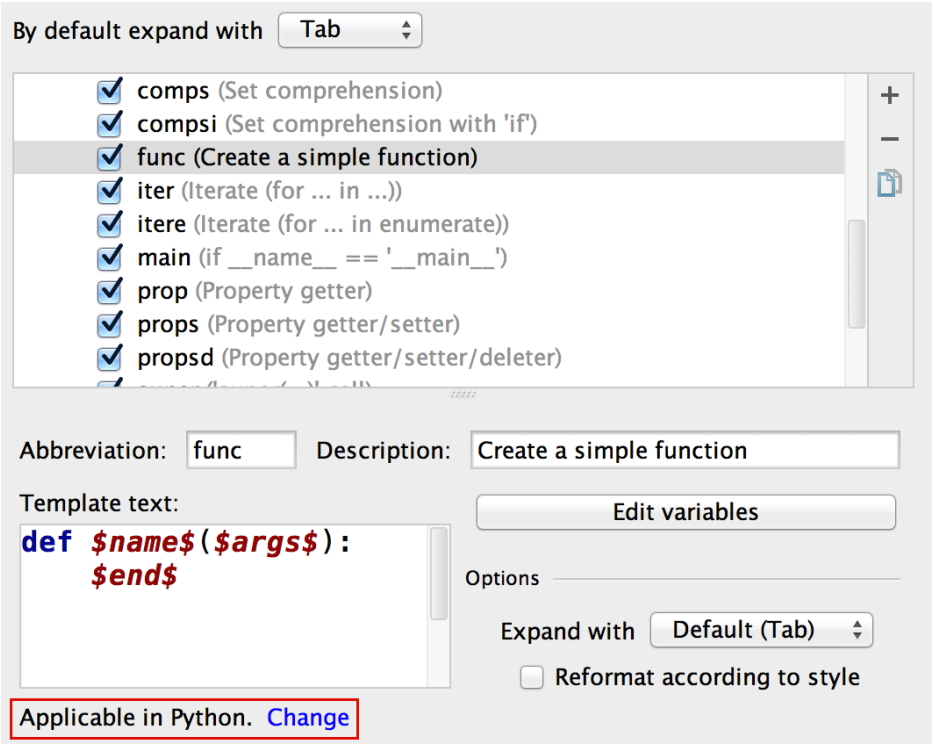


PyCharm has live templates for Django, flask, HTML, JavaScript, Python, SQL, and Zen coding. We can see all the defined templates in *Settings* →

Editor → Live Templates.

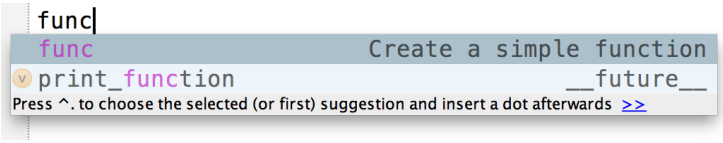
As in other template languages, we can define placeholders to ask the user for a name during the completion. The user can type a name or, in some cases, pick a name from a completion list. User-defined placeholders are delimited by dollar signs (for instance, `$variable$`) and case doesn't matter.

There are two predefined template variables: `$end$` and `$selection$`. The variable `$end$` marks where the cursor will be after the template is expanded. For instance, in the following image, we can see a simple template to create Python functions. To use the template, we type `func` followed by TABs to move to each placeholder. (We need to pick the right context where the template will be expanded. In this case it's, Python.)



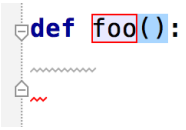
Here's the live template expansion step-by-step:

- Type "func":

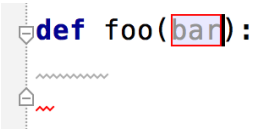


- Type *TAB*; the cursor moves to the place before the open parenthesis:

- Type the function name:



- Type *TAB*; the cursor moves after the open parenthesis. Type the arguments:



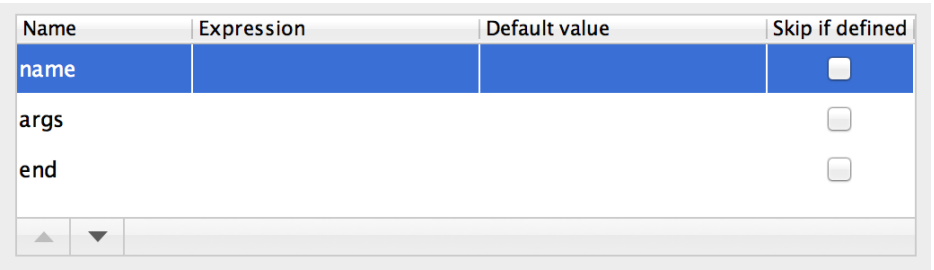
- Type the final *TAB*; the cursor moves to the next line.

```
def foo(bar):
    
```

- Finally, add the content for the function:

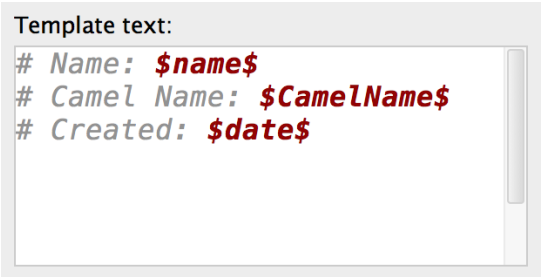
```
def foo(bar):
    return bar
```

Sometimes, it is necessary to edit the order of the variables by clicking in "Edit variables."

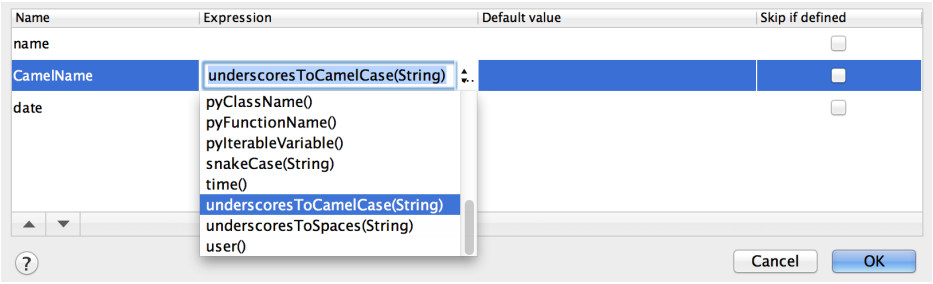


Besides changing the variable order, we can select a function (in the Expression column) that will either perform an action or return a value. For instance, the functions `date` and `time` will return the expected values, whereas the function `complete` will invoke code completion at the variable position. A list of functions is available in the manual.

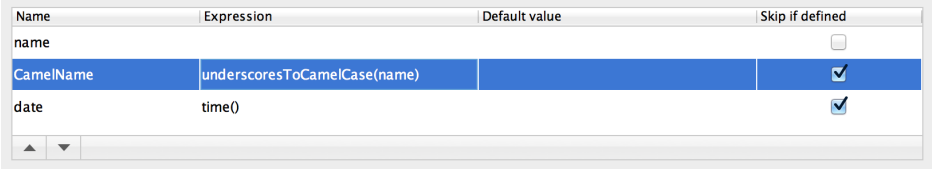
For example, let's define a simple template:



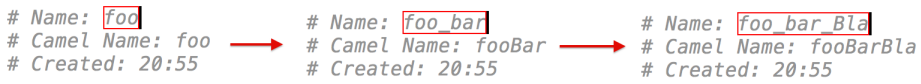
And pick the function `underscoresToCamelCase` for the variable `CamelName`.



We need to edit it so `underscoresToCamelCase` will receive `name` (the variable we defined in the template) as an argument. That is, the variable `CamelName` will be substituted by converting the value of `name` to `CamelCase`.



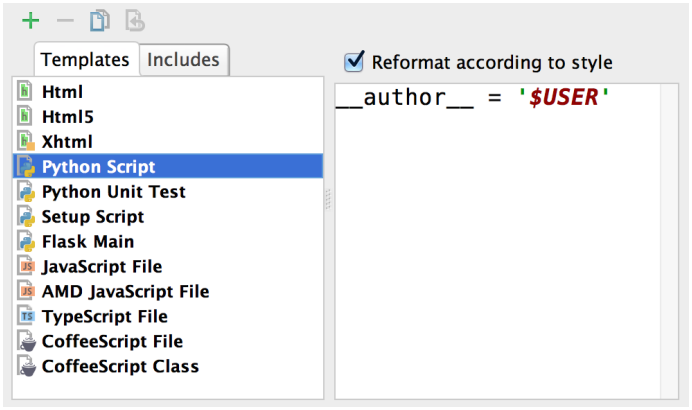
The substitution happens while we type. In the following example, `fooBarBla` in the second line appears while we type `foo_bar_bla` in the first line:



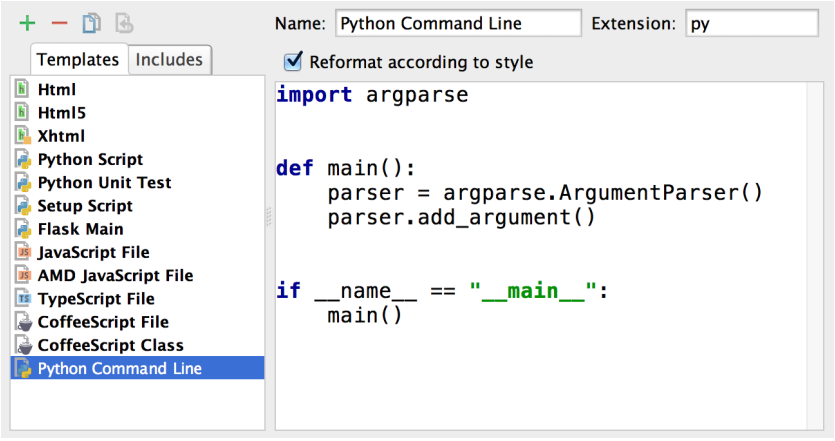
Check the PyCharm manual for a complete list of variable functions.

File Templates

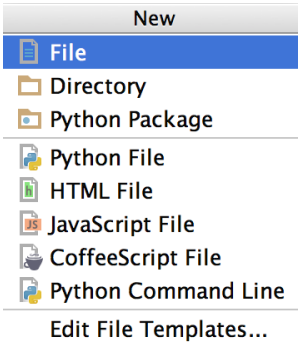
File templates are used to define the initial content when we create new files. We can see the existing templates and create and edit new ones in *Settings → Editor → File and Code Templates*. For example, this is the template that is used when we create a new Python file:



In the following example, I created a simple template for a command line utility (named "Python Command Line"):



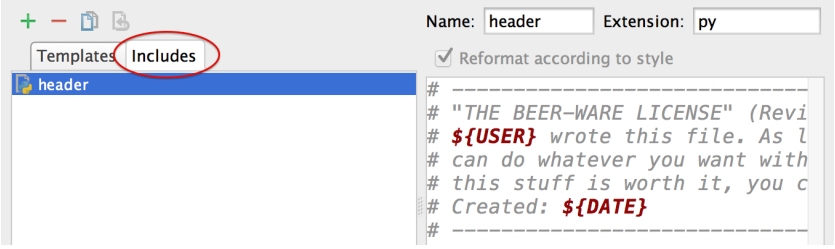
The template is readily available when we create a new file (*File* → *New*):



We can use the `#parse` directive to include data inside a template; for instance, to add a common header to files. We include the header `header.py` in our example:

```
#parse("header.py")

import argparse


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument()


if __name__ == "__main__":
    main()
```

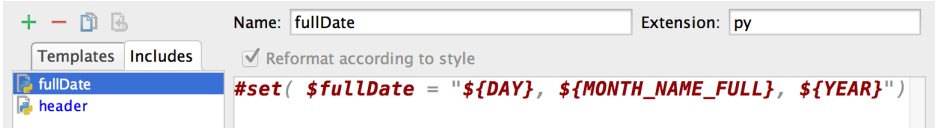
The file included by `#parse` must be defined in the Includes tab. In this example, I'm using the Beerware license and special variables such as `${USER}` and `${DATE}`:



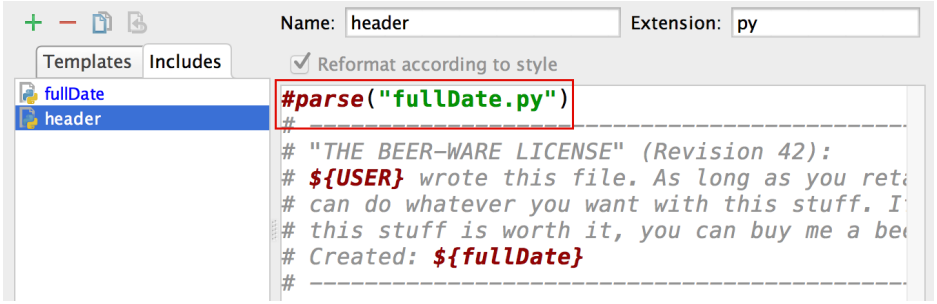
This is a snippet of the result:

```
# ----------------------------------------
# "THE BEER-WARE LICENSE" (Revision 42):
# kroger wrote this file. As long as you
# can do whatever you want with this stuf
# this stuff is worth it, you can buy me
# Created: 26/06/2014
# ----------------------------------------

import argparse


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument()


if __name__ == "__main__":
    main()
```

We can define our own variables using `#set`. In the next example, I define the variable `$fullDate` in the file `fullDate.py`:



And I replace the `${DATE}` variable in our previous example with our new `${fullDate}`:



And this is a snippet of the result:

```
# ----------------------------------------
# "THE BEER-WARE LICENSE" (Revision 42):
# kroger wrote this file. As long as you
# can do whatever you want with this stu
# this stuff is worth it, you can buy me
# Created: 26, June, 2014
# ----------------------------------------
```

These are the predefined template variables as listed in the manual at *PyCharm Usage Guidelines → File and Code Templates → File Template Variables*:

- `${PROJECT_NAME}` – the name of the current project.

- `${NAME}` – the name of the new file that you specify in the New File dialog box during the file creation.
- `${USER}` – the log-in name of the current user.
- `${DATE}` – the current system date.
- `${TIME}` – the current system time.
- `${YEAR}` – the current year.
- `${MONTH}` – the current month.
- `${DAY}` – the current day of the month.
- `${HOUR}` – the current hour.
- `${MINUTE}` – the current minute.
- `${PRODUCT_NAME}` – the name of the IDE in which the file will be created.
- `${MONTH_NAME_SHORT}` – the first three letters of the month name. Example: Jan, Feb, etc.
- `${MONTH_NAME_FULL}` – full name of a month. For example: January, February, etc.

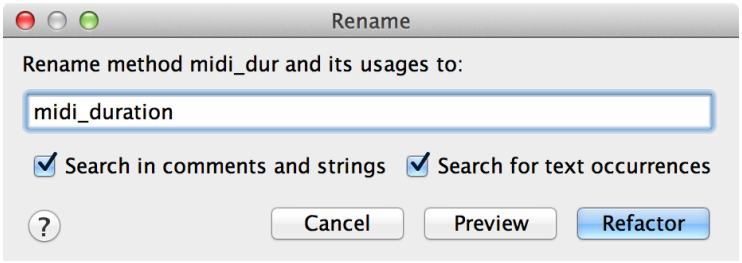
We can also create a template from an existing file with *Tools → Save File as Template*. PyCharm will create a new template with the content of the current file and open the "File and Code Templates" dialog for further editing.

Refactoring

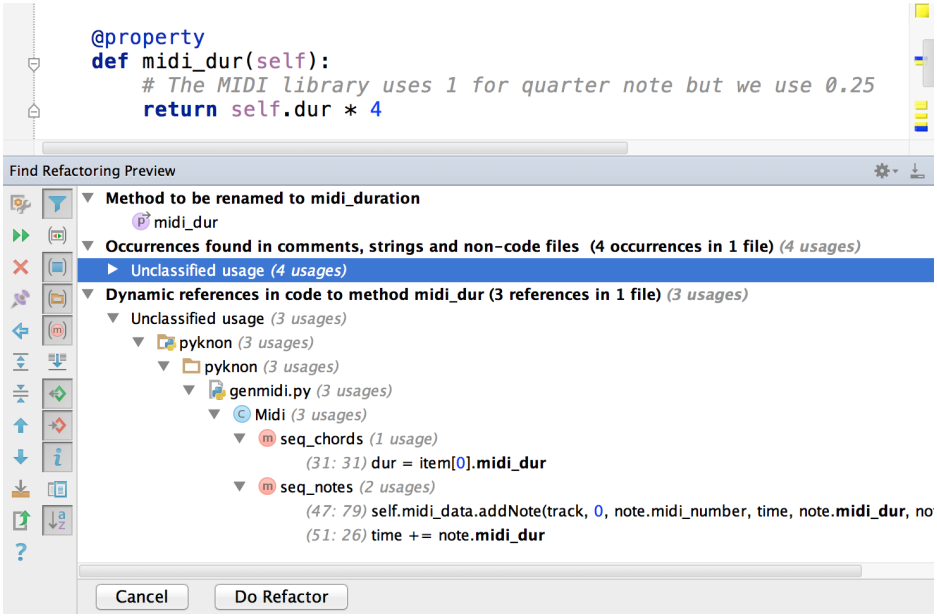
Renaming is probably the most common refactoring technique, and PyCharm will rename pretty much everything, including file names. To

rename a symbol, we go to *Refactor → Rename (S-F6, S-F6)*, and with the cursor on the symbol, we want to rename and type the new name on the popup window. We can choose to rename the symbol in comments and strings, in addition to symbols. In the following example, I'm renaming the function `midi_dur`:

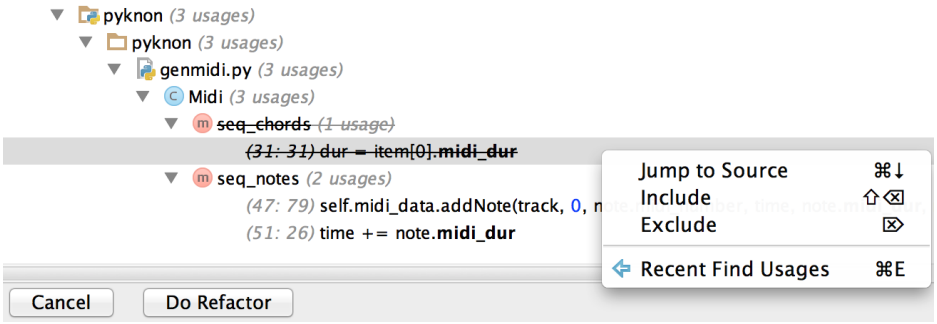
```
@property
def midi_dur(self):
    # The MIDI library uses 1 for quarter note but we use 0.25
    return self.dur * 4
```



Before performing the refactoring, PyCharm will show a preview window, where we can double check if everything is OK and include or exclude items from being refactored.

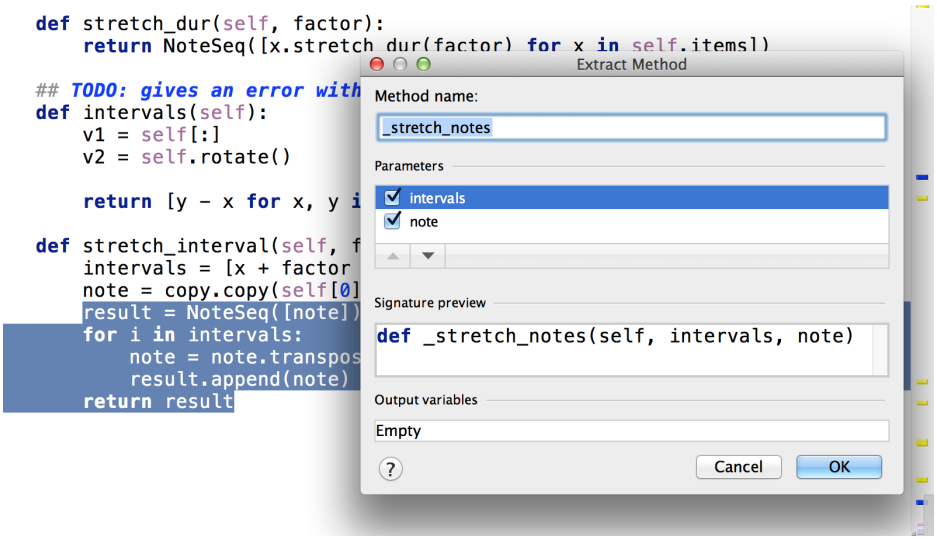


To exclude an item from being refactored (in case PyCharm selects something by mistake), we open the popup menu and click on Exclude.



The extract method refactoring, in *Refactor → Extract → Method…* (*⌘-⌥-m, A-C-m*), moves a code fragment into a separate method. The goal is to have a code whose method name explains what it does. PyCharm will automatically detect the parameters needed by analyzing the free variables in the selected

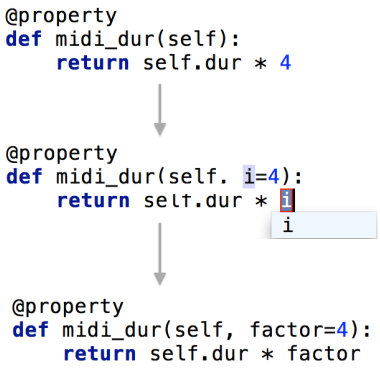
code.



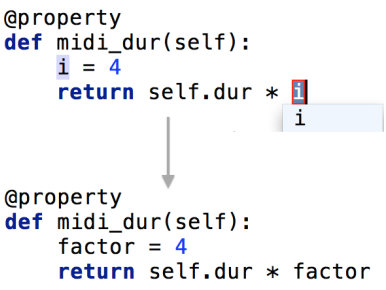
PyCharm will create the new method and replace the original code block with a method call:

```
def _stretch_notes(self, intervals, note):
    result = NoteSeq([note])
    for i in intervals:
        note = note.transposition(i)
        result.append(note)
    return result

def stretch_interval(self, factor):
    intervals = [x + factor for x in self.intervals()]
    note = copy.copy(self[0])
    return self._stretch_notes(intervals, note)
```

Sometimes, it's useful to abstract some literal value into a variable, parameter, constant, or attribute. To extract a parameter, we select the expression, go to *Refactor → Extract → Parameter…* (*⌘-⌥-p, A-C-p*), and type the name of the parameter in place. It's that simple.



The method to extract an expression to a local variable is similar; we select the expression, go to *Refactor* → *Extract* → *Variable…* (*⌘-⌥-v, A-C-v*), and type the name of the new variable in place.



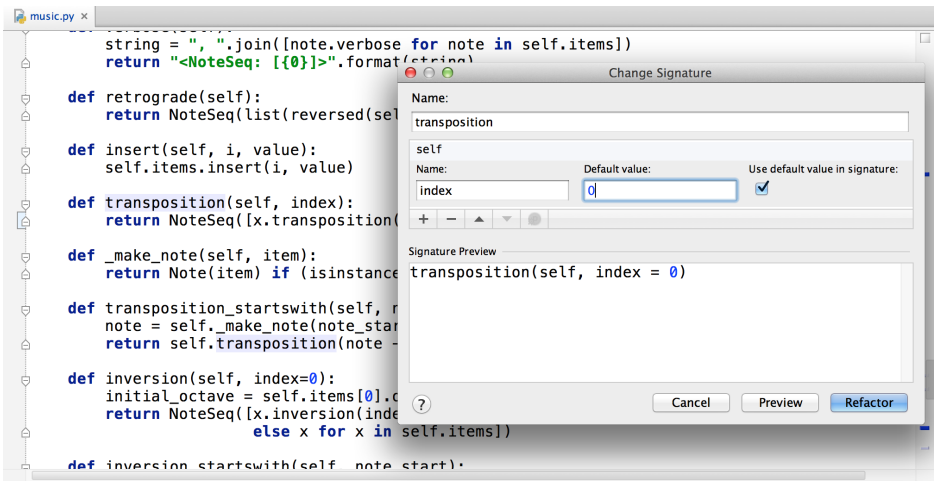
It's even possible to extract a substring to a variable, constant, attribute, or parameter. We select the string to be extracted and use the aforementioned actions (for instance, *Refactor* → *Extract* → *Variable…*). In the following example, I extract a substring to a variable:

```
message = "Warning: your code is wrong!"
```

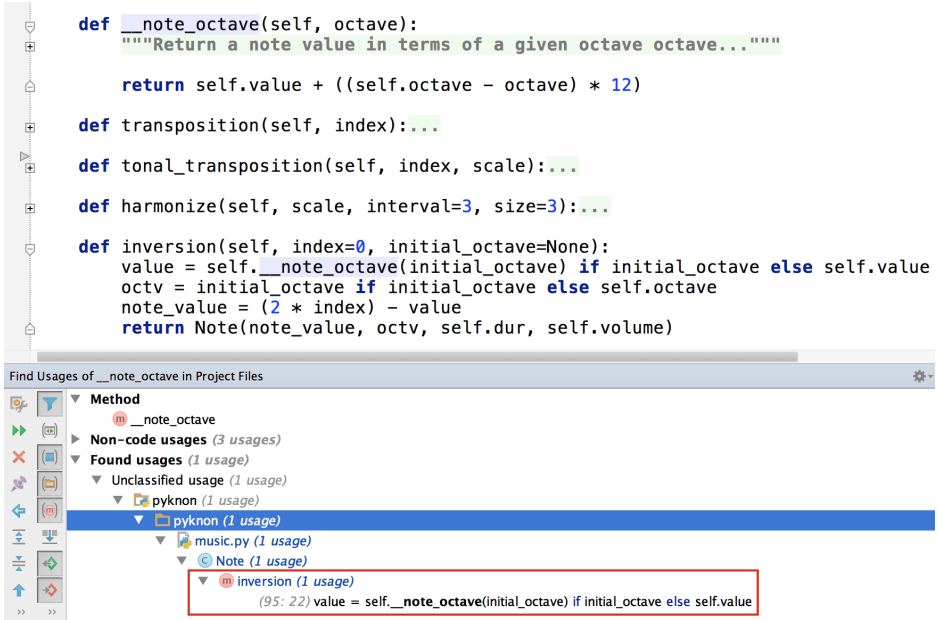


```
warning = "wrong"
message = "Warning: your code is %s!" % warning
```

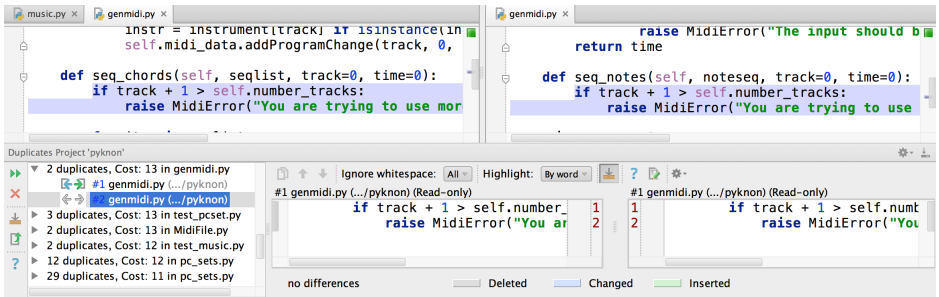
Another useful refactoring technique is to change the signature of a function. We can accomplish this by putting the cursor on the method or function name and going to *Refactor → Change Signature… (⌘-F6, A-F6)*. We can add or remove attributes and set a default value for attributes as well.



One action that is not a refactoring technique per se but is very useful while refactoring is to find where in the code a symbol is used. We can accomplish this with *Edit → Find → Find usages (⌥-F7, A-F7)*. In the following example, I notice that I can move the body of the method `__note_octave` inside `inversion`. I use *Find usages* to make sure `__note_octave` is not called anywhere else in the whole project before moving it.

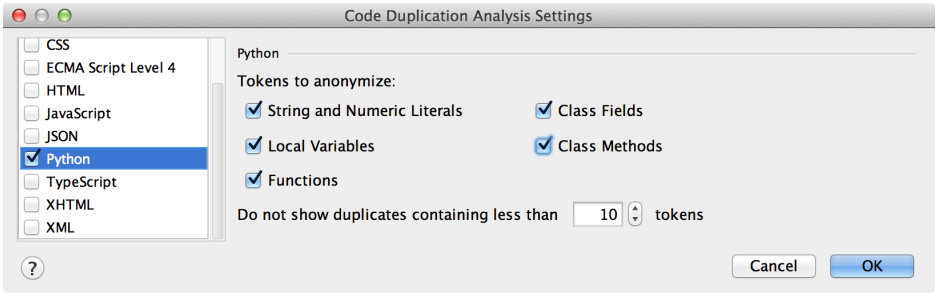


As we all know, duplicate code is often a bad thing to have. We can find duplicates by going to *Code → Locate Duplicates…*. In the following example, PyCharm detected a couple of duplications in my code that could be abstracted in a function. This action won't eliminate the duplication automatically. You will have to either do it manually or use the other refactoring actions.



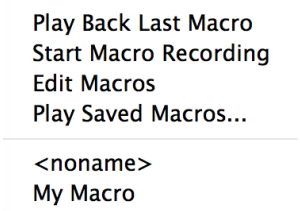
When we run the code duplication analysis, PyCharm will ask us to define

the sensitivity of the search by selecting what tokens should be anonymized.

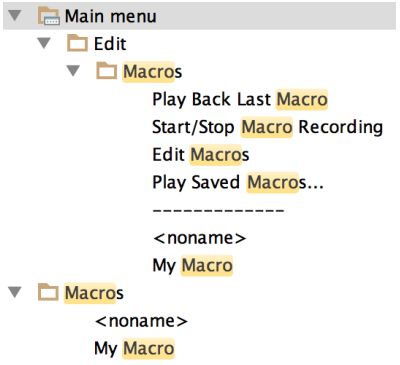


Automation with Macros

Macros in PyCharm are intended for simple editor operations, and they can't record actions such as button clicks and dialog boxes navigation. Despite these limitations, they can be very useful when we need to automate repetitive tasks. We can access the macro-related actions in *Edit → Macros*.

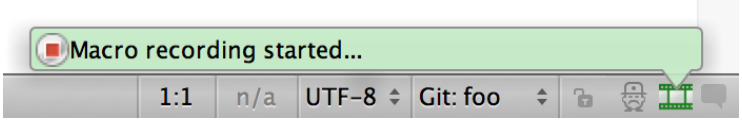


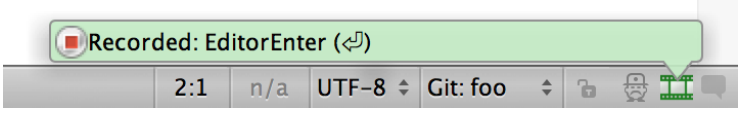
No macro actions have a default shortcut, but it's easy enough to add our own shortcuts (see *Shortcuts*). We can also add shortcuts to macros we have recorded.



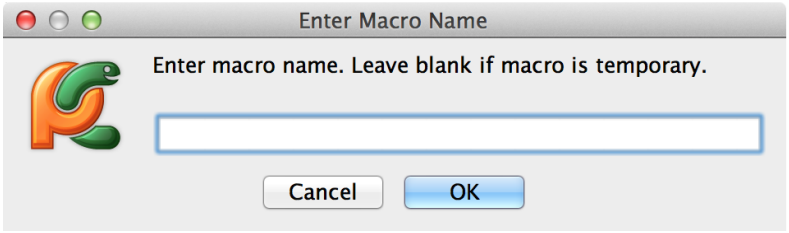
Recording Macros

To record a new macro, we go to *Edit → Macros → Start Macro Recording*. While recording a macro, PyCharm shows a notification on the bottom-right corner with the macro status and the actions it's recording.





We can stop the recording by clicking on *Edit → Macros → Stop Macro Recording* or by clicking on the stop button in the notification area. After recording we will be asked to enter a name. If we leave the field blank, PyCharm will create a temporary macro named `<noname>`.

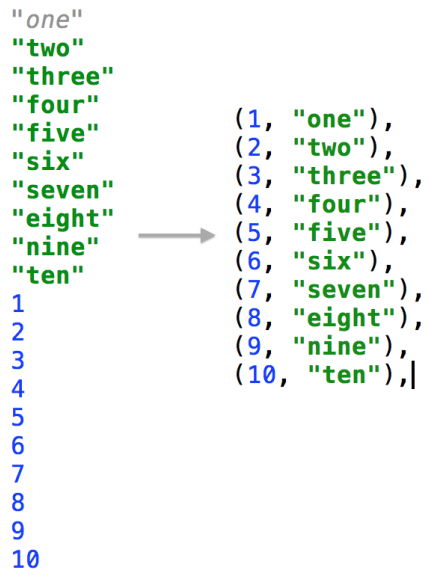


We can access (that is, replay) our macros by going to *Edit → Macros* and selecting the macro name, or by going to *Edit → Macros → Play Saved Macros*.

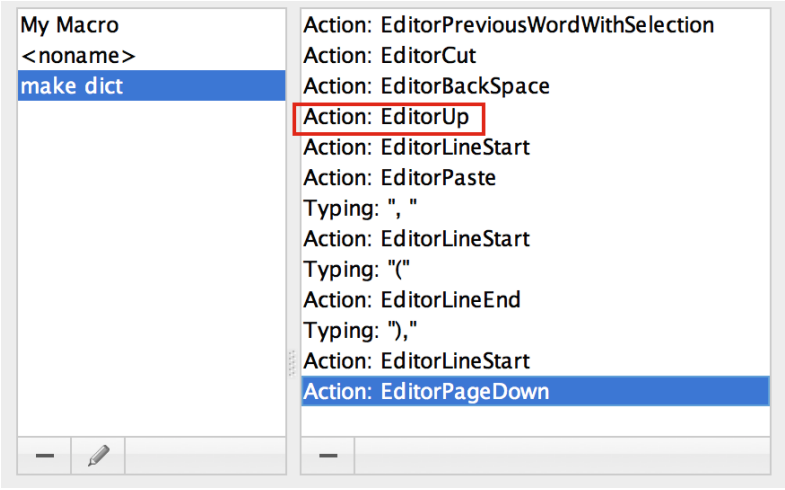
Editing Macros

If we make a mistake while recording the macro, we may be able to edit it without having to record the whole thing again (*Edit → Macros → Edit Macros*). The macro editing features are very simple; we can rename a macro and remove a recorded action, but we can't add new actions.

For instance, we create a macro to interweave two blocks of text into tuples in the following way:



But after recording the macro, we realize that we should have hit the up arrow nine times instead of just one (each up arrow is an `EditorUp` action):

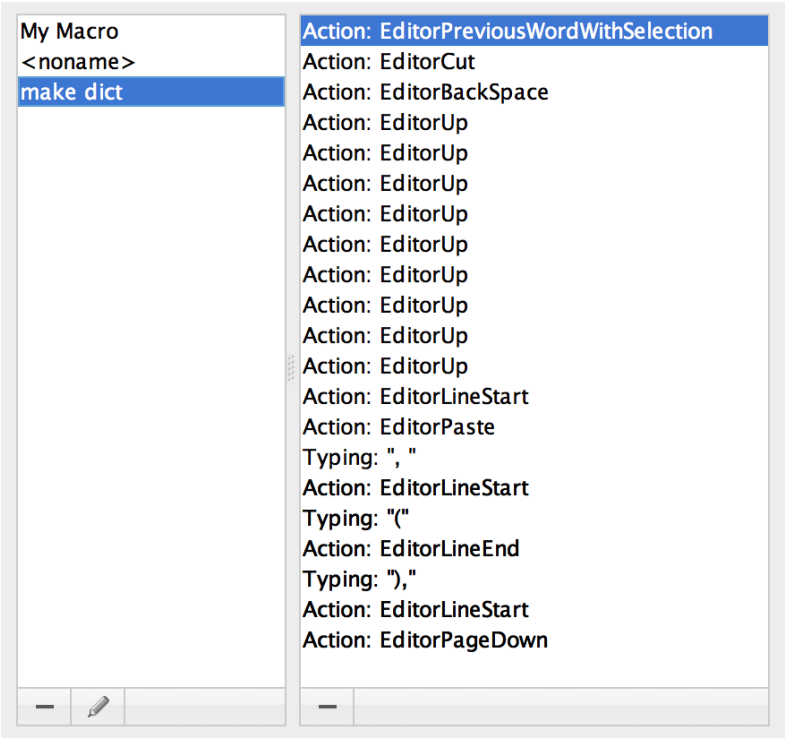


However, if you are feeling adventurous, PyCharm saves all macros in a file

named `macros.xml` in the IDE Settings folder (check the manual on *Reference → Project and IDE Settings* to see where the IDE Settings folder is located in your platform). We can edit this file and add the actions we want. In our example, we just need to copy the line with `<action id=”EditorUp” />` eight more times:

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
  <component name="ActionMacroManager">
    <macro name="make dict">
      <action id="EditorPreviousWordWithSelection" />
      <action id="EditorCut" />
      <action id="EditorBackSpace" />
      <action id="EditorUp" />
      <action id="EditorLineStart" />
      <action id="EditorPaste" />
      <typing text-keycode="87:0;32:0">,&amp;#x20;</typing>
      <action id="EditorLineStart" />
      <typing text-keycode="57:1">(</typing>
      <action id="EditorLineEnd" />
      <typing text-keycode="48:1;87:0">),</typing>
      <action id="EditorLineStart" />
      <action id="EditorPageDown" />
    </macro>
  </component>
</application>
```

The actions are available in the macro interface after we restart PyCharm.



Naturally, this is not the intended way to work with macros in PyCharm (it's kind of hacky). But it's a good trick to know in case you need to fix a mistake in a long and complex macro and don't feel like recording it again (and possibly making more mistakes).

If you need to write complex macros, you may want to try the Vim plugin (*Vim Plugin*) or quickly open the file in an external editor such as Vim or Emacs (*External Tools*).

Vim Plugin

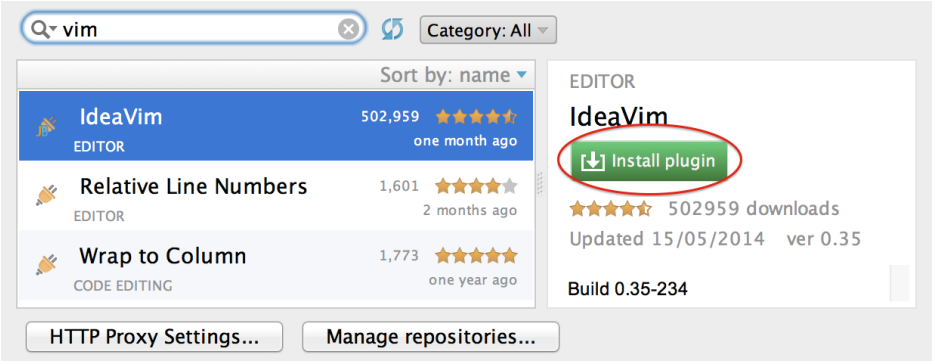
Vim is an advanced text editor with many powerful features, allowing a proficient user to edit text very efficiently. An introduction to Vim is out of the scope of this book. I recommend Drew Neil's *Practical Vim: Edit Text at the Speed of Thought* if you want to learn more about it. A quick disclaimer: I'm not a Vim expert by any measure, although I have used Emacs (another super-powerful editor) for more than 14 years, so I understand the appeal of using a super-editor.

Installation

To install the Vim plugin, go to *Settings* → *Plugins* where you will see a list of installed plugins (the Vim plugin is probably not on the list). Click in "Browse repositories…"

PyCharm will show a huge list with all available plugins. You can browse the list, but the easiest way is to search for "vim," select the IdeaVim entry, and

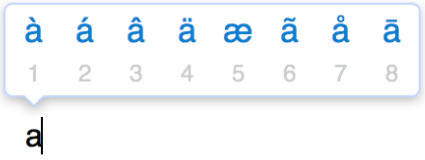
click on the “Install plugin” button:



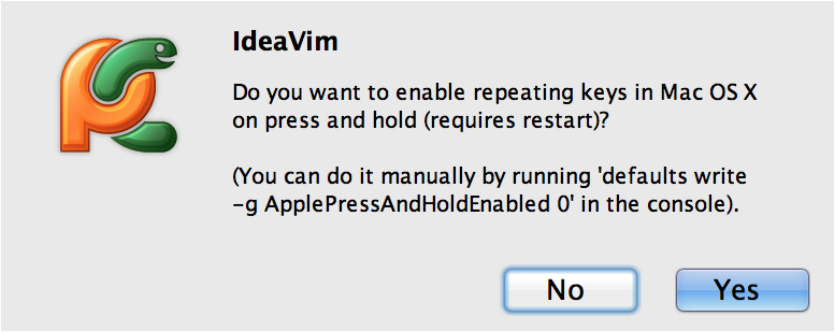
You will need to restart PyCharm for the plugin to work.

The rest of this section deals with how the plugin works on a Mac. If you don’t use a Mac, skip to *Using the Vim Emulator*.

In recent versions of OS X, a popup with accented characters will appear when we hold down some keys (especially the ones with vowels):



This behavior interferes with the Vim plugin; therefore PyCharm asks if we want to enable the old behavior:



For instance, if we press and hold down the letter `l` to move forward quickly, we will get the following instead:

```
def harmonize(self, scal
    i = (interval - 1)
    i  ces = range(1, s
    r  rn [self.tonal_t

def inversion(self, inde
    value = self.__note_
    octv = initial_octav
    note_value = (2 * in
    return Note(note_val
```

We can disable it globally by running the following command on the terminal:

```
defaults write -g ApplePressAndHoldEnabled -bool false
```

Or we can enable repeating keys for specific applications. For PyCharm the command is:

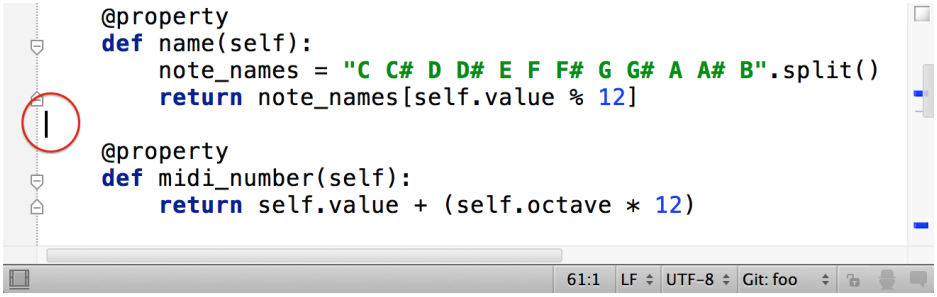
```
defaults write com.jetbrains.pycharm \
  ApplePressAndHoldEnabled -bool false
```

It's necessary to restart PyCharm.

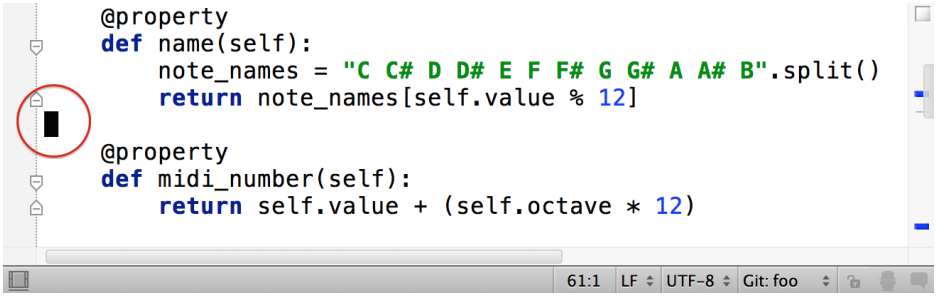
Using the Vim Emulator

We start the Vim plugin by going to *Tools → Vim Emulator (⌘-⌥-v, A-C-v)*.

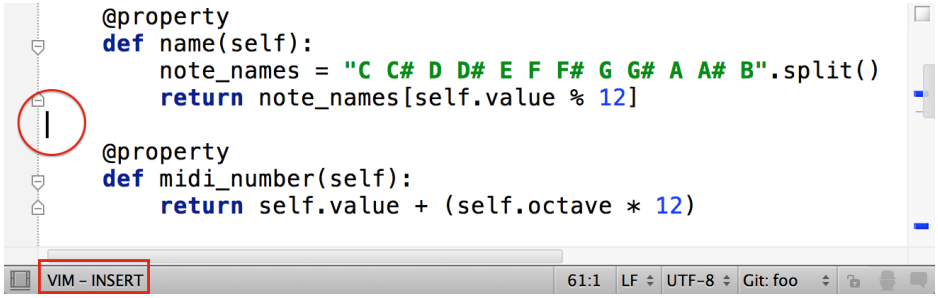
To know if the Vim plugin is enabled, we need to look at the caret's shape. If the plugin is disabled, the caret has a line shape:



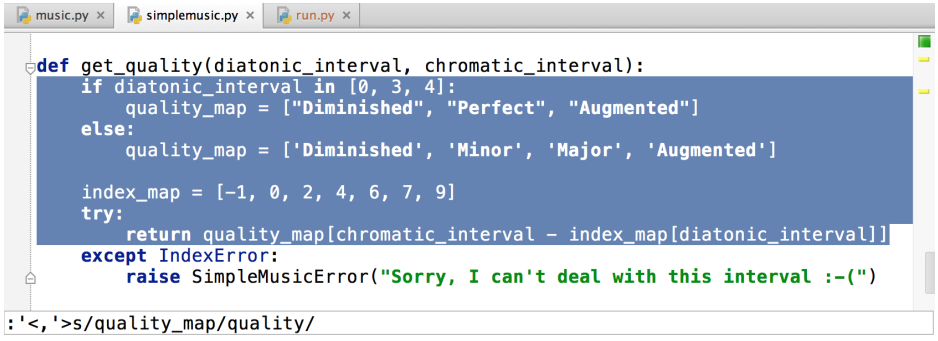
When Vim mode is enabled and in normal mode, the caret is a block:



Finally, when Vim mode is enabled and in insert mode, the cursor is a line. If `showmode` is set on your `.ideavimrc` (which I recommend), the status bar will show "VIM - INSERT" as well:



In the following example, I'm replacing some text by using the visual selection mode:



As you may have noticed, the Vim plugin has a command line just like Vim. It doesn't have completion but it has history. For instance, `:e` works, but without completion, it is not very useful (and PyCharm's way of moving to files is faster).

PyCharm uses the file `.ideavimrc` in your home directory as a configuration file instead of `.vimrc`. It has support for a few `set` commands (from the documentation)

command	abbr.	description
digraph	dg	enable the entering of digraphs in Insert mode
gdefault	gd	the ":substitute" flag 'g' is default on
history	hi	number of command-lines that are remembered
hlsearch	hls	highlight matches with last search pattern
ignorecase	ic	ignore case in search patterns
matchpairs	mps	pairs of characters that "%" can match
nrformats	nf	number formats recognized for CTRL-A command
scroll	scr	lines to scroll with CTRL-U and CTRL-D
scrolljump	sj	minimum number of lines to scroll
scrolloff	so	minimum nr. of lines above and below cursor
selection	sel	what type of selection to use
showmode	smd	message on status line to show current mode
sidescroll	ss	minimum number of columns to scroll horizontal
sidescrolloff	iso	min. nr. of columns to left and right of cursor
smartcase	scs	no ignore case when pattern has uppercase
timeoutlen	tm	time that is waited for a mapped key sequence
undolevels	ul	maximum number of changes that can be undone
visualbell	vb	use visual bell instead of beeping
wrapscan	ws	searches wrap around the end of the file

Supported Features

Here's a list of supported features, lifted straight from the plugin's page:

- Motion keys
- Deletion/changing
- Insert mode commands
- Marks

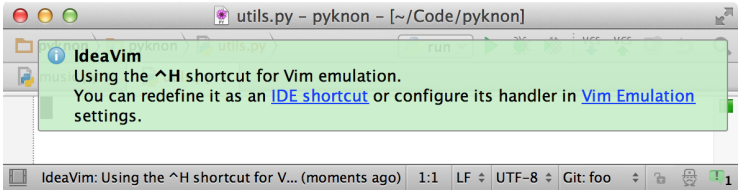
- Registers
- Undo/redo
- Visual mode commands
- Some Ex commands
- Some :set options
- Full Vim regular expressions for search and search/replace
- Key mappings
- Macros
- Digraphs
- Command line and search history
- Vim web help

Custom Keymaps

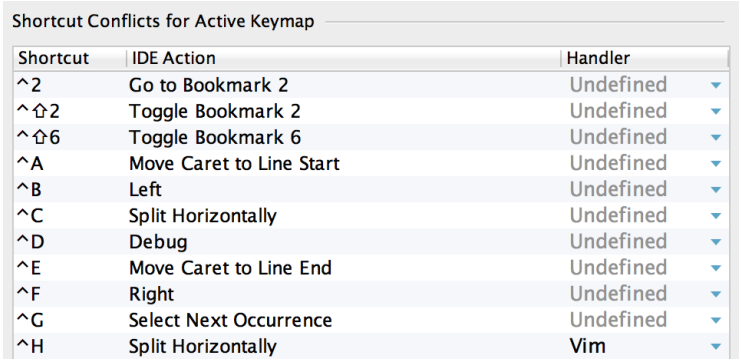
We still can use PyCharm's shortcuts in Vim emulation mode, as long as they don't conflict with Vim's. For example, we still can use *(⌘-o, C-n)* to navigate to a class in Vim's modes (including insert mode).

We can customize the shortcuts for many actions in the Vim plugin in *Settings → Appearance & Behavior → Keymap → Plug-ins → IdeaVim*. One action you may want to customize is "Exit Insert Mode," especially if you want to use something like *C-c*, as many people do. We can also use Vim's key mapping functions such as `noremap` in our `.ideavimrc` file.

PyCharm will warn us when we try to use a shortcut that is defined both in the IDE and in the Vim plugin (that is, when they conflict with each other).

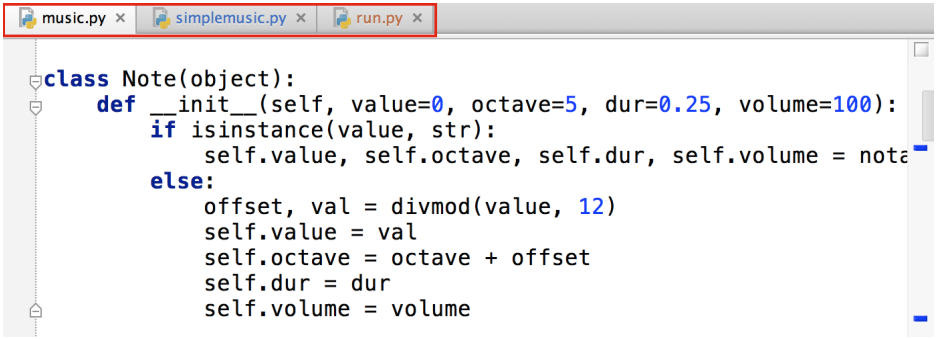


Shortcut conflicts can be resolved by going to *Settings* → *Other Settings* → *Vim Emulation*. It shows the shortcut, the action assigned to the shortcut, and who should handle it (that is, Vim emulation or PyCharm).

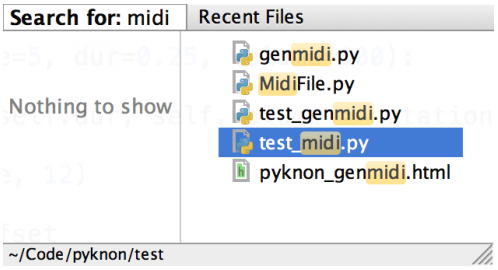


Dealing with Multiple Files

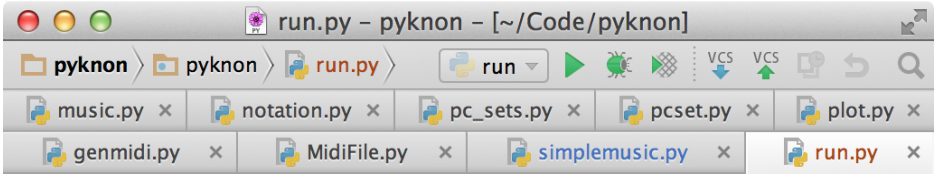
As with many modern editors, PyCharm uses one tab per file:



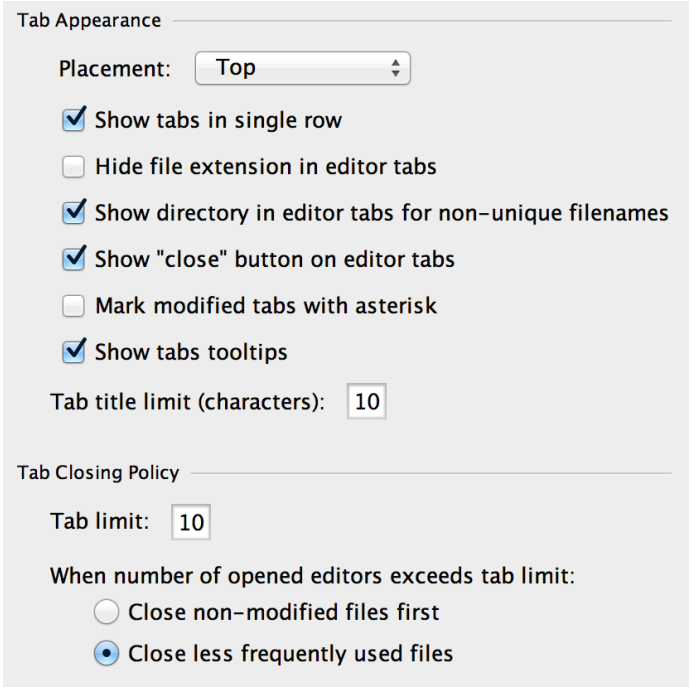
We can go to *Windows → Editor Tabs → Select Next Tab (⌘-S-], A-Right)* and *Select Previous Tab (⌘-S-[, A-Left)* to go to the next and previous tab, respectively. A more direct and faster approach is to use *Navigate → File…*, as we've seen, or *View → Recent Files (⌘-e, C-e)* and *View → Recently Changed Files (⌘-S-e, S-C-e)*. These last two commands will present a list of files where you can click on the file or type a substring to narrow the selection. For instance, in the following example, I only had to type "⌘-e" (or C-e), "ma," and the Enter key to switch to the tab containing the `manager.py` file.



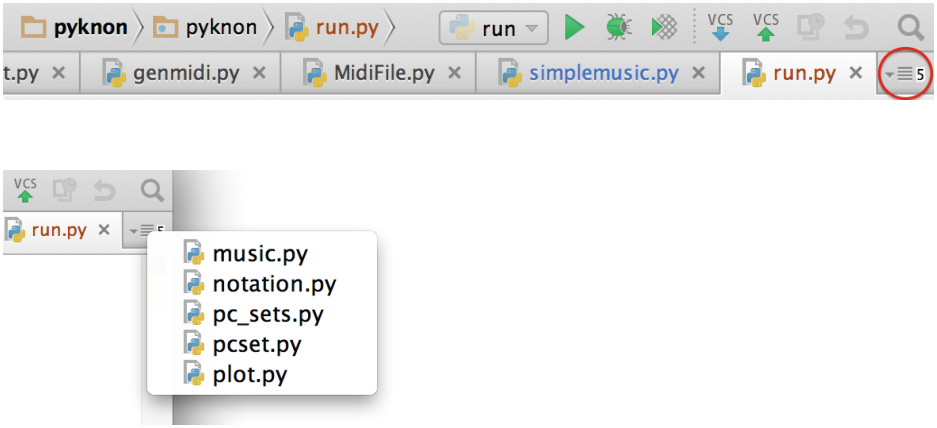
Depending on how many files you have opened, PyCharm may show the tabs in more than one row.



If you prefer to optimize space, you can tell PyCharm to show tabs in a single row by going to *Settings* → *Editor* → *General* → *Editor Tabs* and selecting "Show tabs in single row."

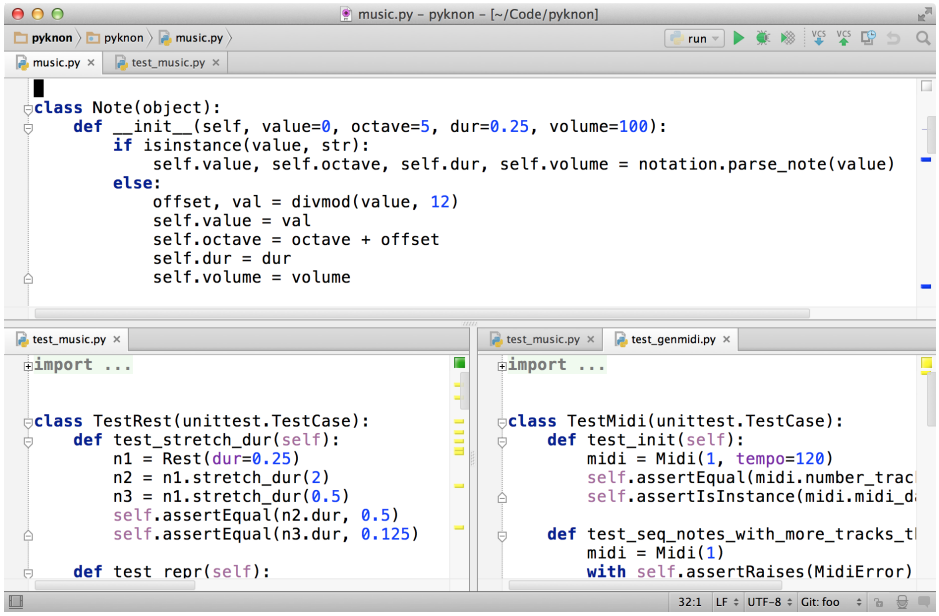


It will show the most recent tabs on the main row and hide the least used ones. We still can access the remaining tabs by clicking on the last icon on the tab row.



Split Windows

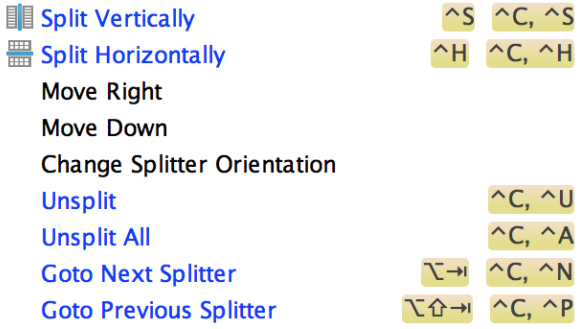
PyCharm has support for split windows. It's not as advanced as in Emacs or Vim, but it's useful enough. We go to *Window → Editor Tabs → Split Vertically* and *Window → Editor Tabs → Split Horizontally* to split a window vertically and horizontally, respectively. We go to *Window → Editor Tabs → Unsplit* to remove the split of the selected window and *Window → Editor Tabs → Unsplit All* to remove all split windows. Finally, we go to *Window → Editor Tabs → Goto Next Splitter (⌥-Tab, no shortcut)* and *Window → Editor Tabs → Goto Previous Splitter (S-⌥-Tab, no shortcut)* to go to the next or previous split window.



Some actions such as *Split Vertically* and *Split Horizontally* don't have shortcuts by default, so you'll have to assign them yourself. These are the shortcuts that I use:

Action	Shortcut
Split Vertically	C-s
Split Horizontally	C-h
Unsplit	C-c C-u
Unsplit All	C-c C-a
Goto Next Splitter	C-c C-n
Goto Previous Splitter	C-c C-p

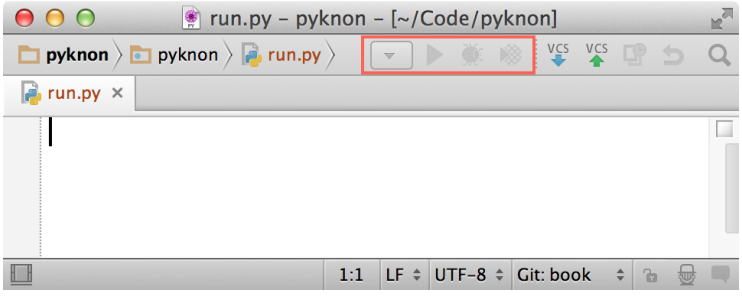
In the following example, we can see the assigned shortcuts in the keymap editor:


Split Vertically ^S ^C, ^S
Split Horizontally ^H ^C, ^H
Move Right
Move Down
Change Splitter Orientation
Unsplit ^C, ^U
Unsplit All ^C, ^A
Goto Next Splitter ⌥→| ^C, ^N
Goto Previous Splitter ⌥⇧→| ^C, ^P

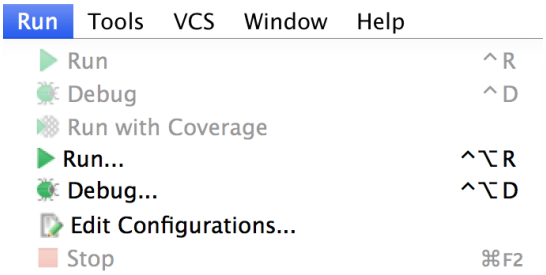
4 | Running, Debugging, and Testing

Running Code

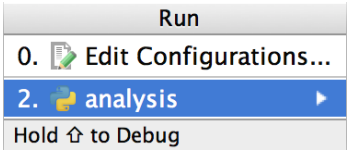
If we haven't run any code in our project before, the icons to run and debug will be disabled:



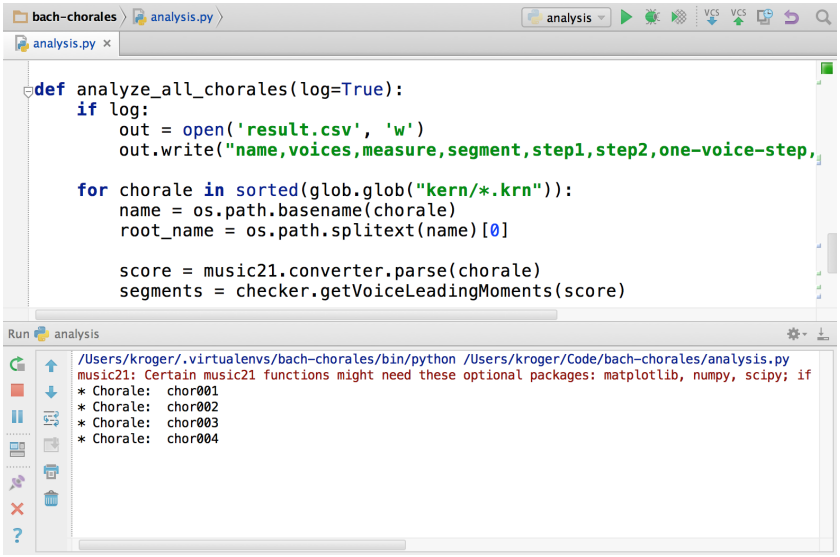
Some items will be disabled in the *Run* menu as well:



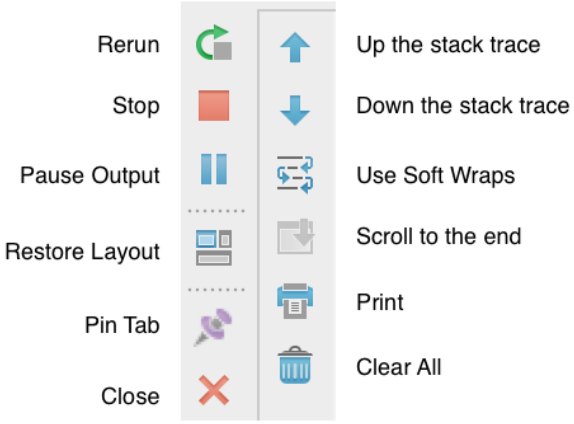
To run a script for the first time, we need to either create a configuration (*Run → Edit Configurations…*) or ask PyCharm to run the current file by going to *Run → Run…* (*⌥-C-r, A-C-r)*, where PyCharm will automatically offer to run the current file. In the following example I'm editing a file named `analysis.py`. Notice how PyCharm creates a new configuration named `analysis`:



PyCharm will run the file and show a tool window with the results, if any.



The Run tool window has many buttons to help us deal with the running script. We can do things like pause, stop, and rerun the script and move up and down the stack trace.



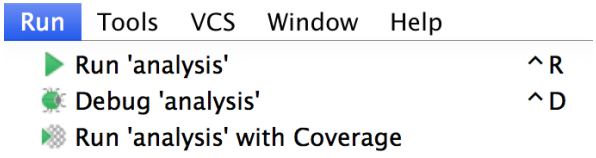
The following table summarizes the main actions. The manual has a complete description.

Name	Shortcut	Description
Rerun	*(⌘-r, C-F5)*	Stop the current process and run it again
Stop	*(⌘-F2, C-F2)*	Stop the current process
Pause Output		Keep running the process but don't show its output
Restore Layout		Restore the default window layout, discarding the current one
Pin Tab		Run code on a new tab
Close	*(⌘-w, S-C-F4)*	Closes the current tab and terminate the process
Up the stack trace	*(⌘-⌥-↑, A-C-↑)*	Navigate up the stack trace
Down the stack trace	*(⌘-⌥-↓, A-C-↓)*	Navigate down the stack trace
Use Soft Wraps		Enable word wrap for long lines
Clear All		Remove all text in the console

We can hide the Run tool window with *(⌘-4, A-4)*.

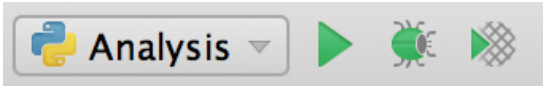
Once a script is run for the first time, it becomes available in the *Run* menu. PyCharm will show the most recently run command in *Run → Run '<script name>' (C-r, S-F10)*, where `<script name>` is the name of the Python file without the extension (`analysis`, in our previous example).

This command will run the original file even if we are working on a different one. In our example, if we are editing a different file (say `chorales.py`) and type *C-r* on the Mac or *S-F10* on Windows or Linux, PyCharm will run the file `analysis.py`, since this is the last file we ran. This is useful when we have a master script and are making modifications to a sub-module, for instance.

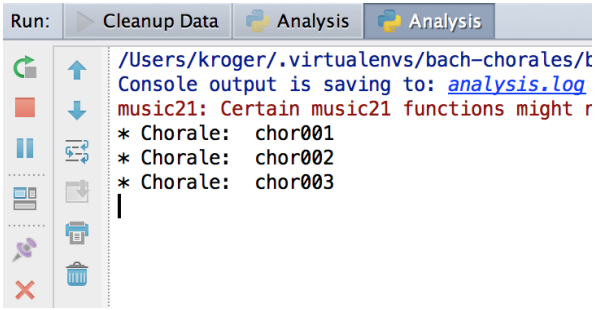


After we run the script for the first time, the related commands in the

menubar will be available, and we can choose the script to run there:

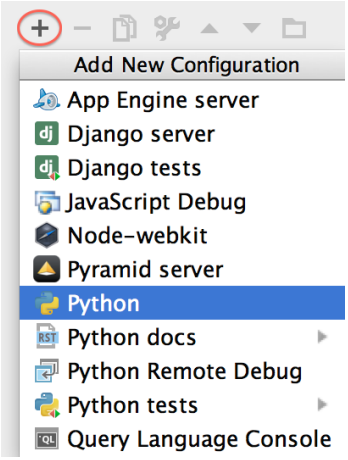


PyCharm will create a new tab every time we run a script. If we don't close the previous tab we may end up with many different tabs, so make sure you close each tab after running, and are, in fact, on the right tab.

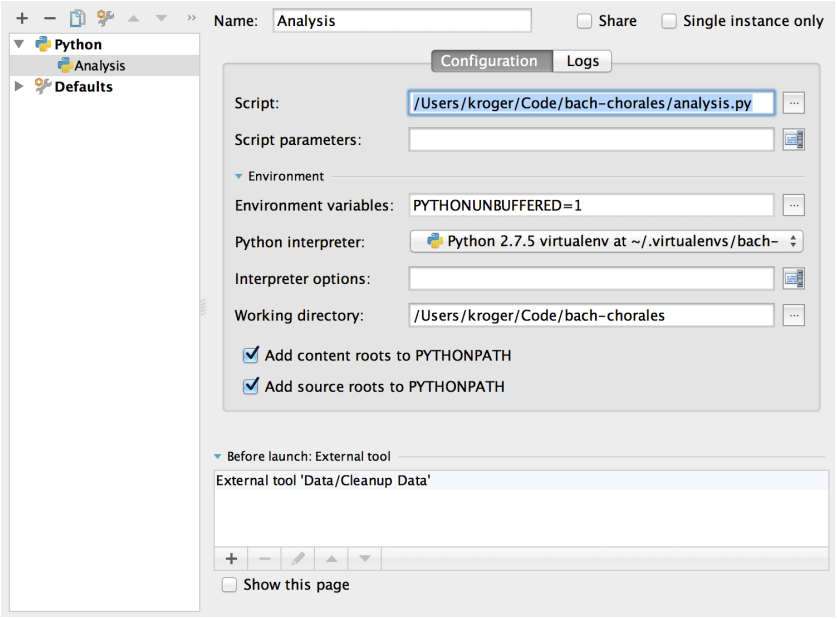


New Configuration

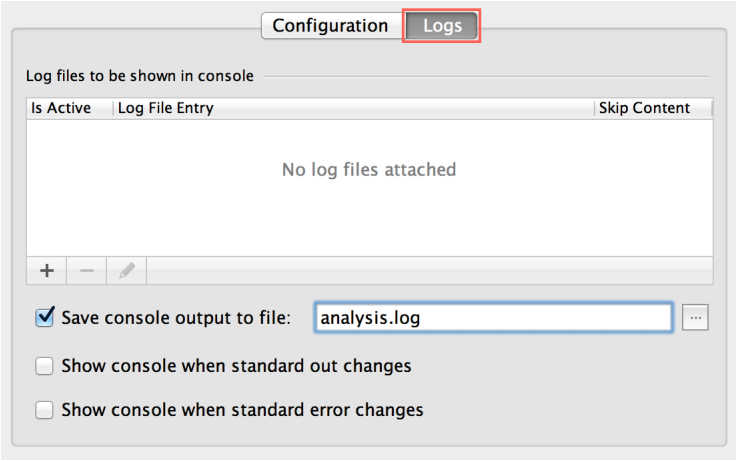
When we want to run things with more control we need to create a configuration by going to *Run → Edit Configurations…* and clicking on the + button *(⌘-n, A-Insert)*. PyCharm will ask what type of configuration we want (that is, if we want to run a Python script, a Django or Pyramid server, and so on) and will show the appropriate options for each type.



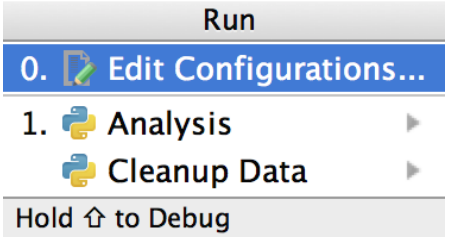
We can choose the script to run, passing parameters if necessary, the Python interpreter, and the working directory. We can also launch an external tool before running the script, which is great if we need to do things like data pre-processing.



In the Logs tab we can add log files to be shown in PyCharm's console during the script execution or save the console output to a file.

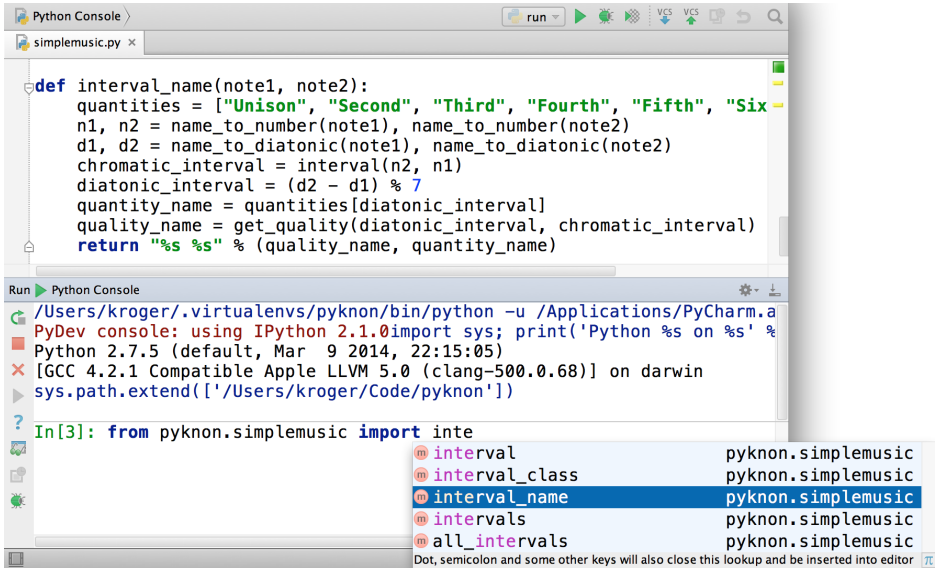


Naturally, we can have multiple configurations; we can switch between them by going to *Run → Run…* *(⌥-C-r, A-C-r)* and picking the one we want.

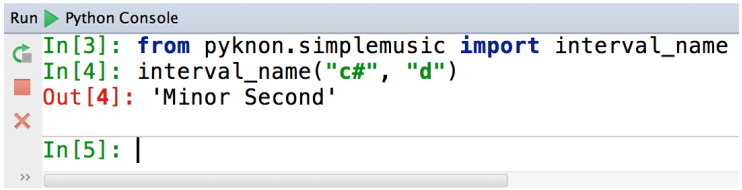


Console

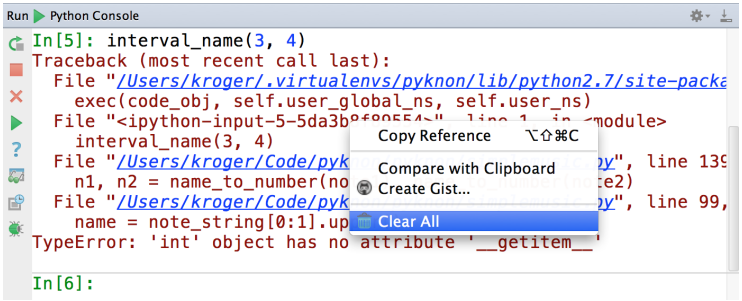
PyCharm has a full-fledged Python console with full code completion available at *Tools → Run Python Console…*. PyCharm will automatically use iPython if it's available.



As expected, we can import and run code in the console:

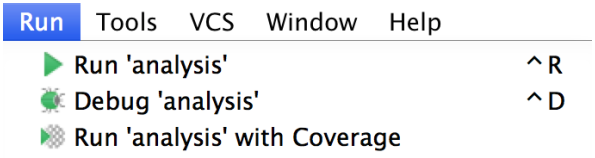


A useful option is Clear All, available in the popup menu (usually on the right button), to clear the console output.

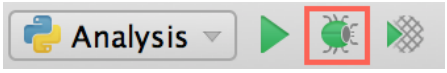


Debugging Code

Running and debugging code in PyCharm work in similar ways (see *Running Code*); we need to create a new configuration or debug a script by going to *Run → Debug… (⌥-C-d, S-A-F9)*. After that it will be available in *Run → Debug '<script name>' (C-d, S-F9)*, where `<script name>` is the name of the Python file without the extension (`analysis`, in our example). As you can see, it works just like running code.



Similarly to the *Run* command, the menubar will now be enabled and we can choose the script to debug by clicking on the insect icon.



One of the main differences between running and debugging a script is that in debugging we want to define breakpoints. In PyCharm, a breakpoint must

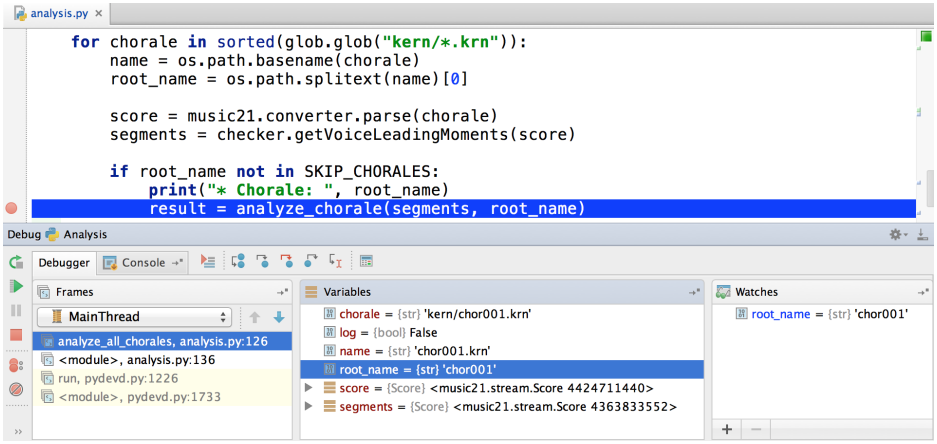
be on a line with working code; it can't be on a line that is blank or starts with a comment. We create breakpoints by clicking on the gutter on the left.

```
        if root_name not in SKIP_CHORALES:
            print("* Chorale: ", root_name)
            result = analyze_chorale(segments, root_name)
...
```

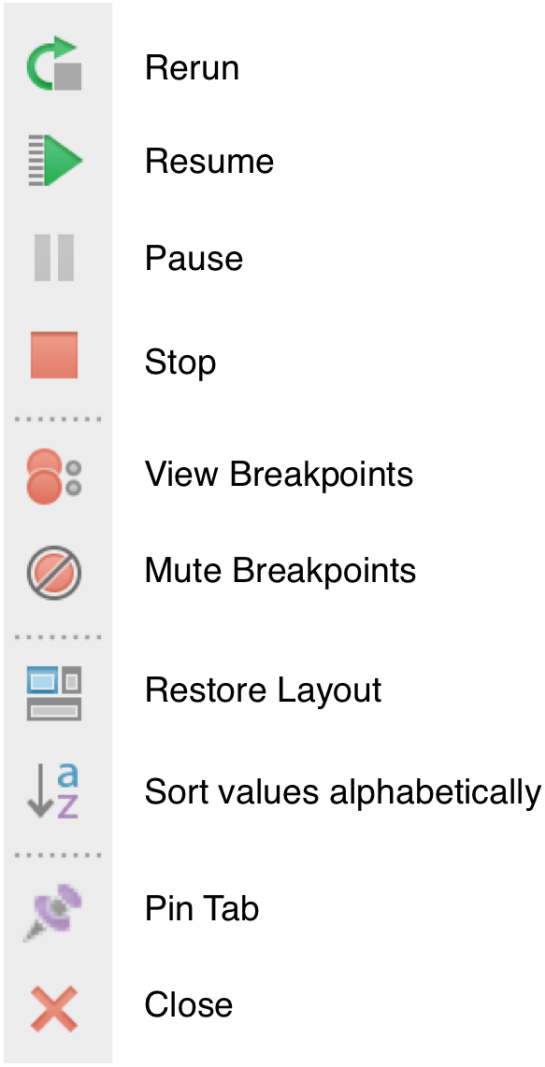
We can also create a temporary breakpoint by Alt-clicking on the gutter. As the name suggests, a temporary breakpoint will be removed after being hit. Notice that the temporary breakpoint icon is different from the regular one. Check the manual for more information about breakpoint icons.

```
        if root_name not in SKIP_CHORALES:
            print("* Chorale: ", root_name)
            result = analyze_chorale(segments, root_name)
...
```

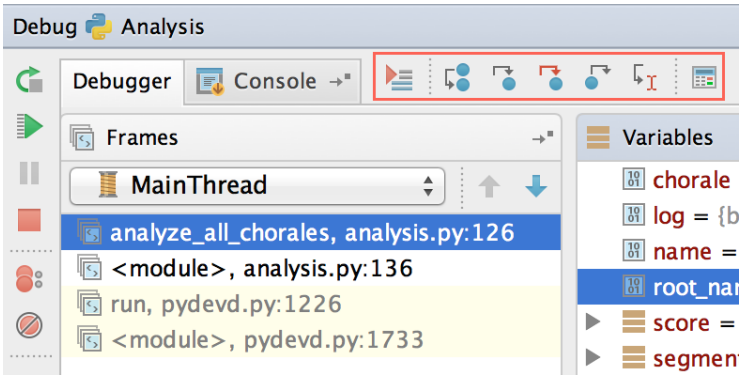
PyCharm's debug tool window has everything we expect in a graphical debugger. From here we can navigate the call stack and inspect the data available in each stack frame. We can keep an eye on some variables by adding them to the watch list.



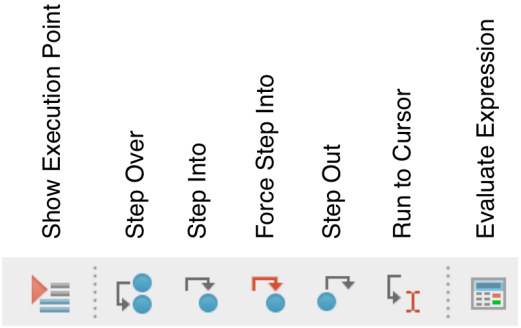
The debug tool window has many of the buttons we have seen in the Run tool window, allowing us to pause, stop, and rerun the script (see *Running Code*). In addition, we can view the breakpoints—also available in *Run → View Breakpoints… (⌘-S-F8, S-C-F8)*—and mute (that is, disable) all breakpoints.



The icons in the top right corner, allowing us to step in and out of the program, are the ones we will use the most:

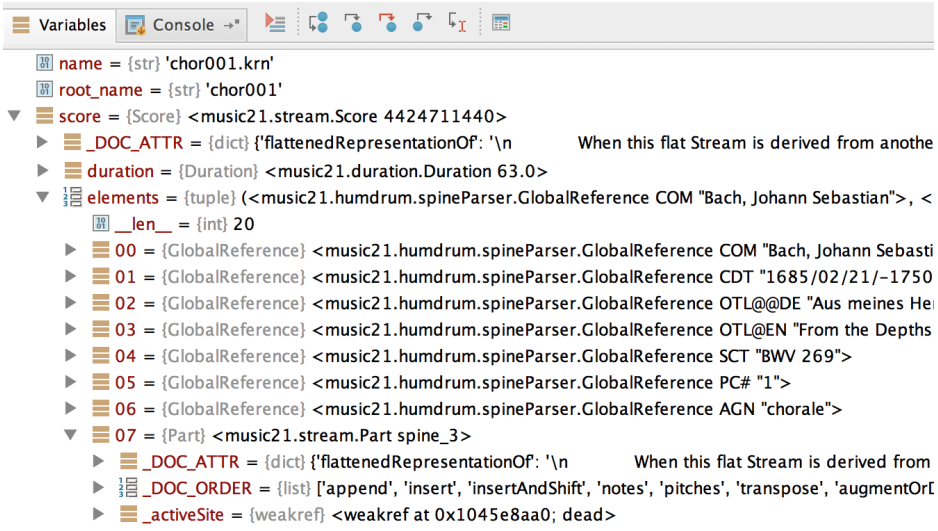


In the following image, we can see each icon with its name. The table below shows the name, shortcut, and description for each icon.

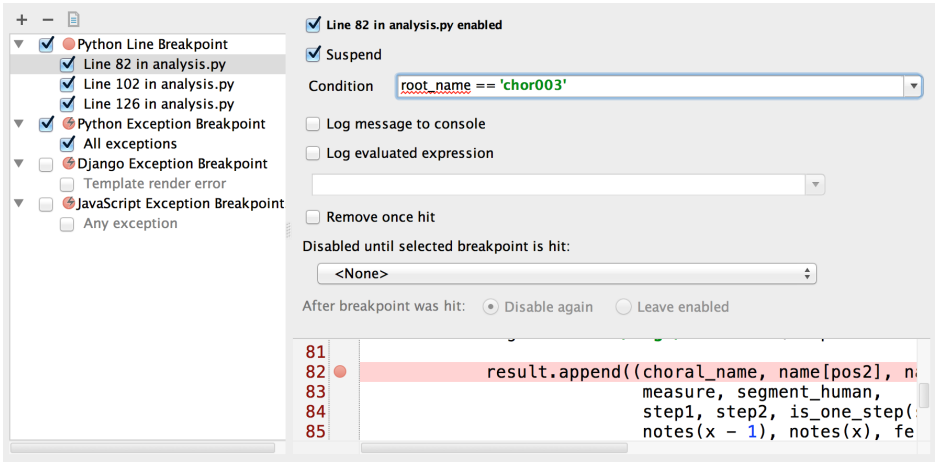


Name	Shortcut	Description
Show Execution Point	*(⌥-F10, A-F10)*	Go back to the execution point. Useful if we visit other files and want to go back
Step Over	*(F8)*	Step to the next line
Step Into	*(F7)*	Descend into a method or function call on the next line
Force Step Into	*(S-⌥-F7, S-A-F7)*	Descend into the method even if it's to be skipped
Step Out	*(S-F8)*	Return to the line where the current function was called
Run to Cursor	*(⌥-F9, A-F9)*	Run the code where the caret is located. Useful to skip stepping over uninteresting code
Evaluate Expression	*(⌥-F8, A-F8)*	Evaluate an expression or code fragment

The variables pane is a great way to explore complex data. It was invaluable when I was learning Django and getting acquainted with the `HttpRequest` object, for instance. In the following example, we can see some `music21` data, especially the complex `Score` class. The Variables pane helps me to inspect attributes such as `duration` and `elements`.

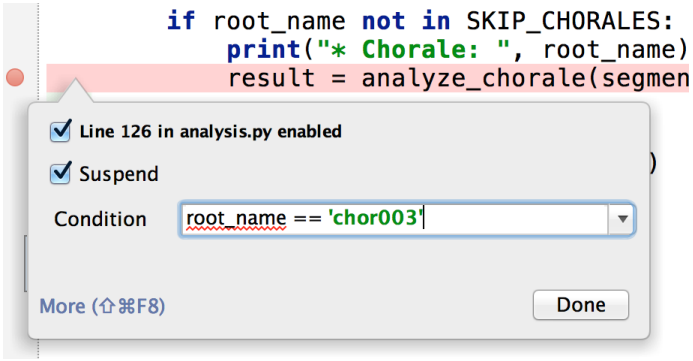


We can see all breakpoints in *Run → View Breakpoints… (⌘-S-F8, S-C-F8)*. From here we can add, remove, or edit breakpoints; and set conditional breakpoints and log messages or expressions.

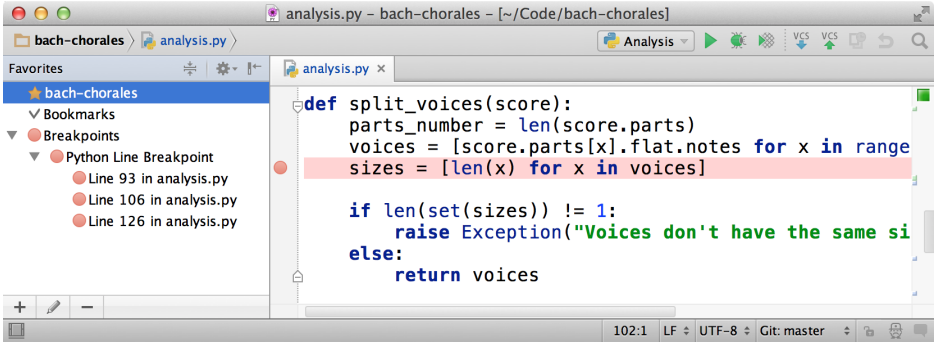


One way to quickly edit a breakpoint is to right-click on the breakpoint on the gutter. A popup will allow us to enable the breakpoint and add a

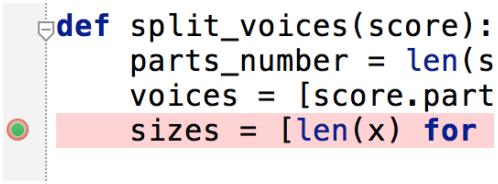
condition.



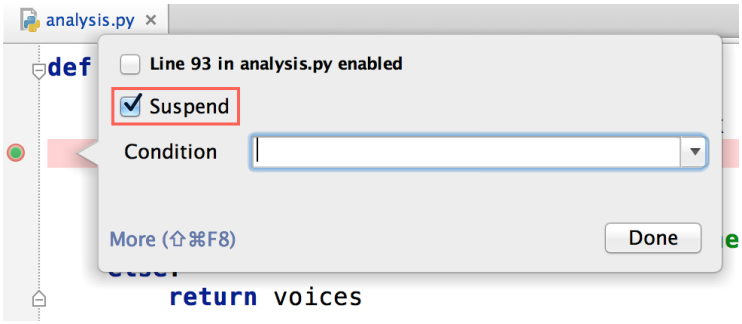
We can also see the breakpoints by selecting the Favorites Tool *(⌘-2, A-2)*.



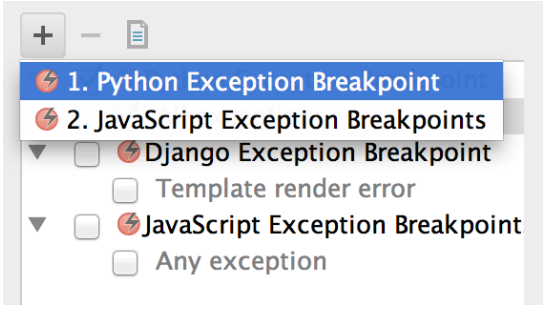
A quick way to disable a breakpoint is by clicking on it with the Alt key. The breakpoint icon will change.



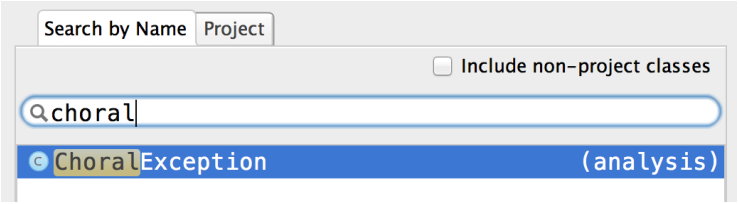
Another way to disable a breakpoint is by right-clicking on it and unchecking the first option ("Line 93 in analysis.py enabled" in the following example).



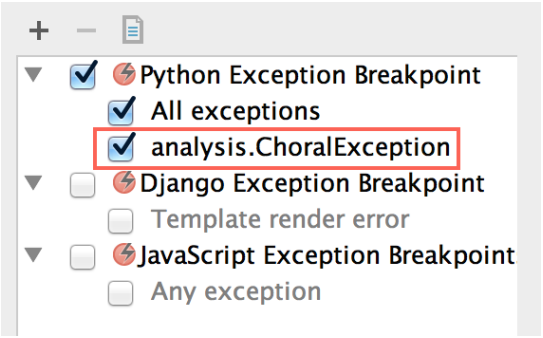
Besides defining breakpoints when a line of code is reached, we can define a breakpoint that is triggered when an exception is thrown. To add an exception breakpoint, we go to *Run → View Breakpoints… (⌘-S-F8, S-C-F8)* and select Python Exception Breakpoint after clicking the + button.



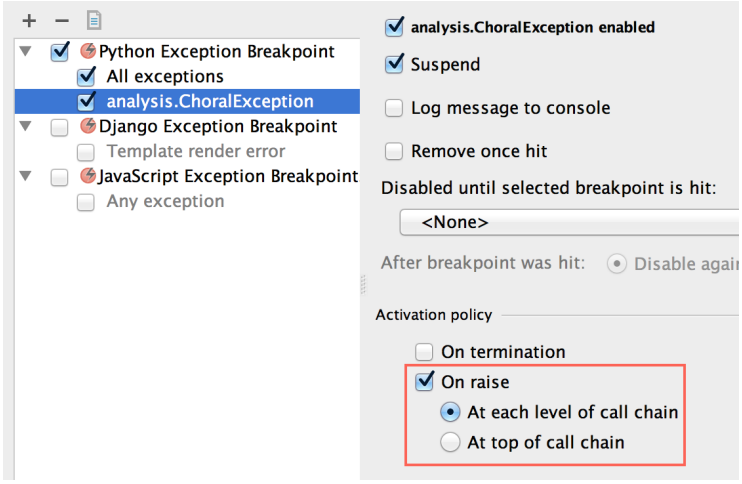
We need to include the exception class by searching for its name. The search field works with both built-in and custom exception classes.



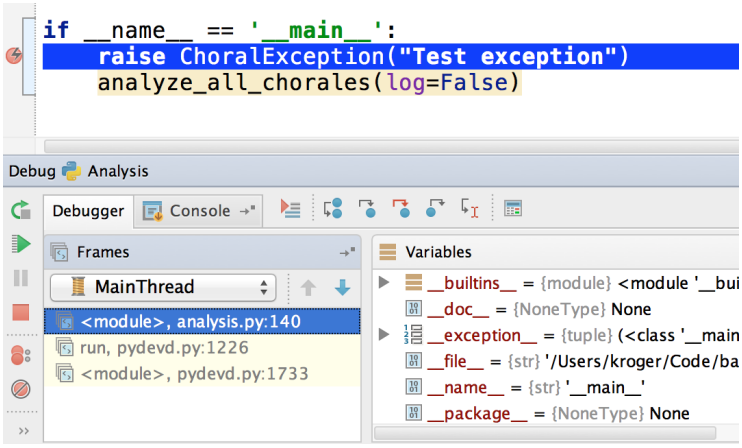
Our new exception now appears on the list, where we can customize it.



In most cases, we want to activate the breakpoint when the exception is raised (the default is to activate the breakpoint when the exception terminates):

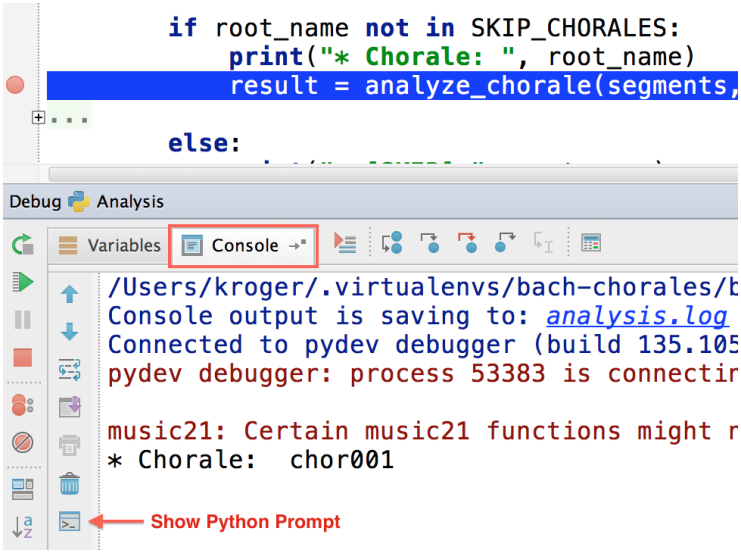


Having configured an exception breakpoint, PyCharm will create the breakpoint automatically when the exception is raised (`ChoralException`, in our example). Notice that the exception breakpoint icon is different from the regular one.

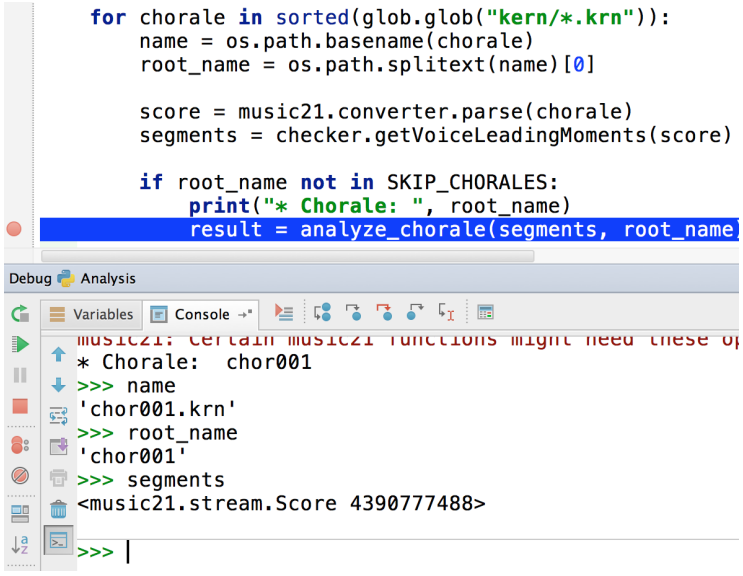


Python Prompt

Sometimes it's useful to be able to access a Python prompt while debugging. We can do this by clicking on the Console tab when the debugger is running and selecting the Show Python Prompt icon.

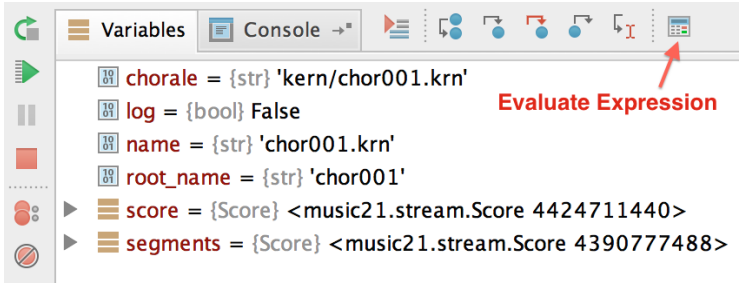


Now we can use the prompt to inspect and change the variables in the current scope.

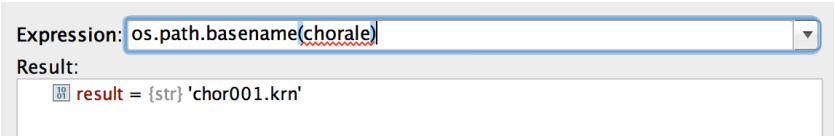


Evaluating Expressions

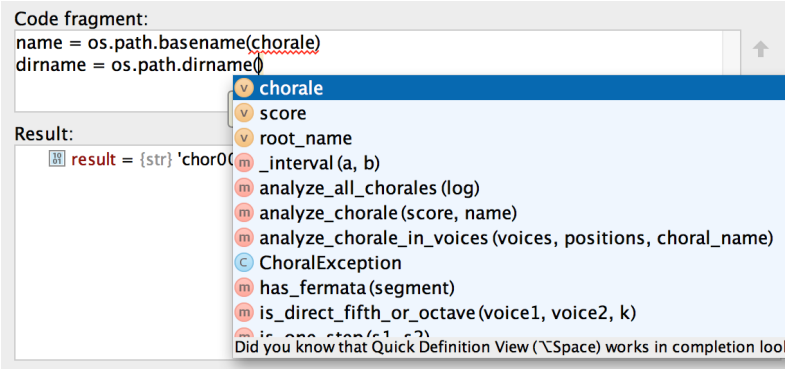
Although it's super-useful to have a full Python prompt available while debugging, the Evaluate Expression feature may be sufficient if we just want to quickly evaluate an expression.



As expected, we can use any variable in the current scope:

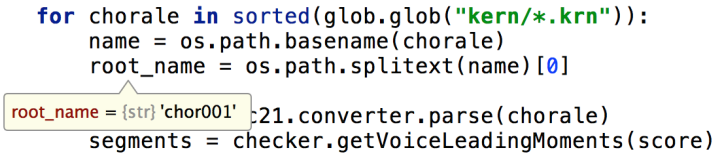


If we select the Code Fragment Mode button, we can add multiple lines of code. Completion works here as usual.

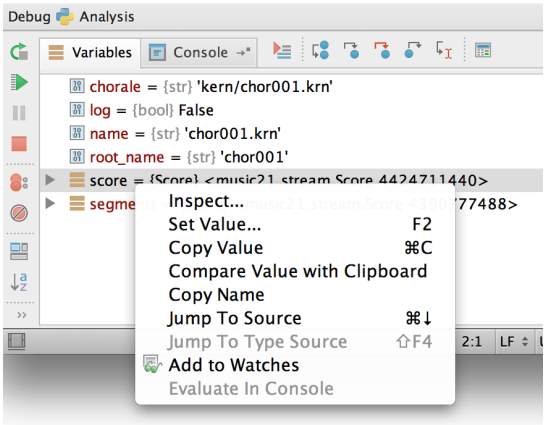


Inspecting and Watching Variables

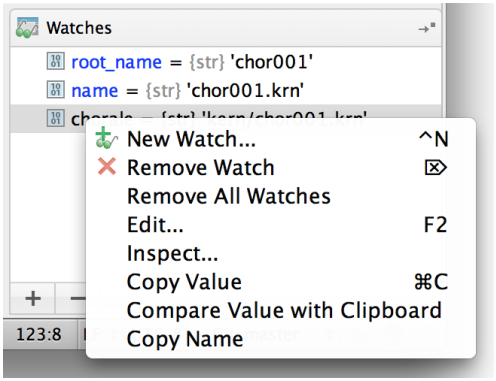
The fastest way to inspect the value of variables while debugging is by hovering the cursor over a variable name. PyCharm will show the variable name and its value.



And, as expected, we can inspect and set a variable value by choosing the correct option in the popup menu on the Variables pane.



The same apply to watched variables; we can edit and inspect them by using the popup menu.



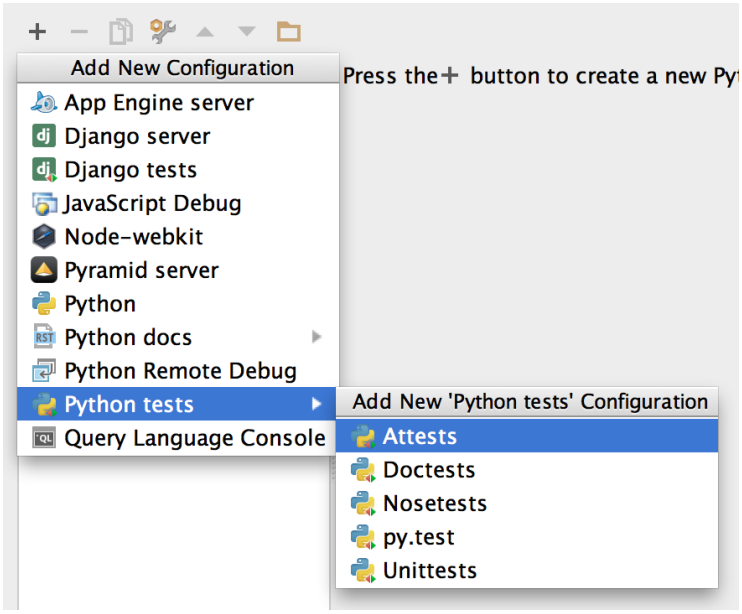
Remote Debugging

PyCharm has support to debug code remotely. Probably, the most popular and useful way is to debug code in a Vagrant box (see *Vagrant* and *Virtualenv and Packages*).

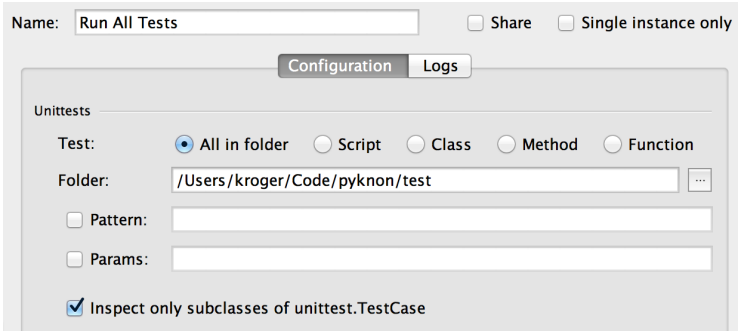
Testing Code

Running Tests

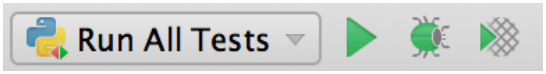
To run a test for the first time, we need to create a new run configuration in *Run → Edit Configurations…* and select a test runner in "Python tests." In the following examples I'm using `unittest`, but the process is similar for the other test runners.



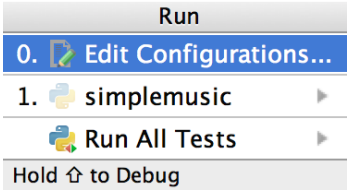
Next, we need to select the test files. I like to run all tests in a folder named "test," but you could choose a script, class, method, or function as well:



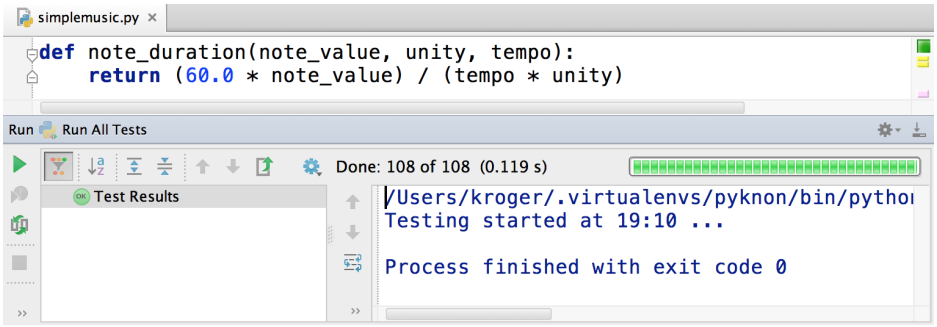
The name of the configuration in our example is "Run All Tests" and it's available in the toolbar after we create it.



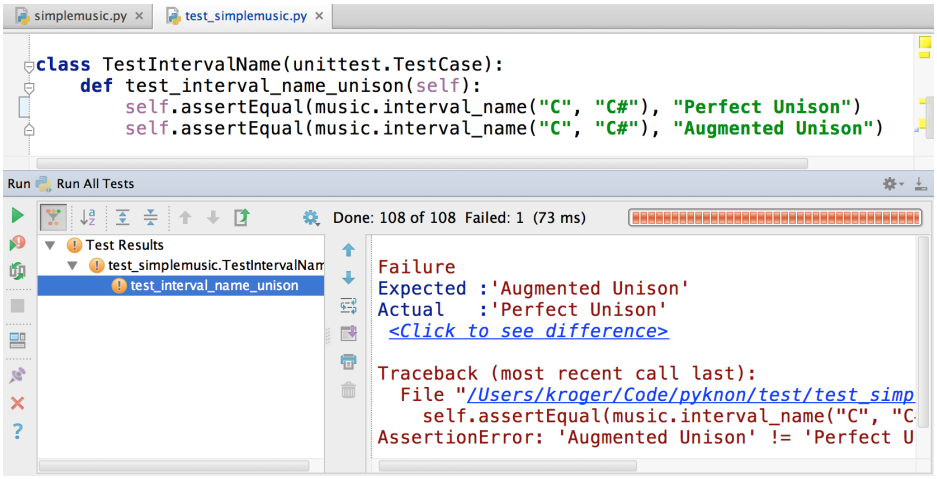
Since tests are just another run configuration, we will use the same commands to run a script such as *Run → Run '<configuration name>' (C-r, S-F10)*. As we have see in *Running Code*, we can switch running configurations quickly in *Run → Run… (⌥-C-r, A-C-r)*.



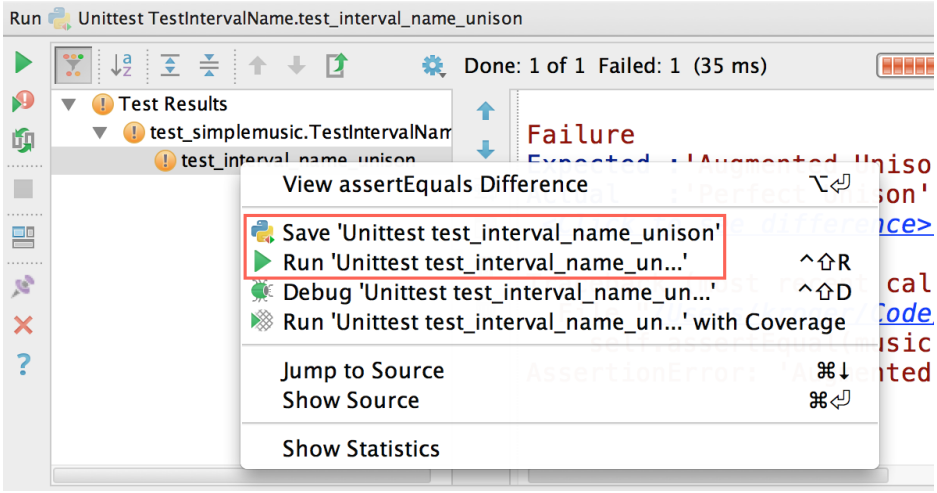
If all tests pass, we get a nice green bar and a status message.



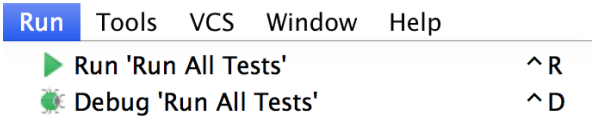
But things are more interesting when tests fail. On the left we can see the tests that failed, with a message on the right. We can click on the link to go to the test implementation.



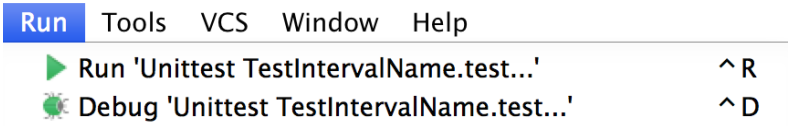
We can also run only one test, either by right-clicking on the test in the Test Results pane and choosing *Run '<test name>'*, or by selecting the test and using the shortcut *(S-C-r, S-C-F10)*.



When we run our test case ("Run All Tests" in our example), it'll be available in the *Run* menu:



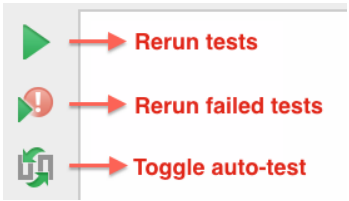
When we run only one test, it will replace the previously run command in the menu:



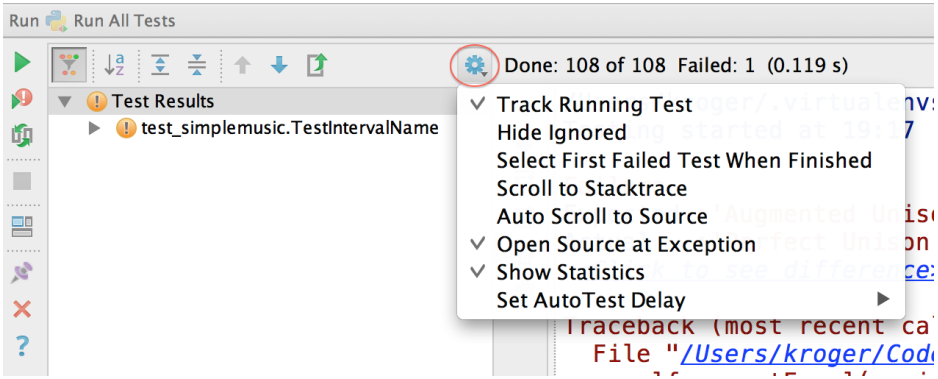
It may sound complicated, but it's very simple and useful. PyCharm remembers the last command in the menu, so we can run it again with *(C-r, S-F10)*.

We have three test-related buttons in the Run tool window: rerun tests, rerun

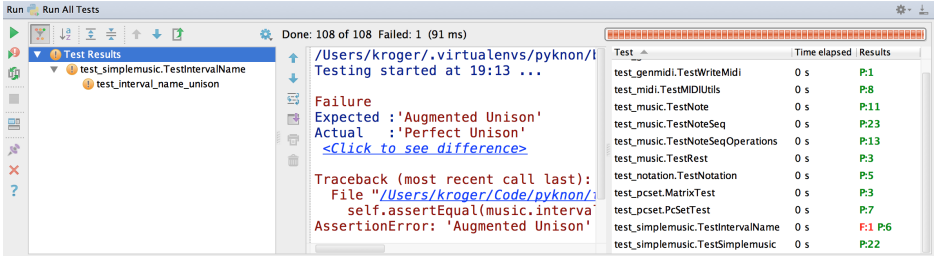
only the failed tests, or rerun the tests automatically when the source code has changed.



There are some useful options such as Scroll to Source and Open Source at Exception in the settings icon:



The option Show Statistics is particularly useful to see a list of test cases and the results.

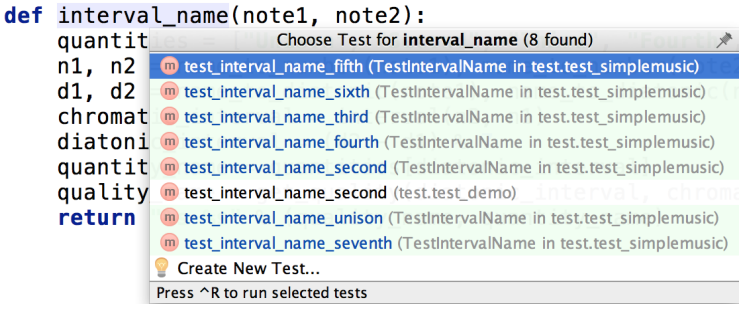


By clicking on a test case, we can see its tests and results. In the following example, we can see the details for the case named `TestIntervalName`:

Test	Time elapsed	Results
test_interval_name_fifth	0 s	Passed
test_interval_name_fourth	0 s	Passed
test_interval_name_second	0 s	Passed
test_interval_name_seventh	0 s	Passed
test_interval_name_sixth	0 s	Passed
test_interval_name_third	0 s	Passed
test_interval_name_unison	0 s	Assertion failed

If we right-click on a test, we can run it, debug it, and go to its definition:

When writing and debugging unit tests, a common pattern is to switch back and forth between the code and its tests. With the caret anywhere in a function or method definition, we can switch to its test by going to *Navigate → Test (⌘-S-t, S-C-t)*. PyCharm will show a list if it finds more than one test:

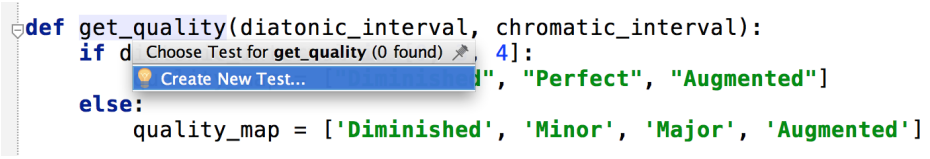


The opposite is also true; we can navigate from a test to the test subject by going to *Navigate → Test Subject (⌘-S-t, S-C-t)*. In my experience, this doesn't work well; often PyCharm won't recognize the right test subject. A more reliable way to jump from the test to the test subject is by jumping to the function declaration with *(⌘-Click, C-Click)*, since we are using it in the test anyway (see *Finding Your Way in the Source Code*).

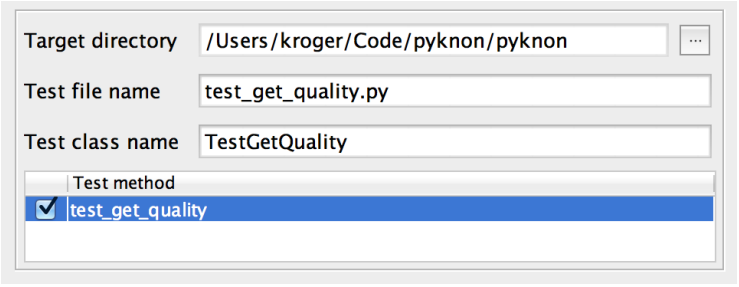
```
self.assertEqual(music.interval_name("D", "A"), "Perfect Fifth")
self.assertEqual(music.interval_name("C", "Gb"), "Diminished Fifth")
```

Creating Tests

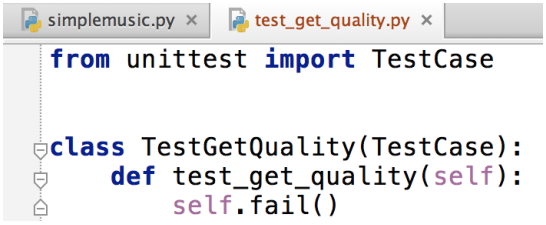
When we try to navigate from the body of a function, method, or class to a test with *Navigate → Test (⌘-S-t, S-C-t)*, PyCharm will offer to create a new test if it can't find an existing one.



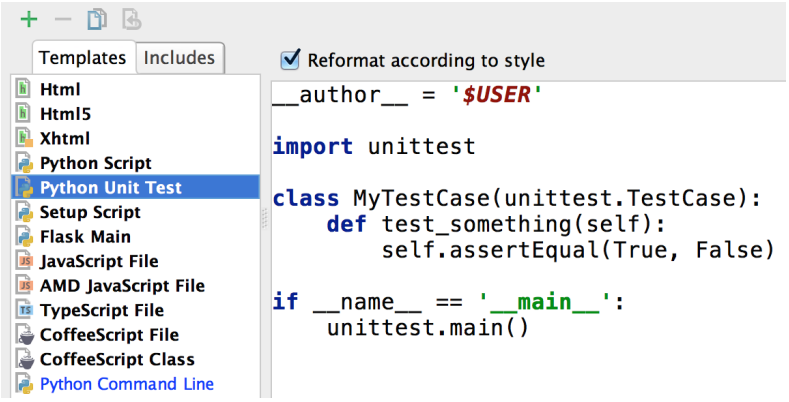
We will be able to set the target directory, test file name, and test class name. PyCharm will suggest one or more test methods by default; click on the checkbox to accept the suggestion.



PyCharm creates tests using `unittest`, even if we are using a different test runner. The following is the resulting code for the previous example:

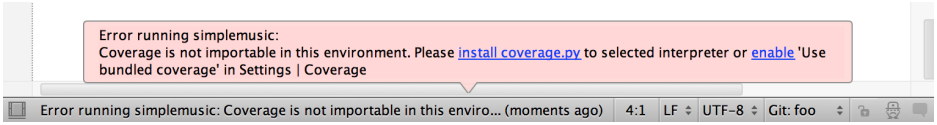


But since the generated code is created by a file template (see *File Templates*), we can create our own file template, or modify the default one if we want to use a different syntax for the tests, such as the one used by `py.test`.

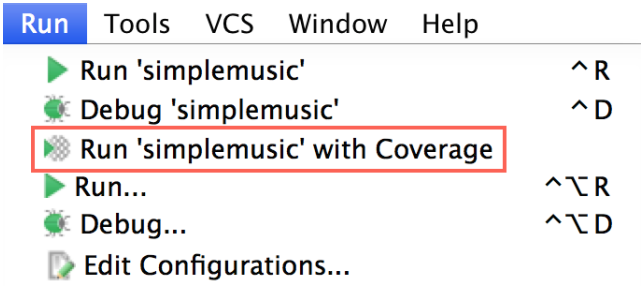


Code Coverage

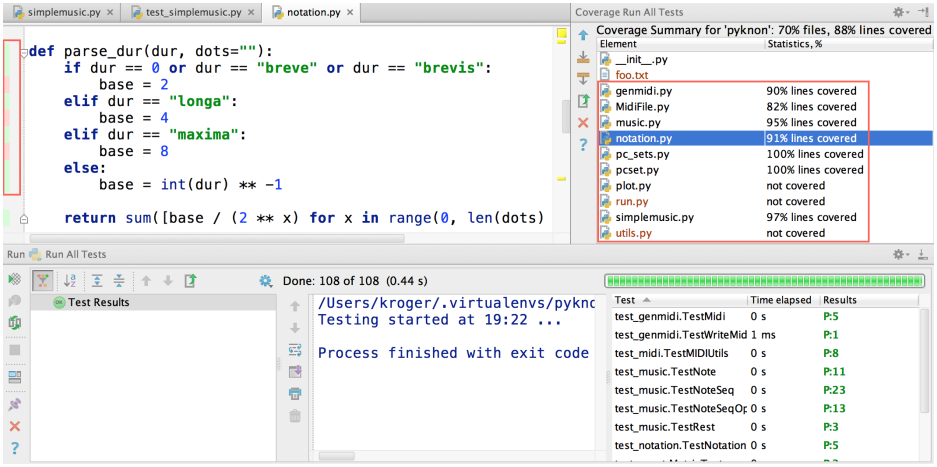
Code coverage is one of the most useful metrics in programming. It allows us to see precisely how much code is being tested and what lines of code are not being tested. The standard tool for coverage in Python is coverage.py, created by the always-remarkable Ned Batchelder. PyCharm uses `coverage.py` to generate the coverage reports, so we need to either install it or use the version bundled in PyCharm:



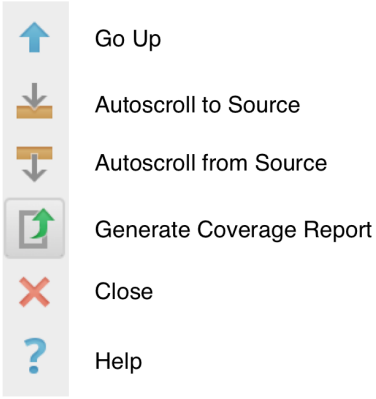
To see the code coverage, we need to run the code with coverage support by going to *Run → Run '<script>' with Coverage*.



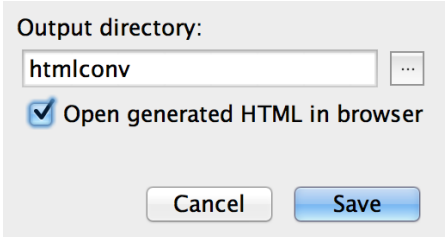
This will run `coverage.py` on the background and present the summary on the Coverage Summary pane on the right. On the far left, the colors in the gutter will show if the line has been tested (green)—that is, if the line is covered by unit tests—or not (red).



The tool `coverage.py` generates an attractive HTML report that some may prefer (I know I do). To generate the report, click on the Generate Coverage Report icon in the Coverage Summary pane:



PyCharm will ask where we want to save the report and if we want to open it in the browser right away:



The HTML has all the metrics we expect, and we can click on the file names to see the annotated source code.

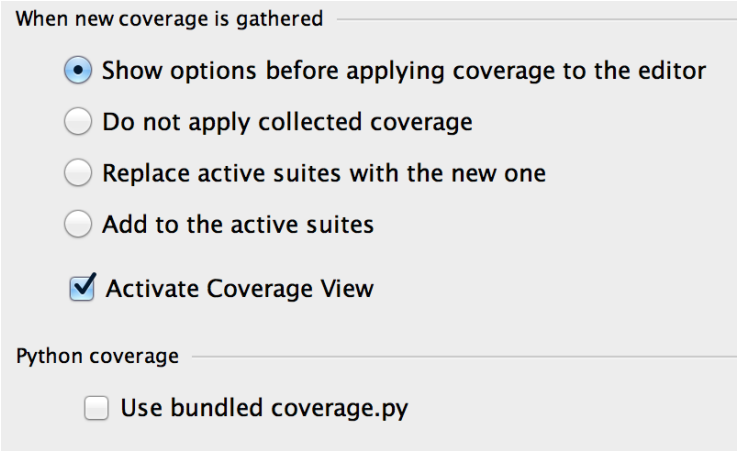
Coverage report: 93%

Module	*statements*	*missing*	*excluded*	*coverage*
/Users/kroger/Code/pyknon/pyknon/MidiFile	496	85	0	83%
/Users/kroger/Code/pyknon/pyknon/__init__	0	0	0	100%
/Users/kroger/Code/pyknon/pyknon/genmidi	40	4	0	90%
/Users/kroger/Code/pyknon/pyknon/music	163	8	0	95%
/Users/kroger/Code/pyknon/pyknon/notation	46	4	0	91%
/Users/kroger/Code/pyknon/pyknon/pc_sets	1	0	0	100%
/Users/kroger/Code/pyknon/pyknon/pcset	37	0	0	100%
/Users/kroger/Code/pyknon/pyknon/simplemusic	85	2	0	98%
/Users/kroger/Code/pyknon/test/test_demo	14	11	0	21%
/Users/kroger/Code/pyknon/test/test_genmidi	33	0	0	100%
/Users/kroger/Code/pyknon/test/test_midi	194	2	0	99%
/Users/kroger/Code/pyknon/test/test_music	266	0	0	100%
/Users/kroger/Code/pyknon/test/test_notation	48	0	0	100%
/Users/kroger/Code/pyknon/test/test_pcset	36	0	0	100%
/Users/kroger/Code/pyknon/test/test_simplemusic	143	0	0	100%
Total	**1602**	**116**	**0**	**93%**

In the following example, I neglected to create a test to exercise a few lines in `parse_dur`. When creating new tests, my priority should be to correct that.

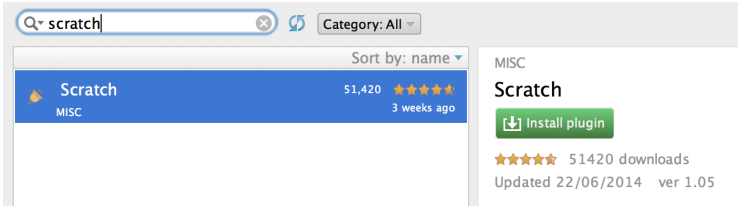
```
26 def parse_dur(dur, dots=""):
27     if dur == 0 or dur == "breve" or dur == "brevis":
28         base = 2
29     elif dur == "longa":
30         base = 4
31     elif dur == "maxima":
32         base = 8
33     else:
34         base = int(dur) ** -1
```

There are a few options in *Settings → Build, Execution, Deployment → Coverage* to define how the coverage data will be processed. You can find more information in the manual.

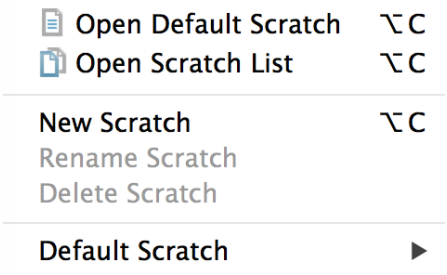


Scratch File

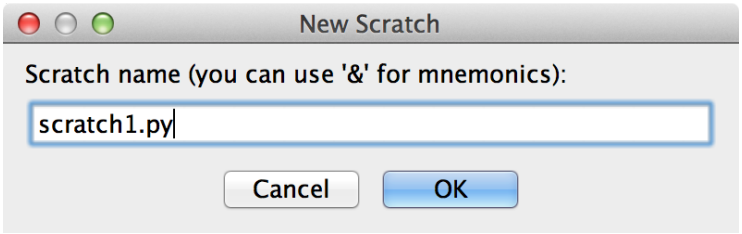
Sometimes we want to run arbitrary code for quick calculations without creating a file in the disk. We can accomplish this with the Scratch plugin. To install it, go to *Settings → Plugins*, click in "Browse repositories…" and search for "Scratch."



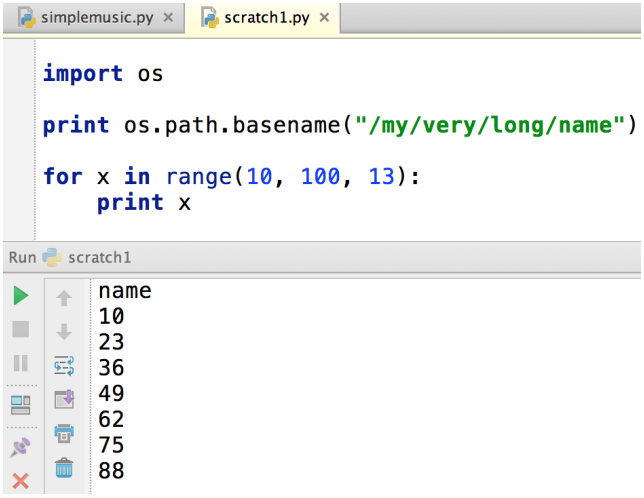
Scratch has many options, such as *A-c A-c* to open the default scratch, *A-c A-s* to open a list with all scratches, and *A-c A-a* to add a new scratch. You can see a full list of shortcuts in its webpage. These actions can also be accessed by going to *Tools* → *Scratch*.



When we create a new scratch, we give it a name:



We can run some basic computations in the scratch buffer, with some limitations. It can import built-in modules, but it won't import custom modules. This plugin was originally intended to edit plain text data (for quick notes) and XML data.

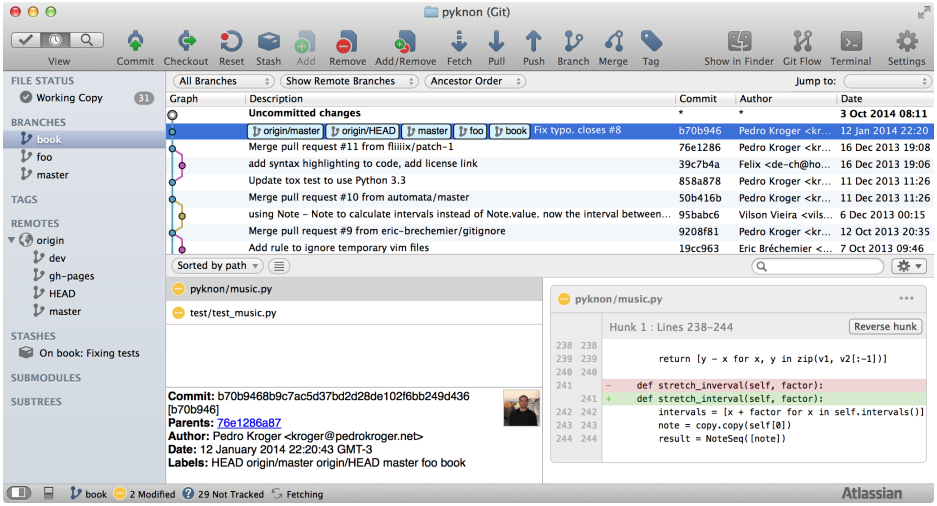


5 | Tools

Version Control

There are three main ways to use version control: in the command line, using a dedicated GUI tool such as Tower (Git) or SourceTree (mercurial and Git), or using the features in the IDE. PyCharm, of course, offers the last option.

Since I use different editors (Emacs, Vim, TextMate) and IDEs (PyCharm, Xcode), I like to use version control in the command line and in a dedicated GUI tool, so I don't need to learn how each editor and IDE implements the features for version control. Nevertheless, having these features available in the IDE is very useful. In the following image we can see a repository in SourceTree.



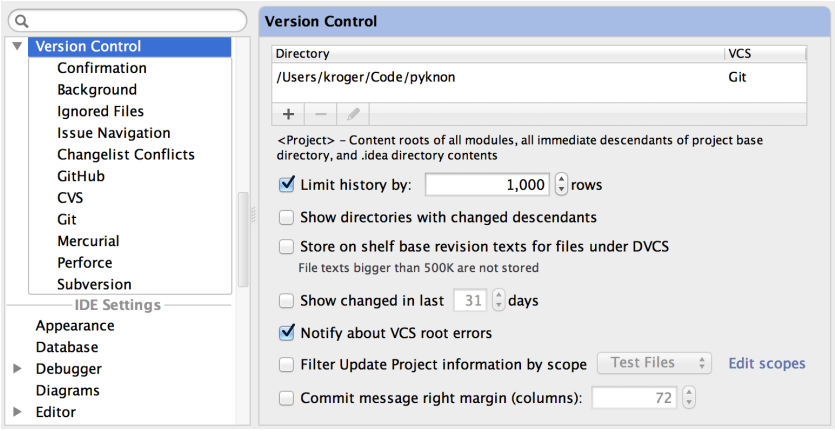
PyCharm has many useful features for version control. Although PyCharm has many features to deal with distributed version control systems, I find PyCharm's implementation a little too file-oriented; it works well with CVS and Subversion but doesn't quite match how tools like Git and Mercurial work. For instance, there's no way to commit parts of a file (hunks) in PyCharm (you may want to vote in this issue: IDEA-63201).

In this section I will use Git and GitHub as examples, but the main process is the same os with the other version control systems. An introduction to version control and Git is beyond the scope of this book, but you can find great documentation and books online: The Git Documentation, Pro Git by Scott Chacon, Mercurial: The Definitive Guide by Bryan O'Sullivan, and Version Control with Subversion by Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato.

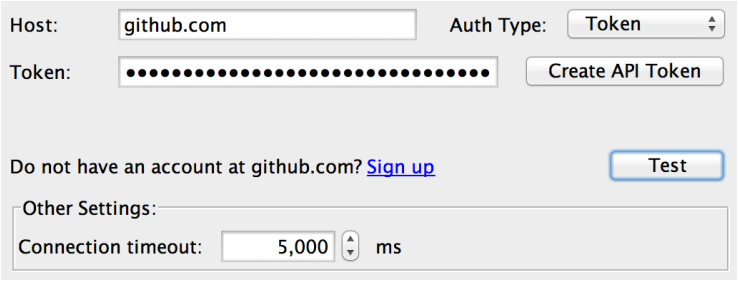
Configuration

PyCharm has many options for version control in *Settings → Version Control*, including specific tools such as CVS, Git, Mercurial, Perforce, and

Subversion.



We can also set up direct access to GitHub by configuring a login. I suggest you use an access token instead of your username and password.

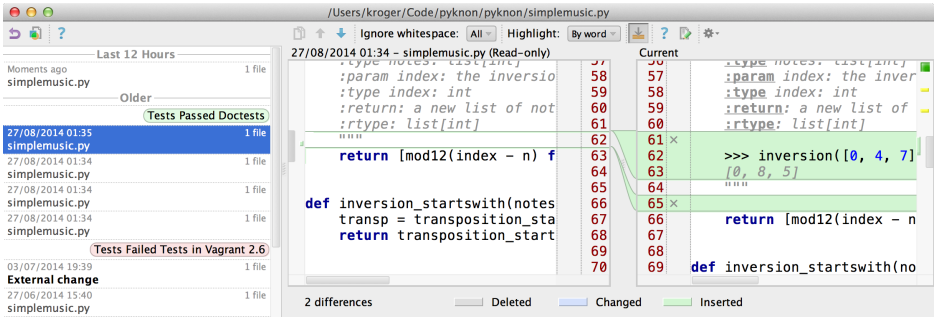


Local History

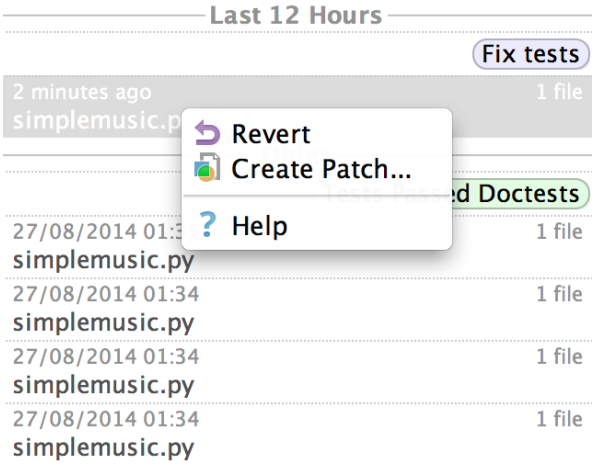
Before we delve into the version control features, we must check one of the coolest and least-known features of PyCharm: Local History.

PyCharm saves the changes in the source code even if we are not using a version control tool. We can see the past history in *VCS → Local History →*

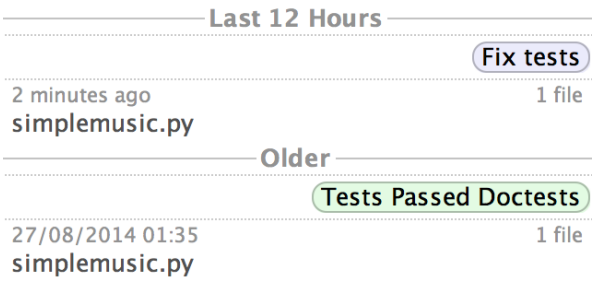
Show History.



We can easily revert a change by using the popup menu:



And we can even add a label to a change in *VCS → Local History → Put Label…*.

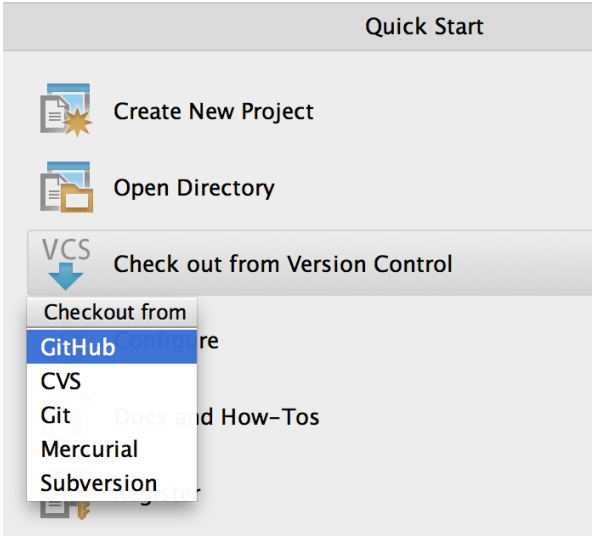


We can also see the history for only a selected portion of code in *VCS → Local History → Show History for Selection*.

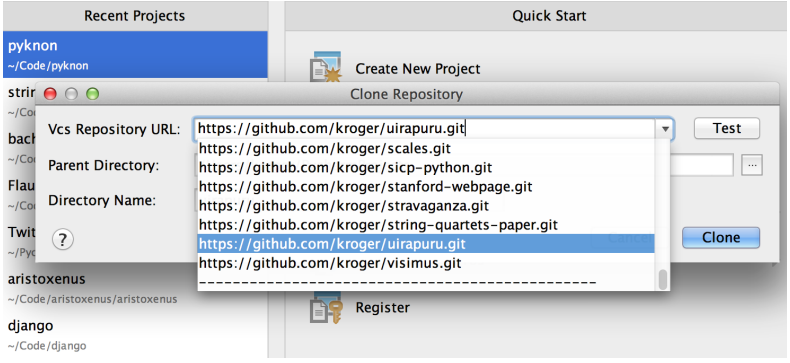
Keep in mind that this feature is not a substitute for a real version control system. Local changes are saved in a local cache only; they are not shared with other users, and they will be deleted when a new PyCharm version is installed or when the cache is cleared. Also, PyCharm will not save the history of binary files.

Checkout from Version Control

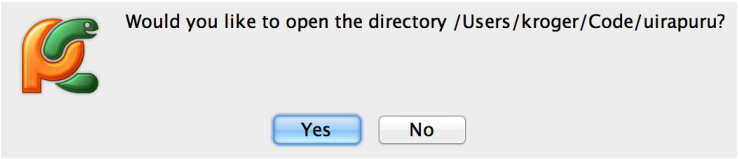
Checkout from a repository couldn't be simpler. We click Check out from Version Control in the Quick Start window and choose the repository from which we want to check out.



We need to enter the repository URL, the location where the code will be saved, and the checkout name. If access to GitHub is configured, PyCharm will show a list with all available repositories:

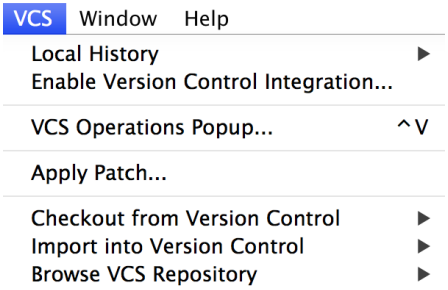


PyCharm even offers to open the checked-out code.

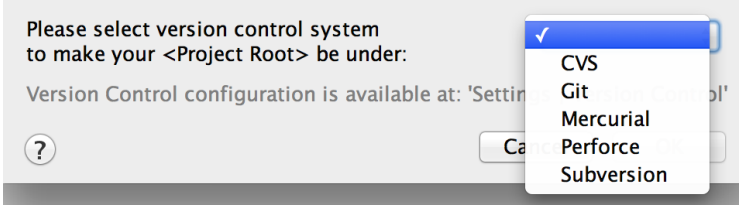


Enable Version Control to a Existing Project

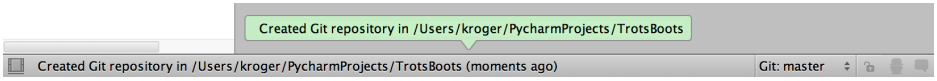
If a project is already using version control, PyCharm will detect the tool automatically. If the project is not under version control, we can add it by going to *VCS → Enable Version Control Integration…*.



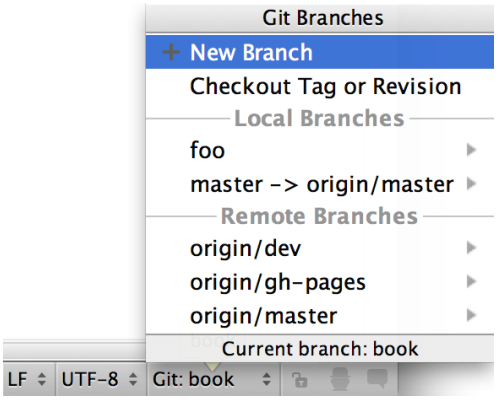
PyCharm will ask what version control system we want to use.



And it will notify us when the repository is created.



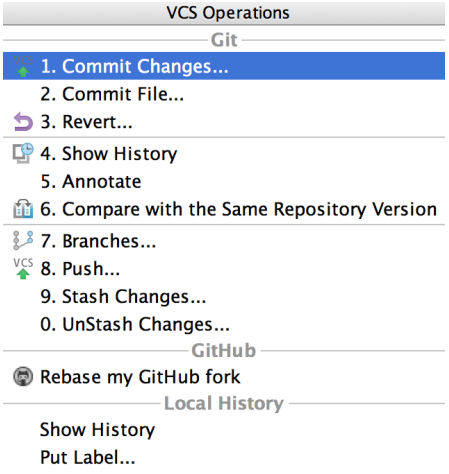
We can see the current branch in the bottom right. By clicking on the branch name we can switch and create branches on the *Branches* popup menu.



PyCharm will detect automatically whether a project is already using version control.

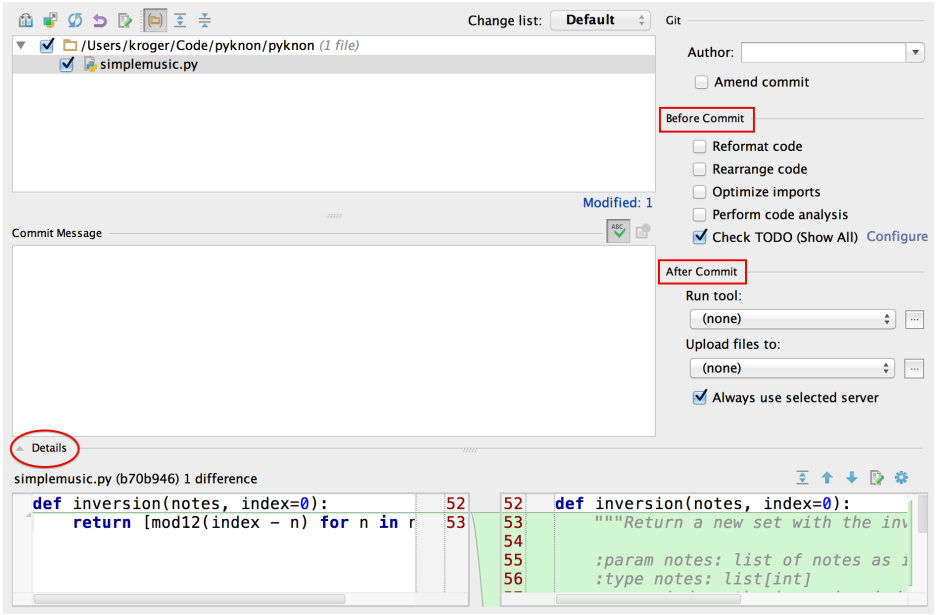
Using Version Control

The *VCS* menu has many options to deal with version control, but the easiest way is to use the VCS Operations popup in *VCS → VCS Operations Popup… (C-v, A-backquote)*. It makes the main version control operations easily accessible. With the popup opened, we can type the number on the left to quickly access the operation.

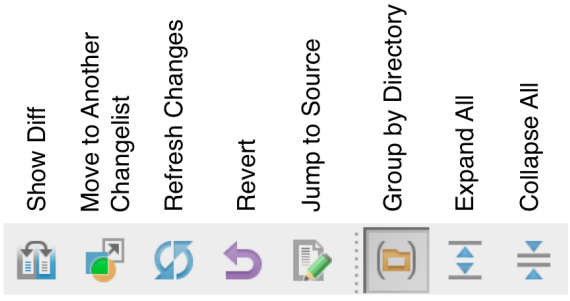


Commit Changes

To commit changes, we go to *VCS → Commit Changes… (⌘-k, C-k)*. We can select the files to commit, write a commit message, and see what changed in the Diff view. On the right side of the commit changes dialog we can choose actions to happen before and after the commit, such as reformatting and deploying the code. I like to expand the Details disclosure triangle in the bottom to show the contextual diff for the selected file.



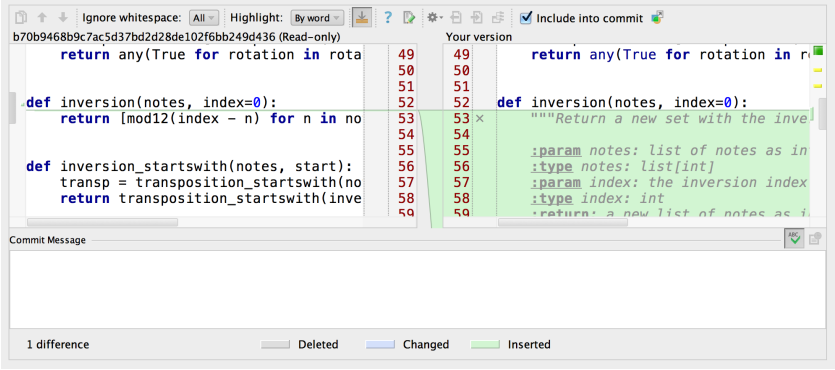
On the top left we have an icon bar with a few actions:



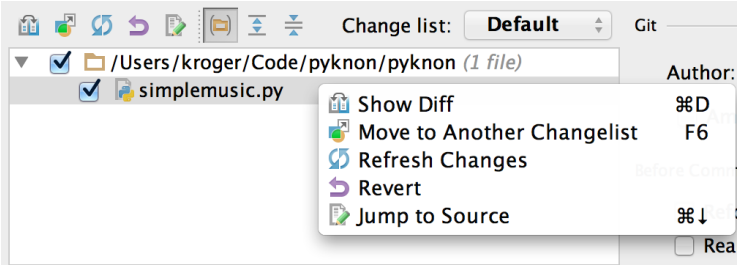
The following table summarizes the main actions. The manual has a complete description.

Name	Shortcut	Description
Show Diff	*(⌘-d, C-d)*	Show the differences in the file since the last commit
Move to Another Changelist	*(F6, F6)*	Move the file to a different changelist
Refresh Changes	*(no key, C-F5)*	Reload the list of changed files
Revert		Revert all changes in the file since the last commit
Jump to Source	*(⌘-↓, F4)*	Open the selected file in the editor
Group by Directory	*(⌘-p, C-p)*	Toggle between flat and tree view
Expand All	*(⌘-+, C-NumPad+)*	Expand all nodes in tree view
Collapse All	*(⌘- -, C-NumPad-)*	Collapse all nodes in tree view

We can access a full Diff window by clicking on the first icon at the top left.



The popup menu has a few options such as showing the diff, moving the changes in the selected file to another changelist, and reverting the changes:

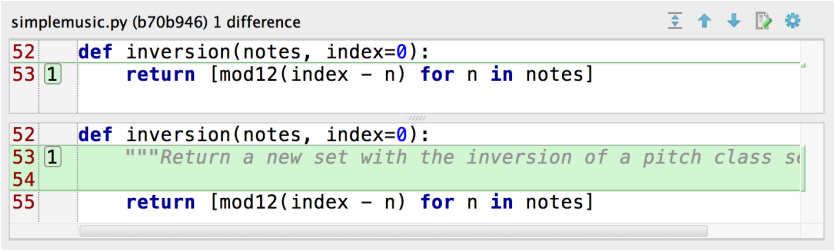


There's a toolbar on the bottom right corner when the Details disclosure is expanded:

The following table summarizes the actions.

Name	Shortcut	Description
More/Less Lines		Choose the number of contextual lines in the diff
Previous Change	*(S-F7, S-F7)*	Go to the previous change
Next Change	*(F7, F7)*	Go to the next change
Edit Source	*(⌘-↓, F4)*	Open the current change in the editor
Settings		Change how the diff view should be displayed

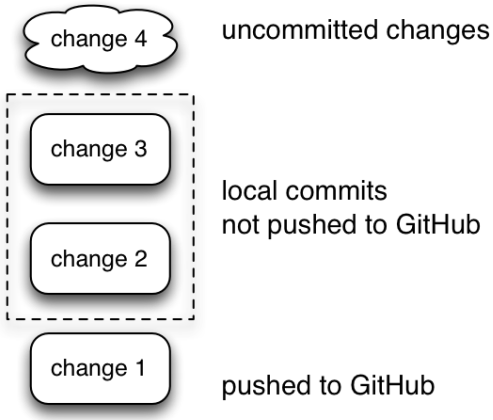
We can change the diff view to be displayed horizontally or vertically and to use soft wraps. In the following image we see the diff displayed vertically:



Compare Files

PyCharm has four actions to compare files: *VCS → Git → Compare with the Same Repository Version*, *VCS → Git → Compare with Latest Repository Version*, *VCS → Git → Compare with…*, and *VCS → Git → Compare with Branch…*.

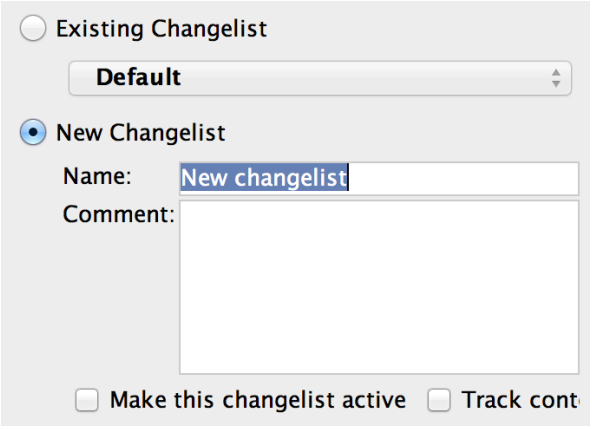
The difference between *Compare with the Same Repository Version* and *Compare with Latest Repository Version* is that the first will compare the current code against the last local commit, while the second will compare it with the last change pushed to the server. The following diagram illustrates how it works. To make it less abstract I will use GitHub as an example for a remote server, but anything will do. The action *Compare with the Same Repository Version* will show the difference between changes 4 and 3, while *Compare with Latest Repository Version* will show the difference between changes 4 and 1.



With *Compare with..*, we compare the latest changes against a specific commit, while with *Compare with Branch...* we compare with a specific branch.

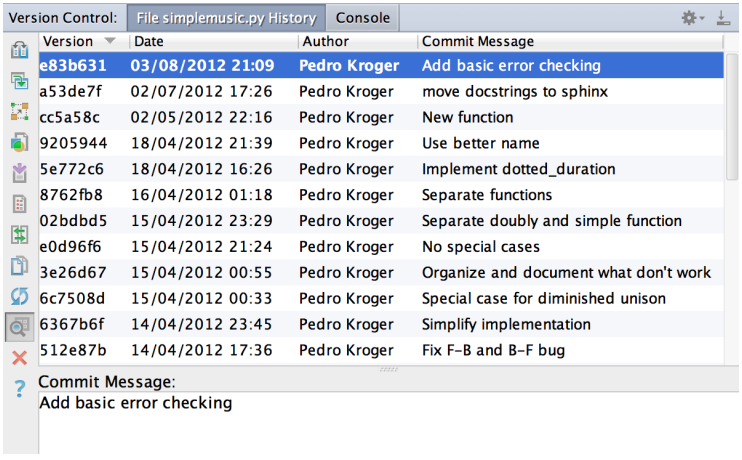
Changelists

In PyCharm, changelists are a way to group related changes. They work in files; we can group changes in files A, B, and C in one changelist and changes in files D and E in another changelist, but we can't select different hunks on the same file to different changelists. Because of this limitation, I don't find changelists to be very useful while working with Git. I prefer to use the interactive staging feature of Git.



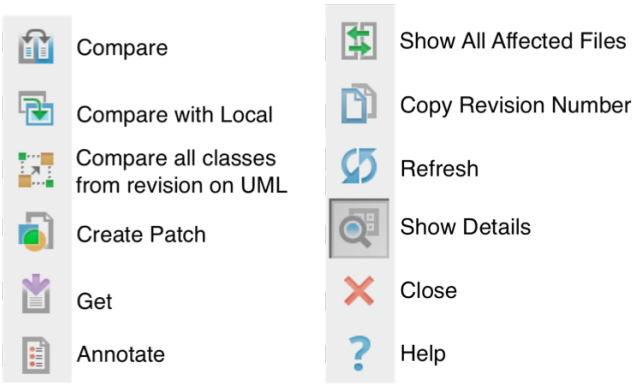
History and Log

The command *VCS → Git → Show History* shows the history of the current file. While useful, this may not be what you want if you are looking for something like `git log`.



On the Show History tool window we have many available actions, such as

creating patches and comparing the selected file with other versions.

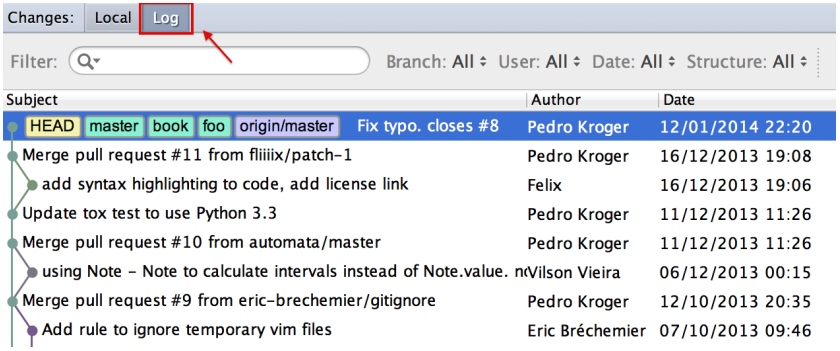


The following table summarizes the actions.

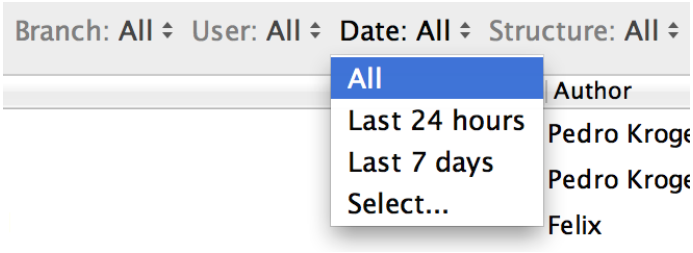
Name	Shortcut	Description
Compare	*(⌘-d, C-d)*	Compare the selected version with the last pushed commit
Compare with Local Version		Compare the selected version with the last local commit
Compare Classes on UML	*(⌘-S-d, S-C-d)*	Compare the selected version with the last file change
Create Patch		Create a patch for the last change against the selected commit
Get		Get the selected version for the file
Annotate		Show annotation (blame) on the left side
Show All Affected Files	*(⌘-C-a, S-A-a)*	Show all changed files in the selected commit
Copy Revision Number		Copy the commit SHA number to the clipboard
Refresh		Refresh the file history
Show Details		Show the long commit message

The action Get will check out the selected version for the current file only, even if other files have been changed in the commit.

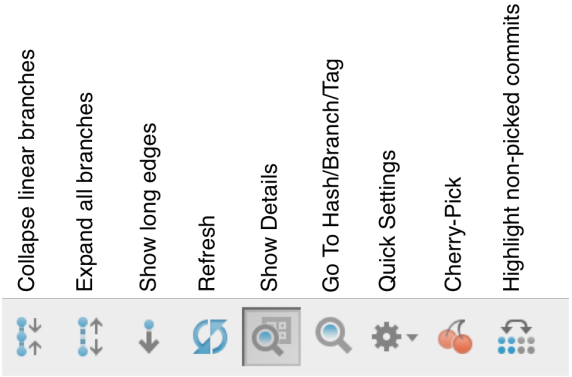
To see something that resembles a Git log, we need to select the Log tab in the Changes tool window in *View → Changes (⌘-9, A-9)*.



Here we can filter the commits by branch, user, date, and files (in the "Structure" list).

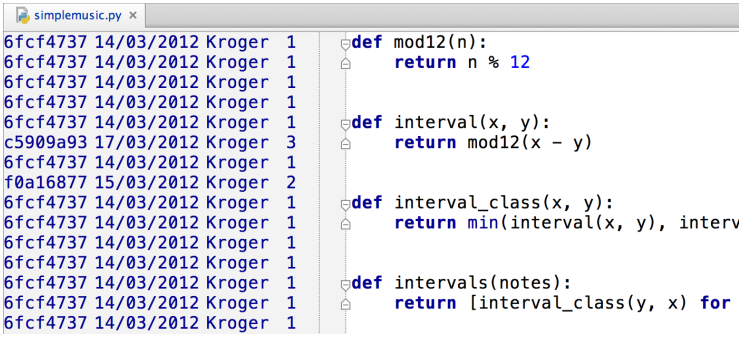


The menu bar has a few actions to visualize and navigate branches, commits, and tags. The Cherry-Pick action is very useful (if not dangerous!) to apply a specific commit from a branch to the current one. Make sure the Branch filter is set to All.

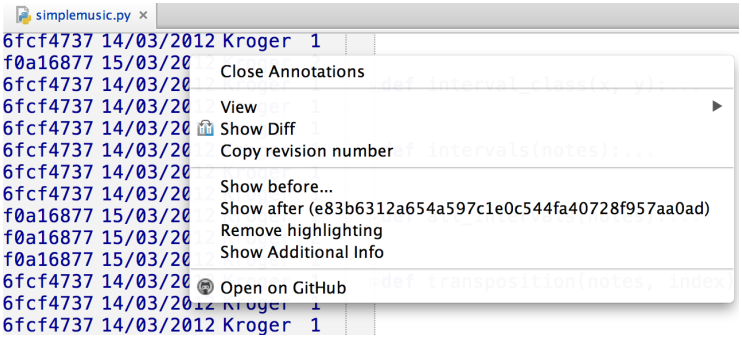


Annotate (blame)

PyCharm can show annotations on the left side of the editor with the commit number, date, and author. To show the annotations we go to *VCS → Git → Annotate* (number 5 on VCS Operations popup).

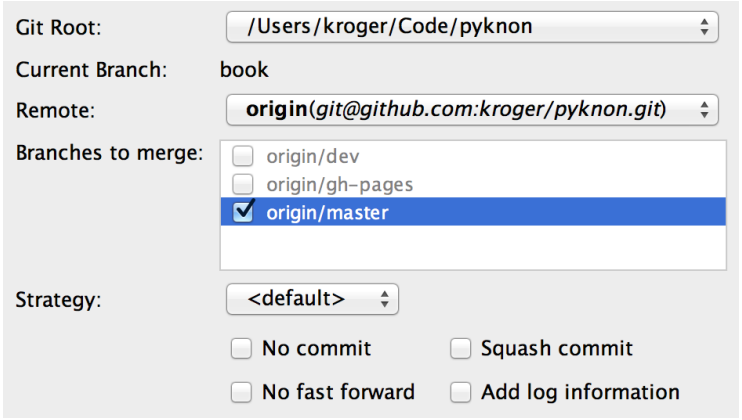


The only way to hide the annotations is to open a popup menu in the annotation area and click on "Close Annotations". The popup menu has a few other interesting actions, such as showing a diff and copying the revision number.

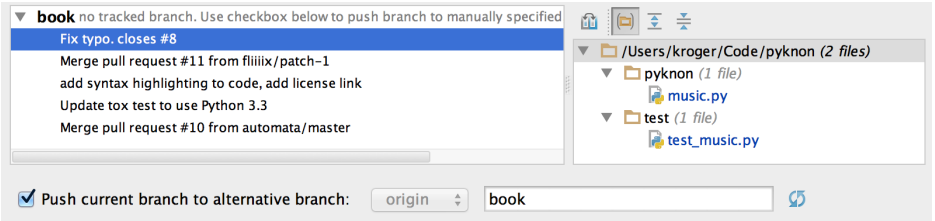


Pull and Push

We can pull changes from branches by going to *VCS → Git → Pull…*. PyCharm lists the available branches and allows some options and merge strategies: resolve, recursive, octopus, ours, subtree. Check the Git documentation for more information about Git's merge strategies and the PyCharm documentation for more information about the pull changes dialog.



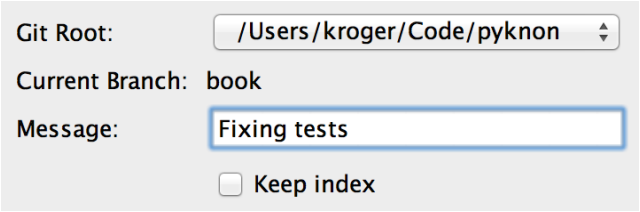
To push the changes to the remote server we go to *VCS → Git → Push… (⌘-S-k, S-C-k)* (number 8 on VCS Operations popup). The option "Push current branch to alternative branch" is useful to track the branch.



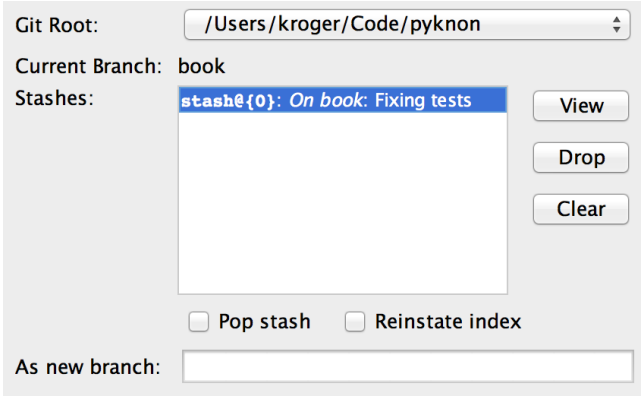
Finally, the option "Auto-update if push of the current branch was rejected" in *Settings* → *Version Control* → *Git* is useful to automatically synchronize the local branch if pushing to the server is rejected because we have an older version.

Stash

We can stash the latest changes easily with PyCharm by going to *VCS* → *Git* → *Stash Changes…* (number 9 on VCS Operations popup).



Applying the stash is equally easy by going to *VCS* → *Git* → *UnStash Changes…* (number 0 on VCS Operations popup).

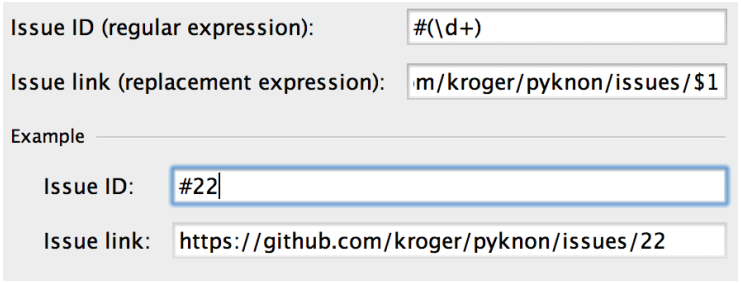


Issue Navigation

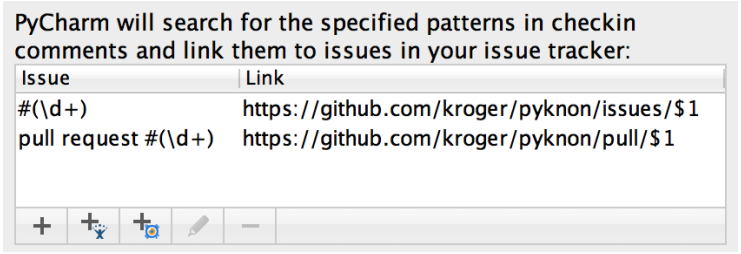
Often we use patterns in commit messages to refer to bug tracking tools, such as "closes `#11`," meaning that the current commit closes the bug number `11` in the bug tracking system. We can define these patterns in PyCharm (in *Settings* → *Version Control* → *Issue Navigation*), which will create a link to our bug tracking system.

PyCharm has out-of-the-box support for JIRA and YouTrack. In the following example, we will create patterns to link our log messages to GitHub Issues.

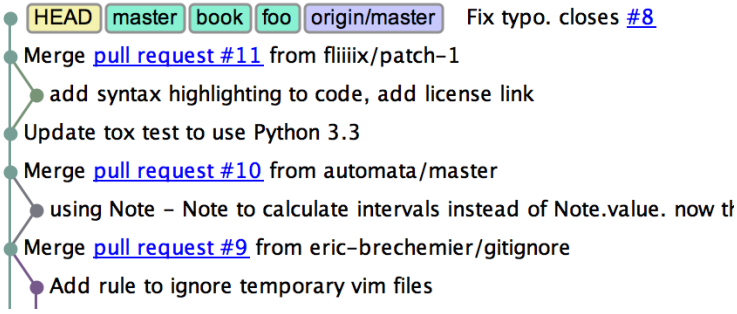
The simplest way is to detect the hash sign followed by a number. You may want to be more specific and include words like "closes" and "fixes" in the pattern, but I want to keep the example simple. We will use `#(\d+)` as the Issue ID and `https://github.com/kroger/pyknon/issues/$1` as the Issue link. The Example section in the Issue Navigation dialog is useful to test whether or not the regular expression is working correctly.



As we can see, I created two links, one for issues and another for pull requests:

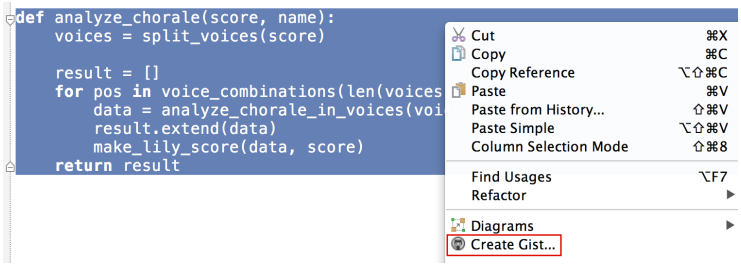


With that done, PyCharm will automatically create links in the Log window.

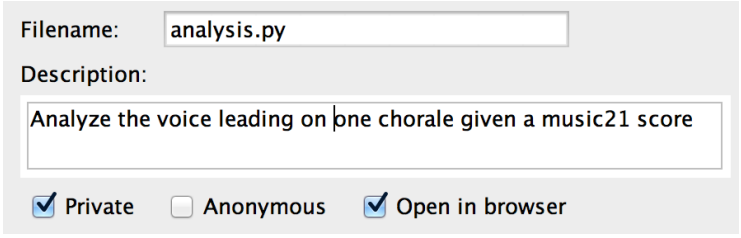


GitHub Gist

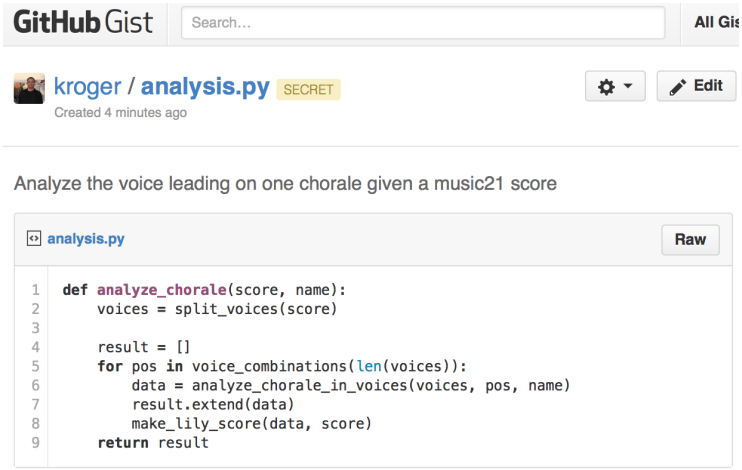
Sharing code using GitHub's Gist is very useful, and using PyCharm to create Gists is a walk in the park. We select the code we want to share, open the popup menu and click on *Create Gist…* (The following image was edited for simplicity.)



PyCharm will ask for the Gist filename and description; check the appropriate box to determine whether the Gist should be private or anonymous.



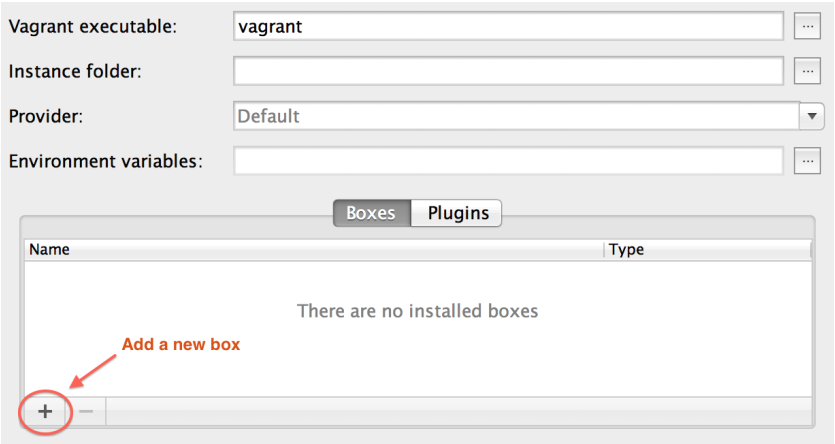
This is the result of our Gist:



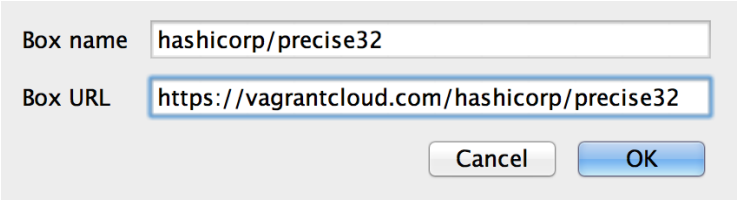
Vagrant

Vagrant is a great tool to create virtual development environments. It works with VirtualBox, AWS, VMware, Docker, and Hyper-V; with the addition of third-party plugins, it can also work with other providers such as Parallels. With Vagrant we can edit code locally and run it remotely in a virtual environment.

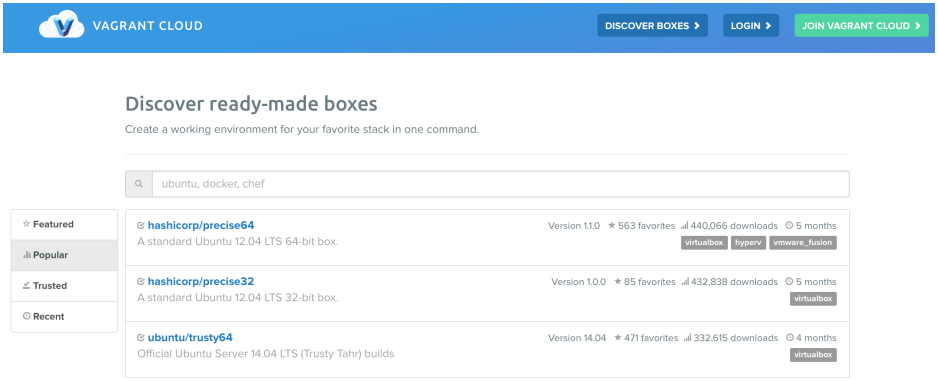
To start using Vagrant, we need to install a provider such as VirtualBox and set up a box. We can follow Vagrant's guide and do it on the command line; alternately, we can set up a new box without leaving PyCharm by going to *Settings* → *Tools* → *Vagrant* and clicking on the + button.



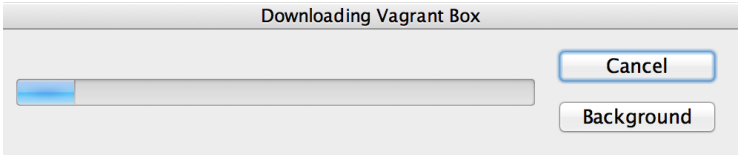
Next, we need to enter a box name and URL. PyCharm suggests the old Lucid Lynx (Ubuntu 10.04) by default, but in the following example I'll use the newer Precise Pangolin (Ubuntu 12.04).



The best way to get this information is by going to the Vagrant cloud website and picking the box you want.

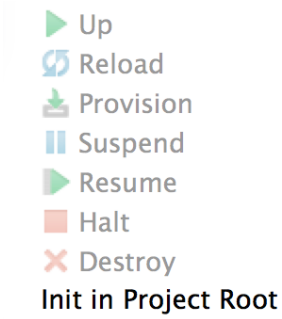


Downloading a box may take a while, so you may want to click on the Background button to close the popup box.

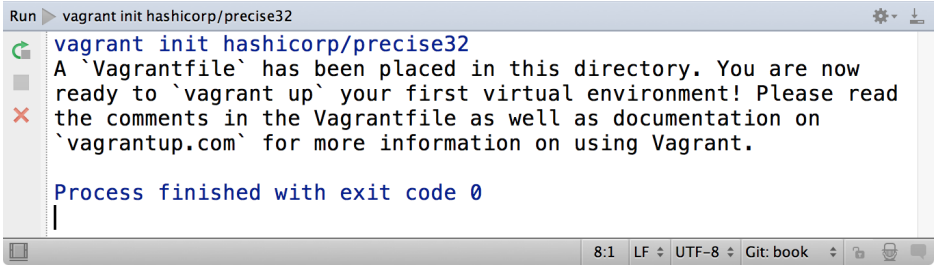


The box can now download in the background while we work on something else.

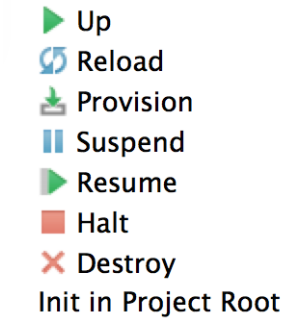
After downloading the Vagrant box, we need to initialize it by going to *Tools → Vagrant → Init in Project Root*. All other commands are disabled until we initialize the box.



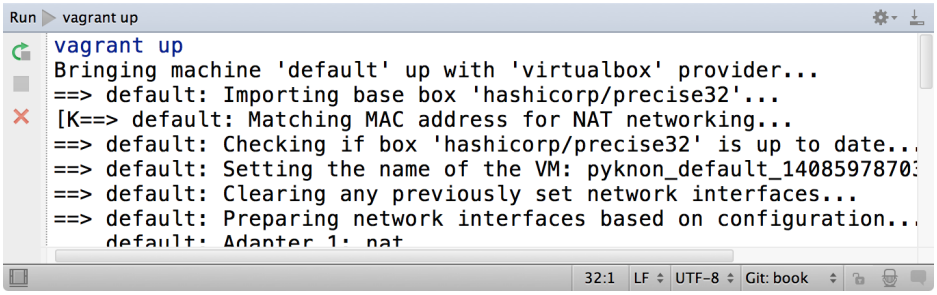
This command will run `vagrant init`, creating a configuration file named `Vagrantfile` in the project's root directory. This file is intended to be version controlled.



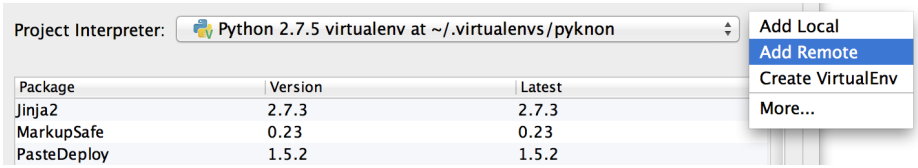
Now we can start the box by going to *Tools* → *Vagrant* → *Up*:



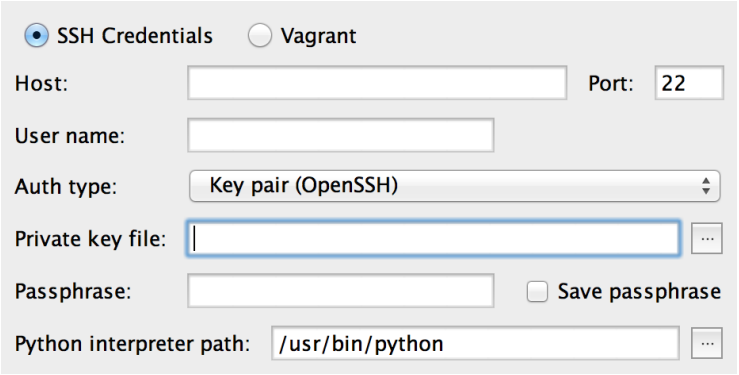
PyCharm will show a tool window with Vagrant's log. We can close this window (by clicking on the red X) without closing the Vagrant box.



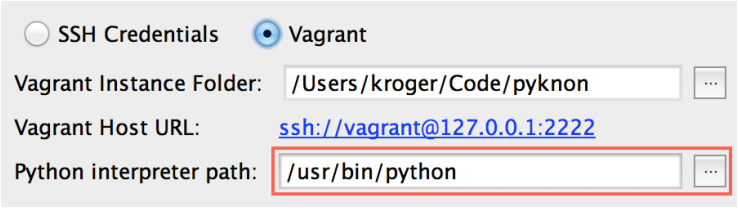
Now that our Vagrant box is up and running, we need to tell PyCharm to use the Python interpreter in the box. We go to *Settings* → *Project Settings* → *Project Interpreter* and select Add Remote after clicking in the settings icon on the right.



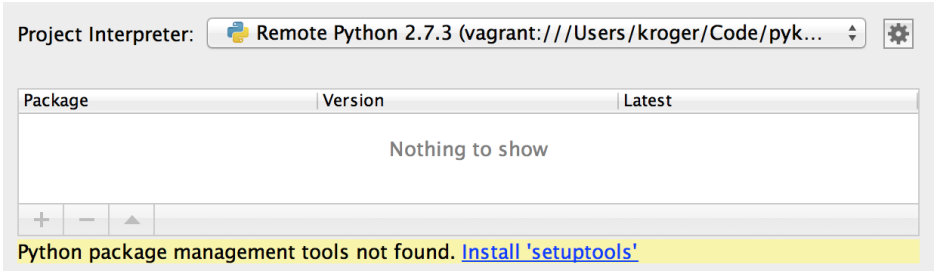
We can set up the SSH credentials ourselves (using keys for authorization is highly recommended):



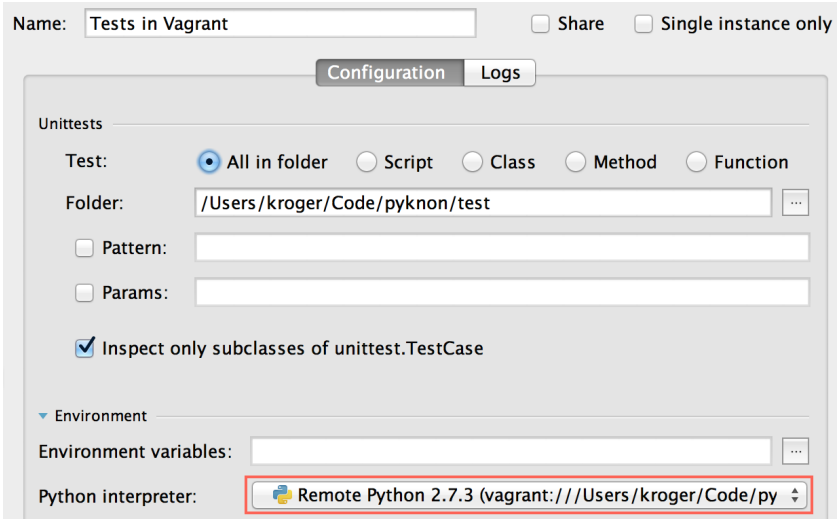
Or we can ask Vagrant to set up things automatically by clicking on the Vagrant option. In the following example I'm using the default Python interpreter in the box, which happens to be Python 2.7. Later we will see how to install and use an older Python version in the Vagrant box.



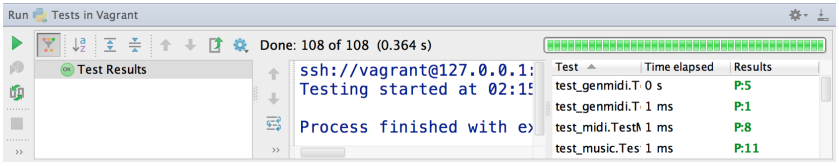
PyCharm may give an alert message that some packages such as `setuptools` and `pip` are not installed in the remote box. For now, just click on the link to install them; later we will see how to automate the installation of packages in a newly provisioned box.



Now that the remote interpreter is configured, we can use it. In the following example, I run my tests using the remote interpreter:



As we can see, all tests pass:



As mentioned previously, we can configure Vagrant to automatically install and configure software when we start a box (that is, we run `vagrant up`). This is called *Provision* in Vagrant parlance. We can use a configuration management utility such as Chef or Puppet, or simple shell scripts if our needs are modest. As mentioned in the Vagrant documentation:

> [Provisioning] is useful since boxes typically aren't built perfectly for your use case. Of course, if you want to just use vagrant ssh and install the software by hand, that works. But by using the provisioning systems built-in to Vagrant, it automates the process so that it is repeatable. Most importantly, it requires no human interaction, so you can vagrant destroy and vagrant up and have a fully ready-to-go work environment with a single command. Powerful.

Powerful indeed. We can use the `config.vm.provision` method call to make Vagrant call `install.sh` after setting up a box. In the following example we can see a basic `Vagrantfile` configuration:

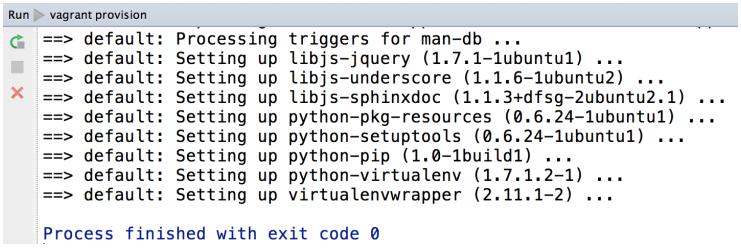
```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"
  config.vm.provision :shell, :path => "install.sh"
end
```

The `install.sh` script will install a few packages and Python 2.6.

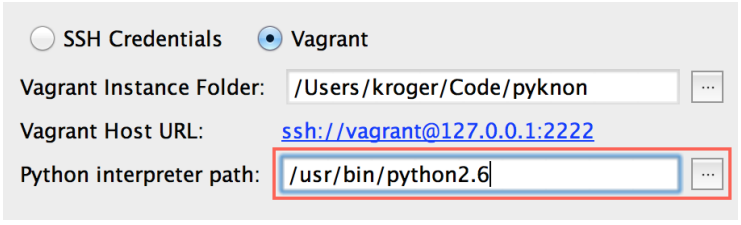
```
sudo apt-get update
sudo apt-get install -y python-software-properties
sudo apt-get update
sudo apt-add-repository ppa:fkrull/deadsnakes
sudo apt-get install -y python2.6
sudo apt-get install -y python-virtualenv virtualenvwrapper
```

After editing `Vagrantfile` we can re-provision our box by going to *Tools* →

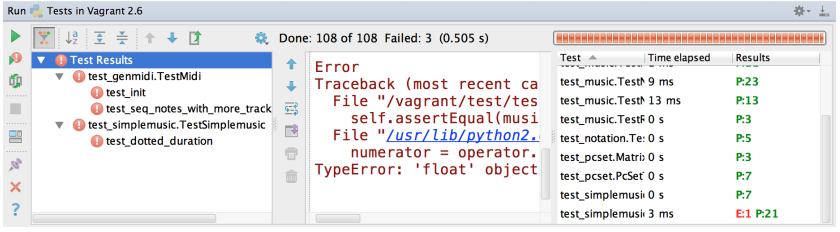
Vagrant → Provision.



Now I can configure a new remote Python interpreter, this time using Python 2.6:



And, as we can see, my tests fail when I run them in Python 2.6.

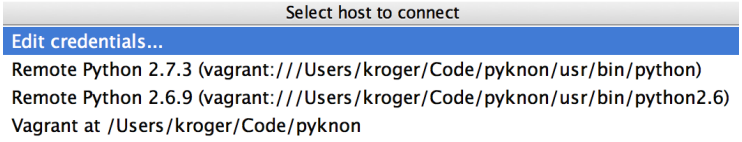


This example is just a demonstration. You should really use tox to test code in different Python versions.

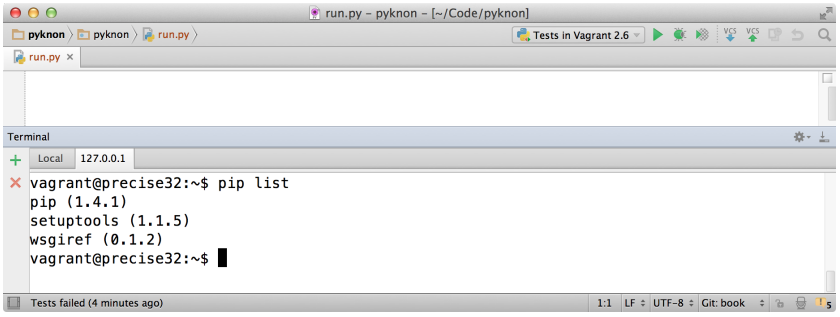
However, the main advantage of using Vagrant is to replicate a server or deployment machine without having to install the software in the

development machine, which can be hard if the server and development machines use different operating systems.

We can SSH to the Vagrant box by going to *Tools* → *Start SSH session...* and choosing the desired host.



We will have access to the machine without leaving PyCharm, and we can run the usual commands in the remote machine using PyCharm's terminal.

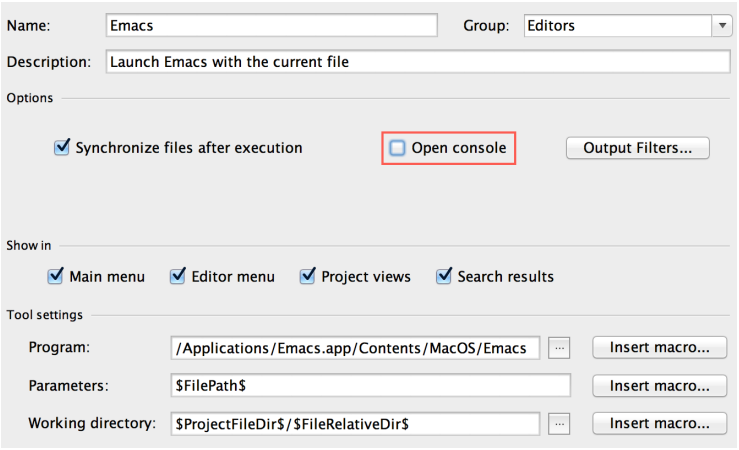


External Tools

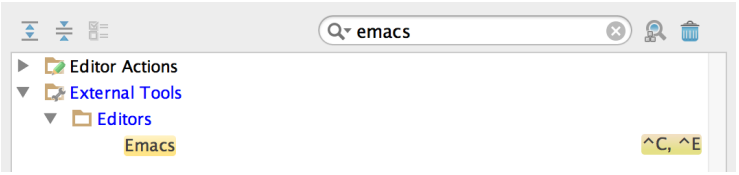
If you need some functionality that is not available in PyCharm, you can launch an external tool from PyCharm. In the following example I will configure PyCharm to open Emacs with the current file. Naturally, the same principle applies to other editors, such as Vim, Sublime, or TextMate.

To configure a new external tool, go to *Settings* → *Tools* → *External Tools* and click on the + button. In the Create tool window insert the tool name and

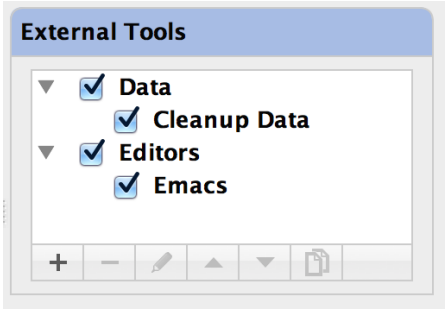
group (optional, but good for organization) and description. Unless you want to see a tool window when launching the tool, uncheck Open console. Insert the path to the program, the parameters (in this case we can get the full file name with `$FilePath$`), and the desired working directory; I like to use `$ProjectFileDir$/$FileRelativeDir$`.



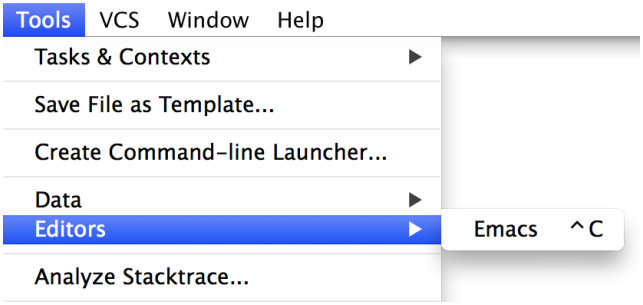
We can add a shortcut to the created tool as usual (see *Shortcuts*):



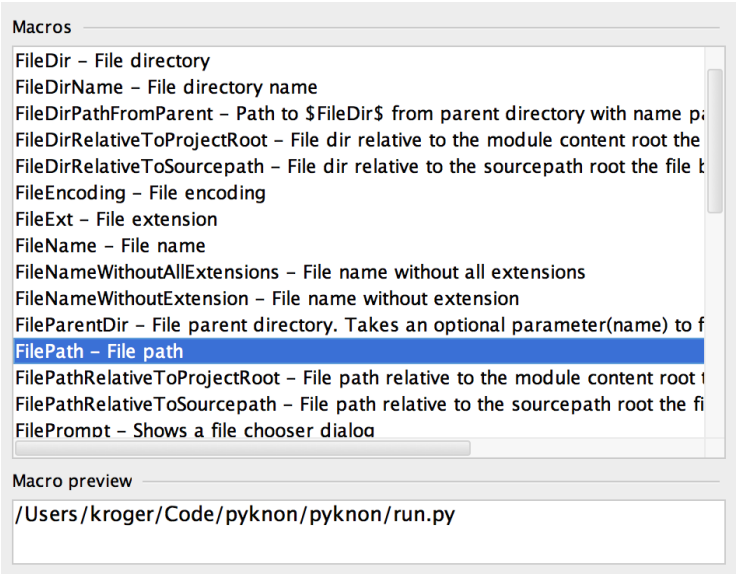
The external tool configurations will be organized by groups:



The group organization will also be reflected in the *Tools* menu, where we can access the external tools:

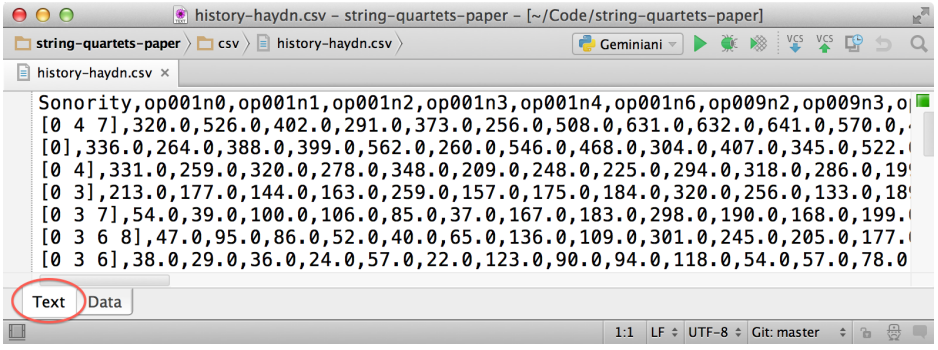


Finally, there are dozens of macros that can be used in the Program, Parameters, and Working Directory fields. These macros are not extensively documented, but most of them are self-explanatory and have a preview at the bottom.

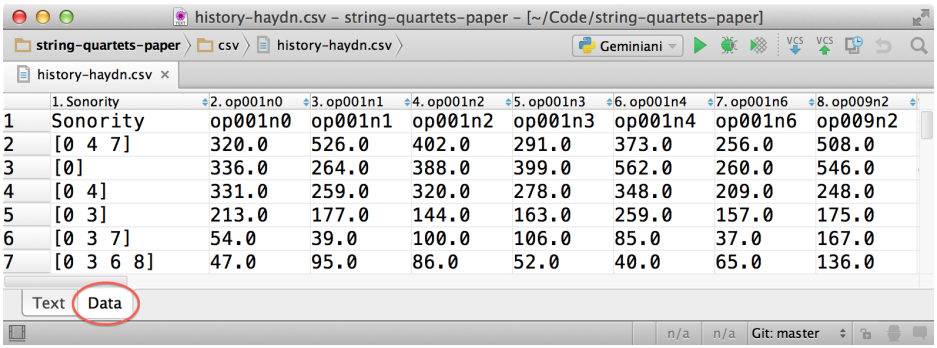


CSV files

It's a little-known fact that PyCharm can view and edit CSV files in a spreadsheet-like way. If we open a CSV file in PyCharm, it opens in Text mode by default, allowing us to edit it as we would in any other text editor.



When we switch to Data mode, PyCharm opens the CSV file as a spreadsheet. We can move a column by dragging and dropping, and add rows with *(⌘-n, A-Insert)*. We can sort a column by clicking on the header.

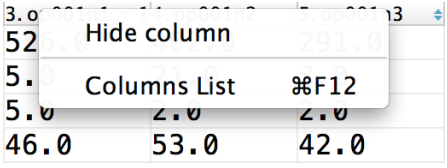


	1. Sonority	2. op001n0	3. op001n1	4. op001n2	5. op001n3	6. op001n4	7. op001n6	8. op009n2
1	Sonority	op001n0	op001n1	op001n2	op001n3	op001n4	op001n6	op009n2
2	[0 4 7]	320.0	526.0	402.0	291.0	373.0	256.0	508.0
3	[0]	336.0	264.0	388.0	399.0	562.0	260.0	546.0
4	[0 4]	331.0	259.0	320.0	278.0	348.0	209.0	248.0
5	[0 3]	213.0	177.0	144.0	163.0	259.0	157.0	175.0
6	[0 3 7]	54.0	39.0	100.0	106.0	85.0	37.0	167.0
7	[0 3 6 8]	47.0	95.0	86.0	52.0	40.0	65.0	136.0

We can edit cells by double-clicking on them:

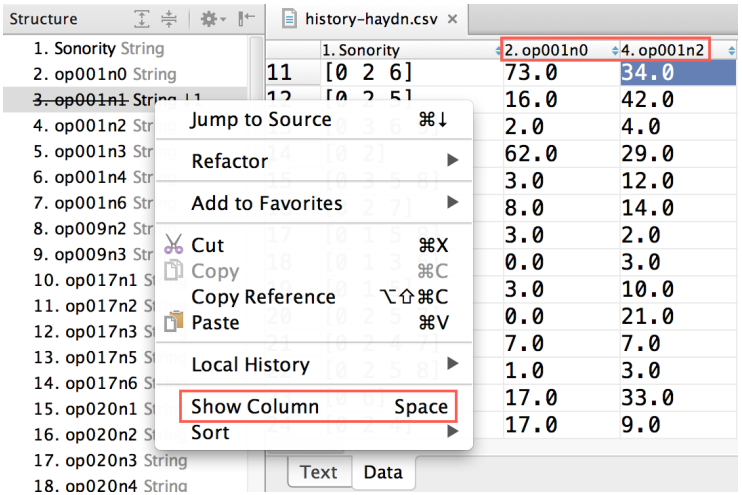
1. Sonority	2. op001n0	3. op001n1	4. op001n2
Sonority	op001n0	op001n1	op001n2
[0 4 7]	320.0	526.0	402.0
[0]	336.0	264.0	388.0
[0 4]	331.0	259.0	320.0
[0 3]	213.0	174.0	144.0
[0 3 7]	54.0	39.0	100.0

To hide a column, right-click on the column and select Hide column on the popup menu.

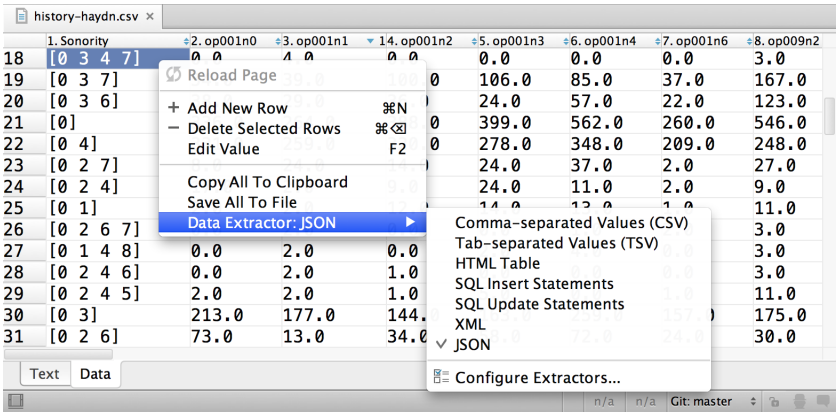


The Structure window in *View → Tools Window → Structure (⌘-7, A-7)*

shows the document's columns, including the hidden ones. By right-clicking on a column in the Structure window, we have access to many options, such as Show Column and Hide Column (we can toggle both with the spacebar).

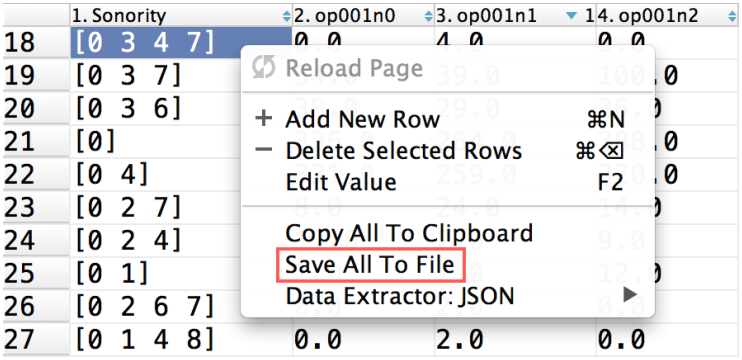


In PyCharm, the output formats are called data extractors. In the popup menu we can select a data extractor such as CSV or JSON.



After selecting the data extractor, we need to go back to the popup menu and

select *Save All To File*:



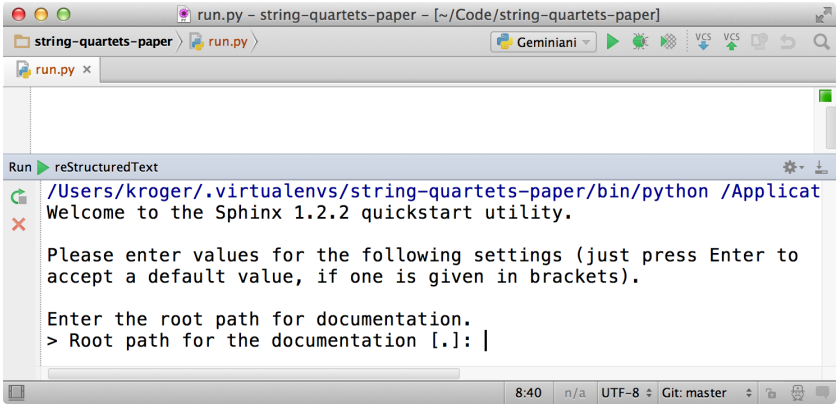
It would be unreasonable to expect PyCharm to have all the features of a full-featured spreadsheet, but it's very useful to be able to do quick edits to CSV documents without having to import and export these files into a spreadsheet.

Writing Documentation with Sphinx

As you know, Sphinx is Python's documentation tool of choice and, as expected, PyCharm has support to make it easy to use Sphinx to write documentation.

Configuring Sphinx

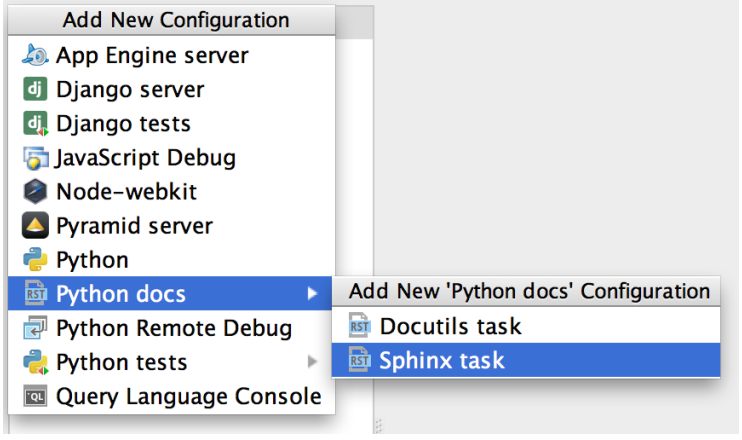
To get started we need to install Sphinx and run the `sphinx-quickstart` script. We can run it in the command line or from PyCharm in *Tools → Sphinx quickstart*. A guide of how to set up Sphinx is beyond the scope of this book, but you can easily find out how to do so in the Sphinx documentation.



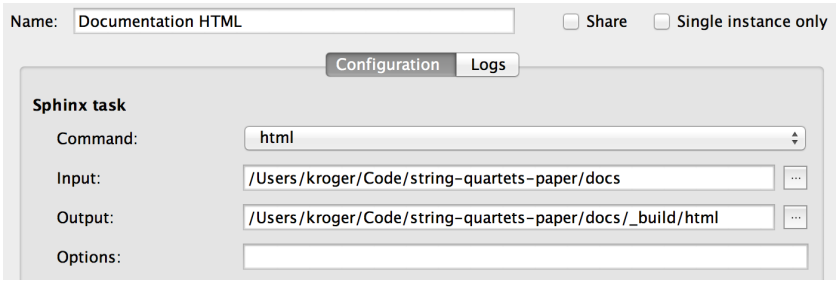
If you want to use `autodoc` and `doctest` with Sphinx, select Yes when asked by `sphinx-quickstart`.

```
Please indicate if you want to use one of the following Sphinx extensions:
> autodoc: automatically insert docstrings from modules (y/n) [n]: y
> doctest: automatically test code snippets in doctest blocks (y/n) [n]: y
```

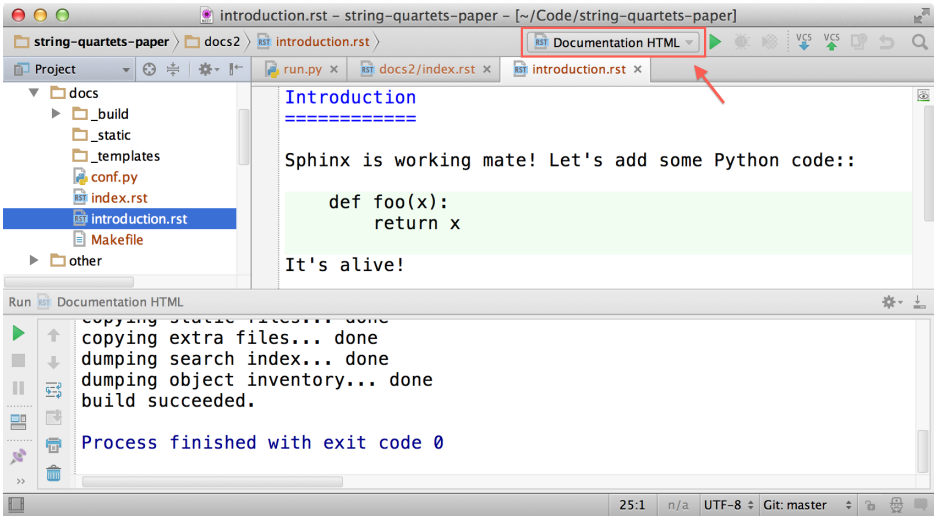
After the initial configuration, we need to create a new run configuration in PyCharm to compile the documentation (see *Running Code*). When adding a new documentation, select *Python docs → Sphinx task*.



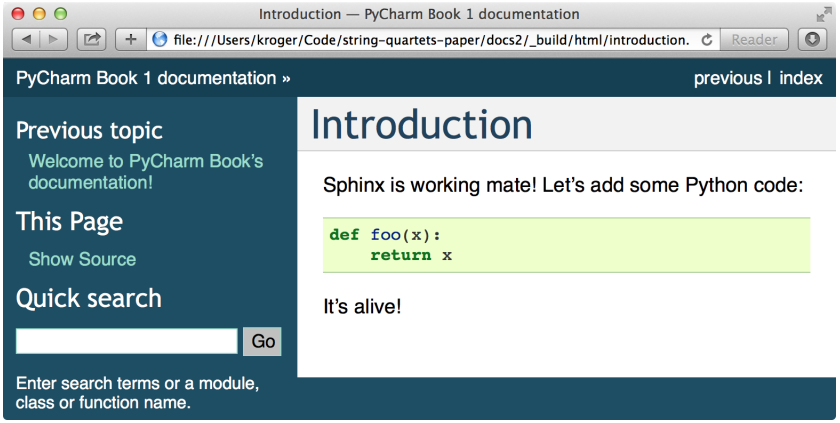
Sphinx can generate many output formats, such as html, pdf, and man pages. Select the format you want in Command and add the directories for the input and output.



With that, we can generate the documentation by using the standard *Run* commands.



And, as expected, we can open the documentation in the browser.



The process of editing the documentation, running Sphinx, and reloading the browser gets old pretty quickly. Fortunately, there's a better way. In *File Watchers* we will see how to generate the documentation and reload the browser automatically whenever the source is changed.

PyCharm has some support to write ReStructuredText files, especially syntax highlighting. Sadly, it doesn't have shortcuts to add sections, format text, or deal automatically with labels and cross-references.

```
Introduction
============

Sphinx is working mate! Let's add some Python code::

    def foo(x):
        return x

Let's add some ``code``, with *emphasis*, and **bold**!
```

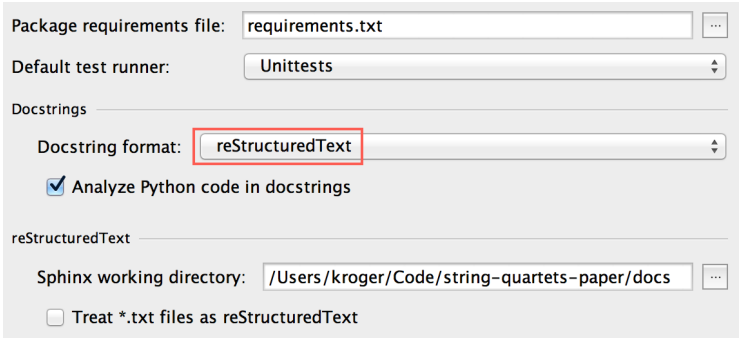
Document Code

An effective way to write documentation with Sphinx is to write a part of the documentation using Sphinx and another part as docstrings in the Python

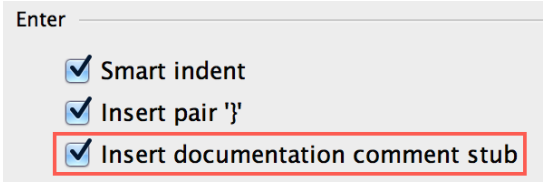
code. We can use directives such as autofunction to bring them together.

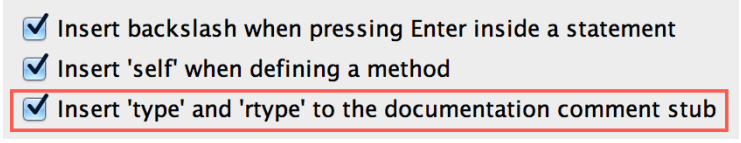
Another advantage of having the documentation as docstrings is that PyCharm can extract and show this information. PyCharm works with docstrings in both reStructuredText and Epytext formats. If we are using Sphinx, reStructuredText is the format we want to use.

We can choose the docstring format in *Settings → Tools → Python Integrated Tools*.



You may want to enable the options "Insert documentation comment stub" and "Insert 'type' and 'rtype' to the documentation comment stub" in *Settings → Editor → General → Smart Keys*.





With these options enabled, we can follow the ensuing steps to add documentation to our code.

- Move the caret to the right after the function declaration.

```
def inversion(notes, index=0):
    |
    return [mod12(index - n) for n in notes]
```

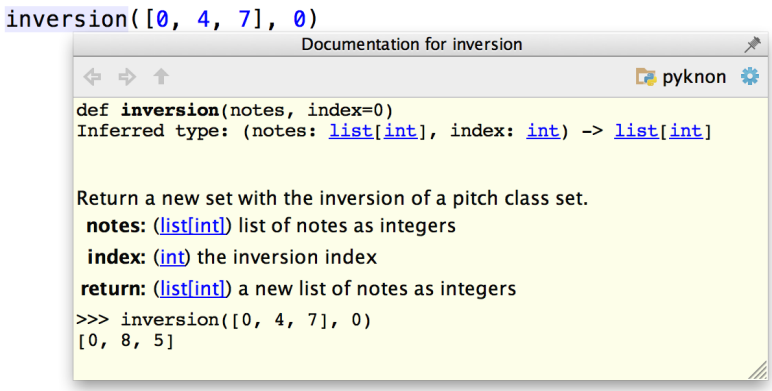
- Type the opening triple quote followed by the *Enter* key. PyCharm will add the documentation stub.

```
def inversion(notes, index=0):
    """

    :param notes:
    :type notes:
    :param index:
    :type index:
    :return:
    :rtype:
    """
    return [mod12(index - n) for n in notes]
```

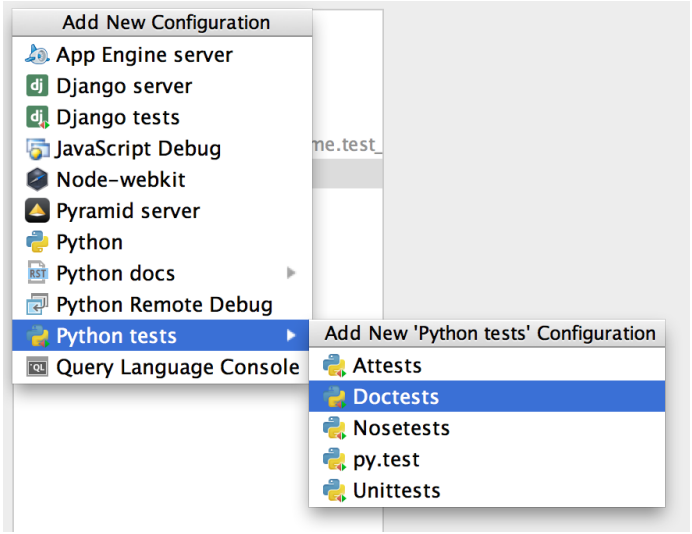
- Complete the description text (`:param`) and type (`:type`) for the arguments. In the following example I also add a doctest. It will serve both as an example and as a test that can be checked automatically.

```
def inversion(notes, index=0):
    """
    Return a new set with the inversion of a pitch class set.

    :param notes: list of notes as integers
    :type notes: list[int]
    :param index: the inversion index
    :type index: int
    :return: a new list of notes as integers
    :rtype: list[int]

    >>> inversion([0, 4, 7], 0)
    [0, 8, 5]
    """

    return [mod12(index - n) for n in notes]
```

And, as we have seen in *Documentation*, we can see the documentation for a user-defined symbol inside PyCharm.



Checking our doctests is as simple as creating a new test configuration (see *Testing Code*) and selecting "Doctests" as a configuration template:

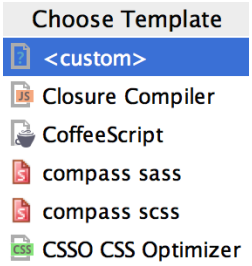


One possible downside to writing long docstrings is that it takes valuable screen space. As we have seen in *Code Folding and Regions*, we can not only fold the docstrings, but we can also configure PyCharm to fold all comments by default.

```
def inversion(notes, index=0):
    """Return a new set with the inversion of a pitch class set...."""

    return [mod12(index - n) for n in notes]
```

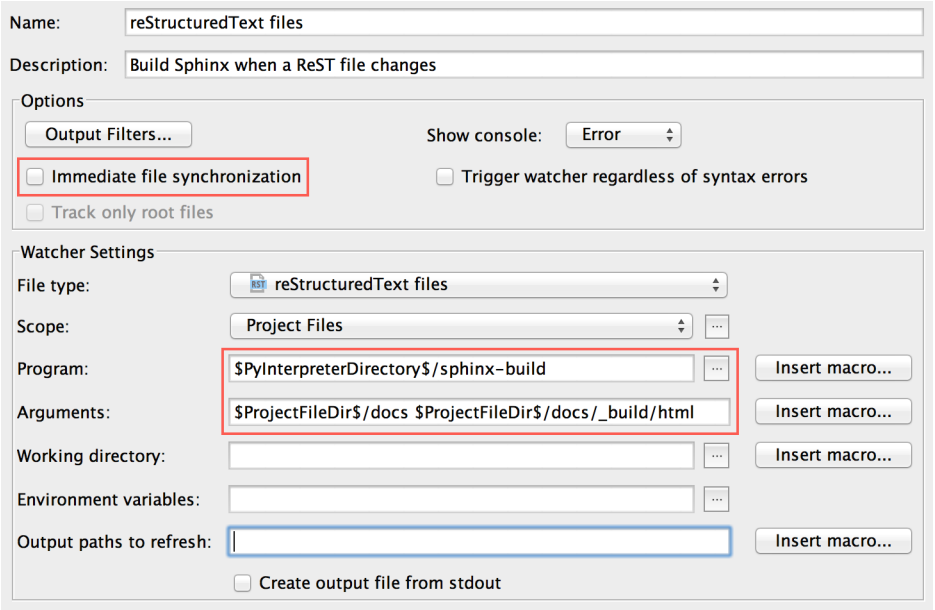
File Watchers

File watchers are useful when we need to generate files from one format to the other, such as CoffeeScript to JavaScript, or HAML to HTML. We can create new file watchers by going to *Settings* → *Tools* → *File Watchers* and clicking on the + button. PyCharm has many templates available and we can create our own from scratch.

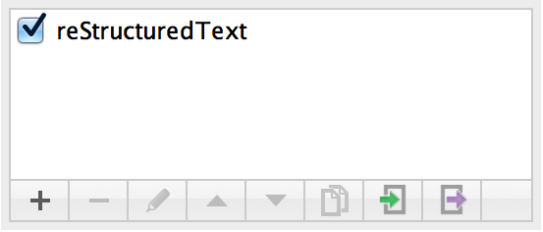


In the following example I'm creating a file watcher to generate the Sphinx HTML documentation when the reST file is saved. Be sure to uncheck "Immediate file synchronization" or it will run Sphinx after each keystroke. Having it unchecked will run Sphinx only when the file is saved or when we move focus from PyCharm.

The macro `$PyInterpreterDirectory$` will return the path of the current Python interpreter, in my case `~/.virtualenvs/<virtualenv name>/bin`. We need to pass two arguments to `sphinx-build`: the source (`$ProjectFileDir$/docs2`) and output (`$ProjectFileDir$/docs2/_build/html`) directories. Check the sphinx-build documentation for more details.



When we click on the OK button, the file watcher is created and begins to run. We can disable it by clicking on the check button.

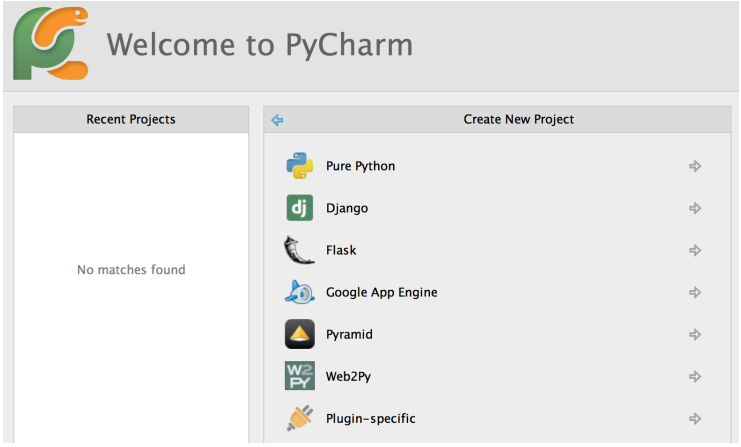


Now, the documentation will be generated every time we save a documentation file. The Firefox extension Firefox Auto Reload will reload a page automatically when it changes (that is, when the documentation is regenerated). For the other browsers, you may want to try Tincr and Live Reload.

6 | Web Programming

Introduction

PyCharm is a great IDE for web development, so much so that I use it to edit my static websites (that is, they don't use a framework like Django or Flask). PyCharm has great support for HTML, CSS, JavaScript, and other web-related technologies, and supports Django, Flask, Google App Engine, Pyramid, and Web2Py. We can choose one of these frameworks when creating a new project in the welcome window.

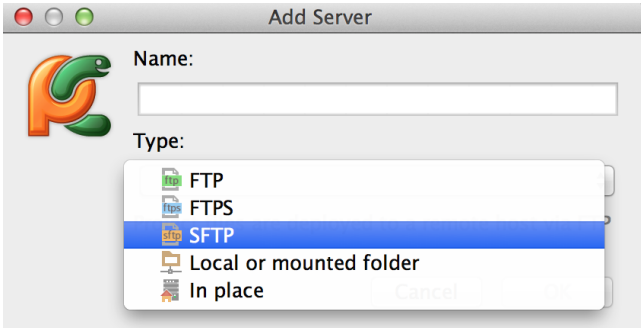


In this chapter, we will see an overview of PyCharm's capabilities for web programming. My goal is to show the tools to get started. (Each topic could fill a whole book!) In this chapter, we will see how to deploy code, work with databases, write HTML and CSS, work with JavaScript, write a static web page with Jinja, and get started with Django, Pyramid, and Flask.

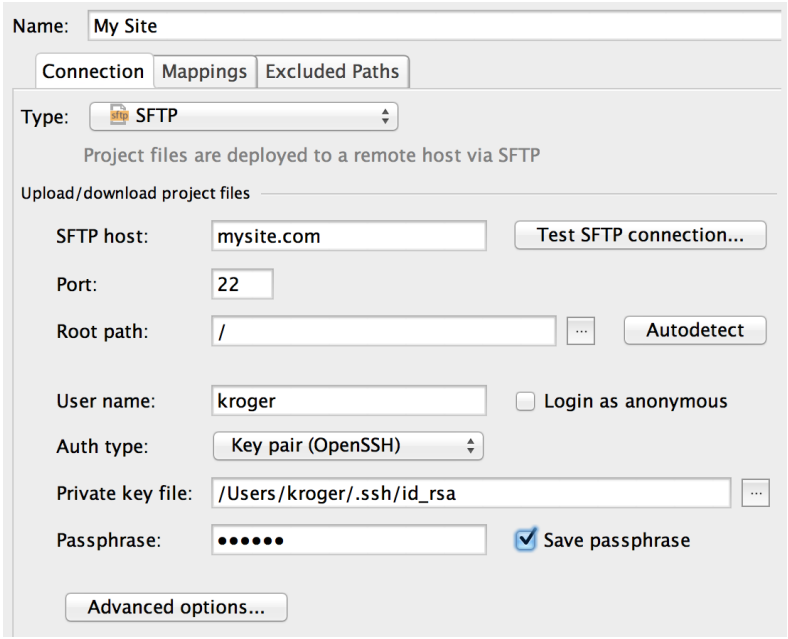
Deployment

PyCharm has a simple functionality to deploy code and files, although I prefer to use Fabric and a version control system for simple deployments, and Chef or Puppet for complex ones.

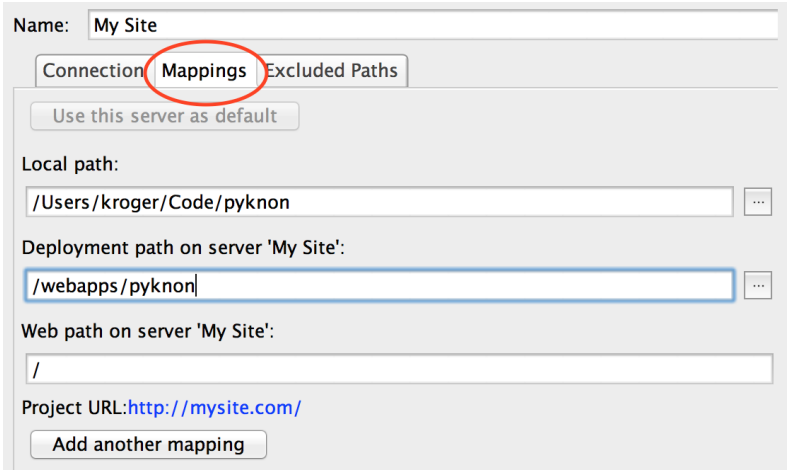
To deploy code with PyCharm, we need to add a web server by going to *Settings* → *Build, Execution, Deployment* → *Deployment*, clicking on the + button, and entering a server name and type.



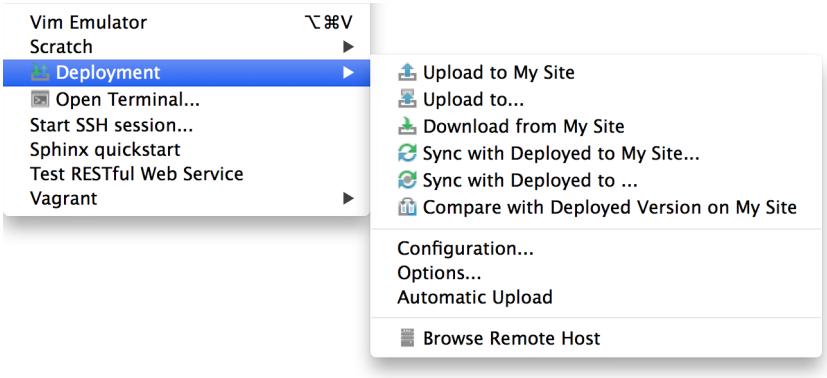
Next, we need to enter the authorization information, such as user name and password (or SSH key file).



In the Mappings tab, we can specify where the local code is and where it should be copied remotely.

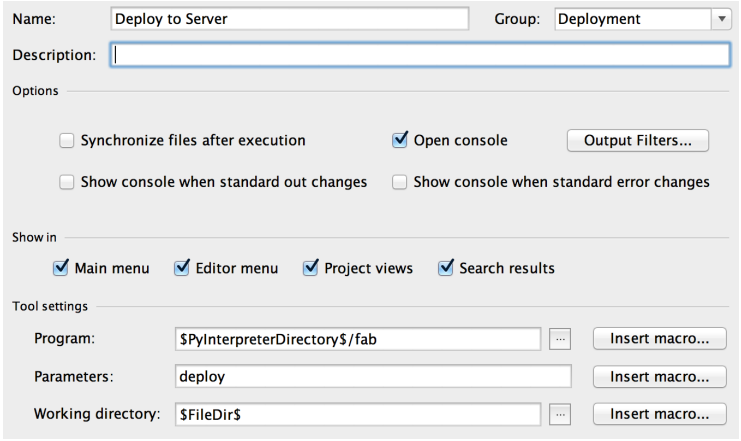


The code can be deployed by going to *Tools → Deployment* and choosing the desired option. Deployment in PyCharm is very granular; we can deploy one single file or the whole source code. Also, PyCharm has actions to compare remote and local versions. Although this is nice, I think it's more reliable to use automatic deployments and a version control system to compare local and remote versions.

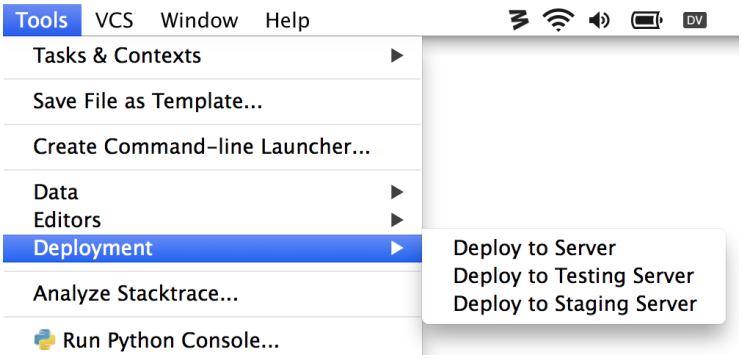


External Tools for Deployment

Another way to deploy code is to use an external tool, such as Fabric, and run it from PyCharm by creating an external tool in *Settings → Tools → External Tools* (see *External Tools*). In the following example, I create an external tool in PyCharm to run the command `fab deploy`.



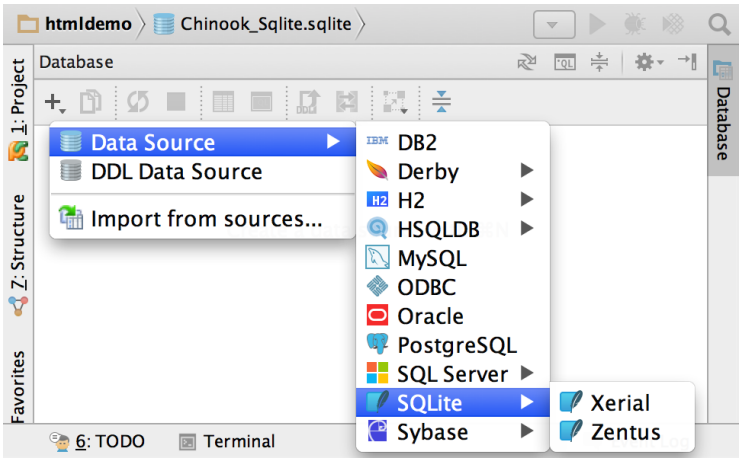
After creating a external tool, it'll be available in the menu in *Tools → Deployment*.



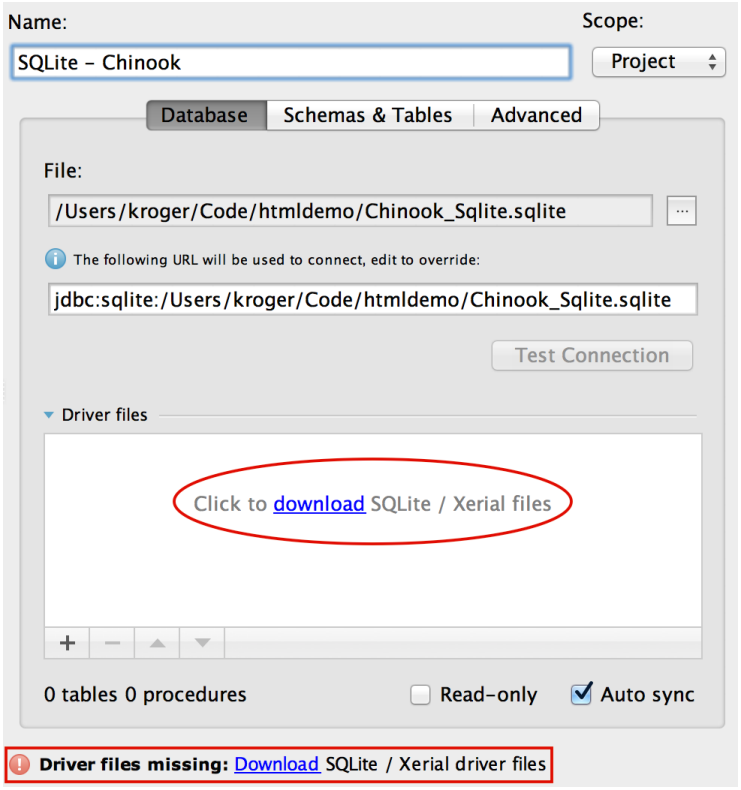
Databases

When working with a database, it's often useful to be able to see and modify data visually. Luckily, PyCharm has a database editor built-in. The examples in this chapter use the fantastic Chinook sample database.

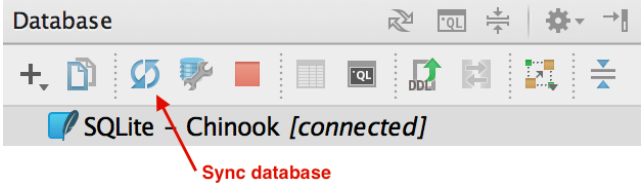
First, we need to configure a data source by going to *View* → *Tool Windows* → *Database*, clicking on the + button in the tool window, and selecting the database type. In this chapter, I will use SQLite as a data source, but the process is similar for other databases.



After selecting the data source, we need to configure the database host and port (or file name, if the database is SQLite). If this is the first time we are configuring a database, we may need to download a specific driver. Just click on the "download" link, and PyCharm will take care of the rest.

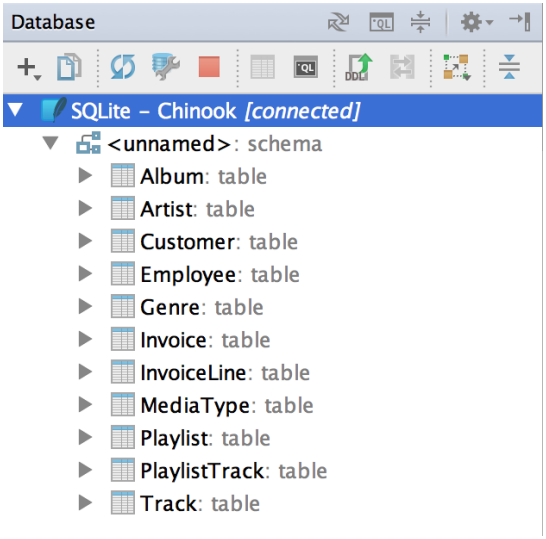


After configuring a database, we may need to synchronize it by clicking on the third button in the database toolbar.

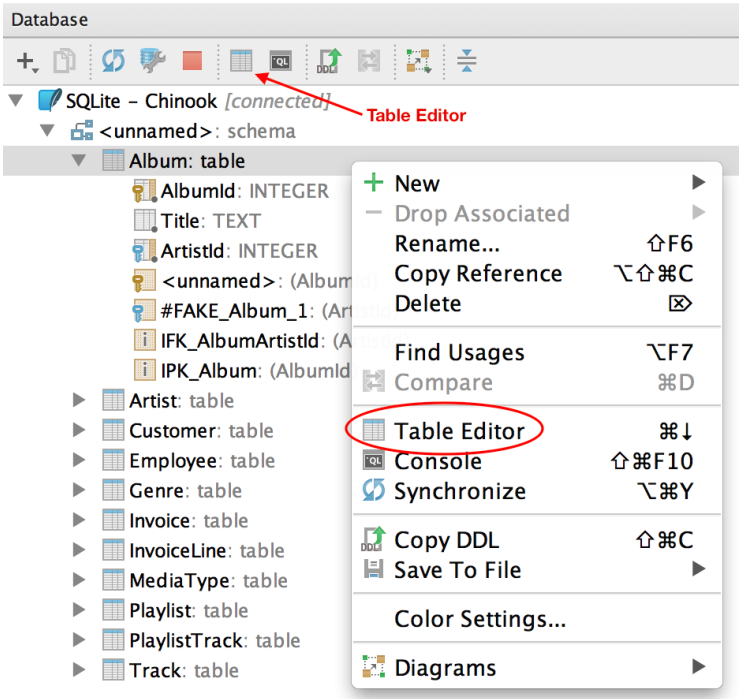


After synchronizing, all tables in the database will be available from PyCharm. In the following figure, we can see all of the tables in the Chinook

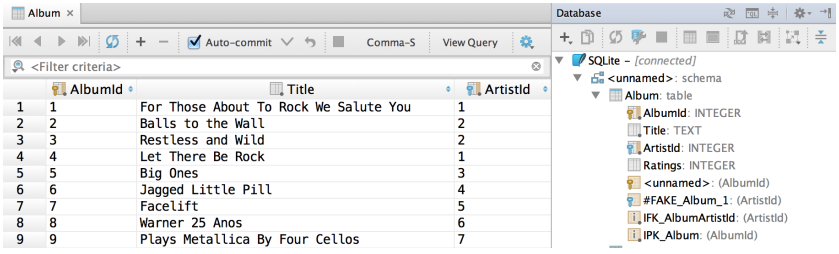
database:



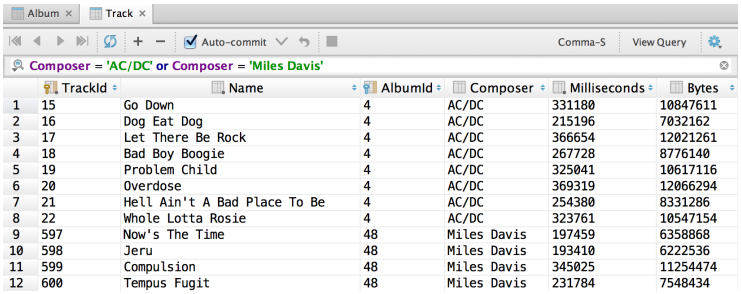
Many useful actions, such as creating and renaming tables, are accessible in a popup menu. The most useful is probably the table editor, which can be opened by clicking on the icon in the toolbar or by clicking in "Table Editor" in the popup menu.



The Table Editor shows the data in a spreadsheet-like way, as expected:

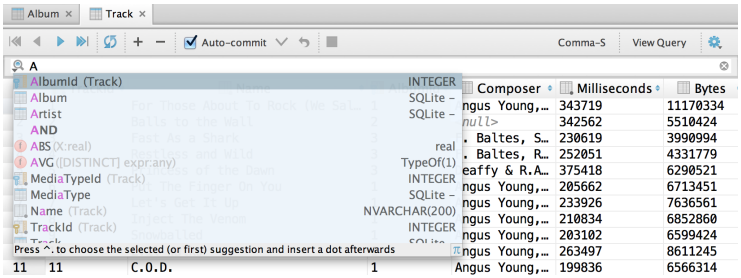


We can inspect data quickly and effectively by using the Row Filter:

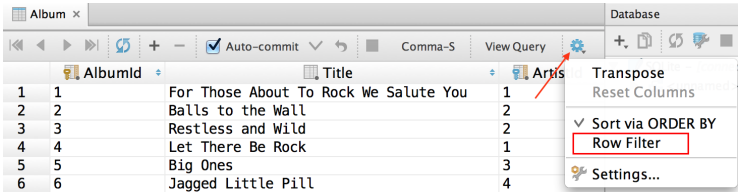


	TrackId	Name	AlbumId	Composer	Milliseconds	Bytes
1	15	Go Down	4	AC/DC	331180	10847611
2	16	Dog Eat Dog	4	AC/DC	215196	7032162
3	17	Let There Be Rock	4	AC/DC	366654	12021261
4	18	Bad Boy Boogie	4	AC/DC	267728	8776140
5	19	Problem Child	4	AC/DC	325041	10617116
6	20	Overdose	4	AC/DC	369319	12066294
7	21	Hell Ain't A Bad Place To Be	4	AC/DC	254380	8331286
8	22	Whole Lotta Rosie	4	AC/DC	323761	10547154
9	597	Now's The Time	48	Miles Davis	197459	6358868
10	598	Jeru	48	Miles Davis	193410	6222536
11	599	Compulsion	48	Miles Davis	345025	11254474
12	600	Tempus Fugit	48	Miles Davis	231784	7548434

The Row Filter has completion:

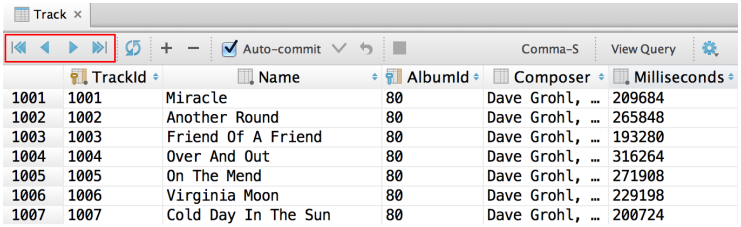


The Row Filter can be toggled by clicking on the settings icon on the right:

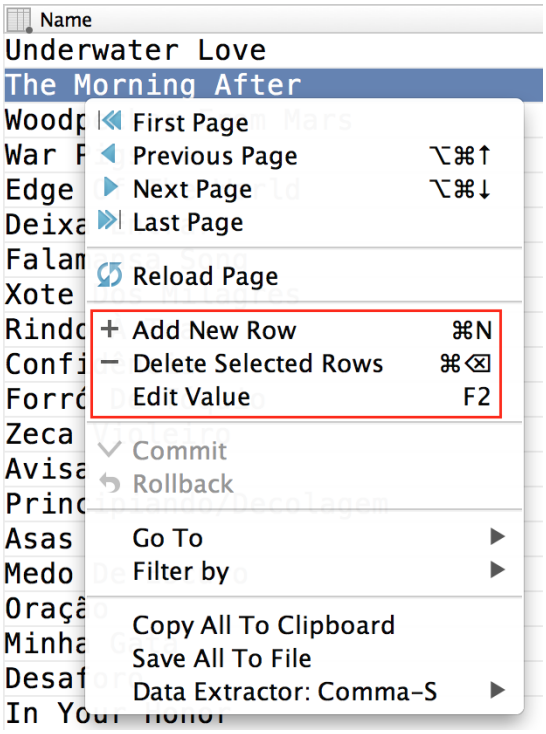


	AlbumId	Title	Artis
1	1	For Those About To Rock We Salute You	1
2	2	Balls to the Wall	2
3	3	Restless and Wild	2
4	4	Let There Be Rock	1
5	5	Big Ones	3
6	6	Jagged Little Pill	4

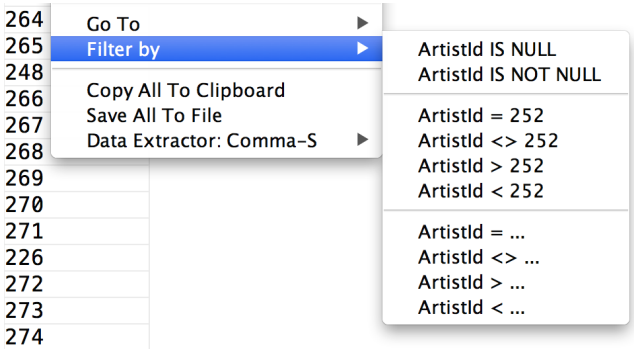
Unlike a spreadsheet that shows the data continuously, the result in the Table Editor is paginated. We can use the icons on the left for navigation. The default is to show 500 items per page. This can be changed in *Settings* → *Tools* → *Database*.



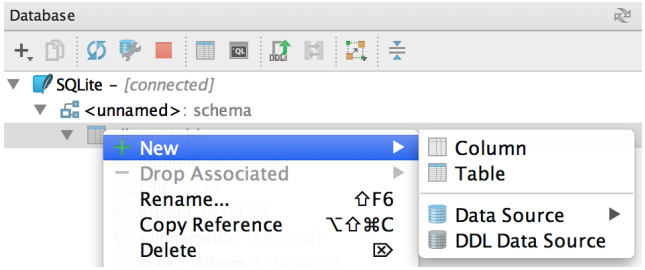
We can perform actions in a table row, such as adding or deleting, by accessing a popup menu in the row.



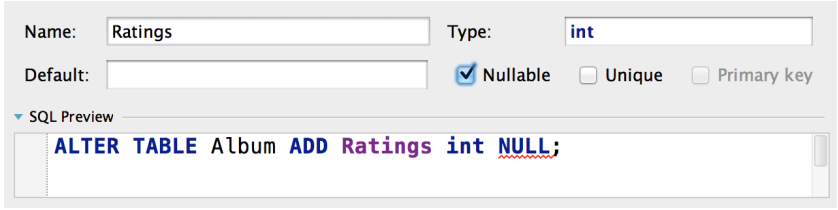
The *Filter by* action in the popup menu is useful for filtering data quickly by using some predetermined filters.



To create a new column, we click on the + button or open a popup menu on a table and select *New → Column*.



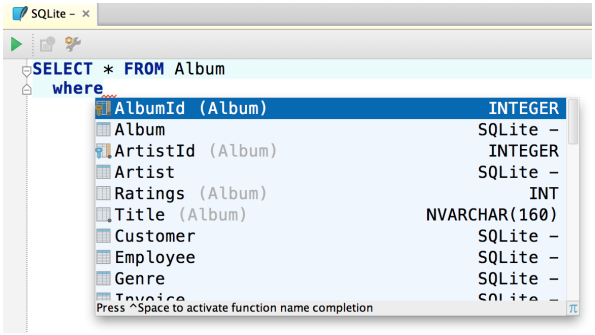
When adding a new column, we can add its name and type, a default value, and whether it can be null or unique:



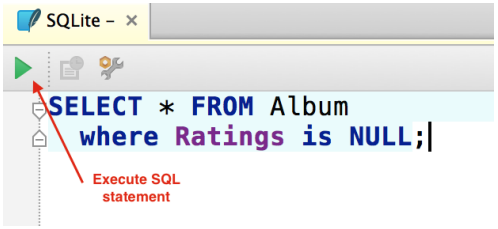
Composing SQL

To run SQL commands, we need to open the Database Console by clicking on the icon in the database toolbar *(⌘-S-F10, S-C-F10)*.

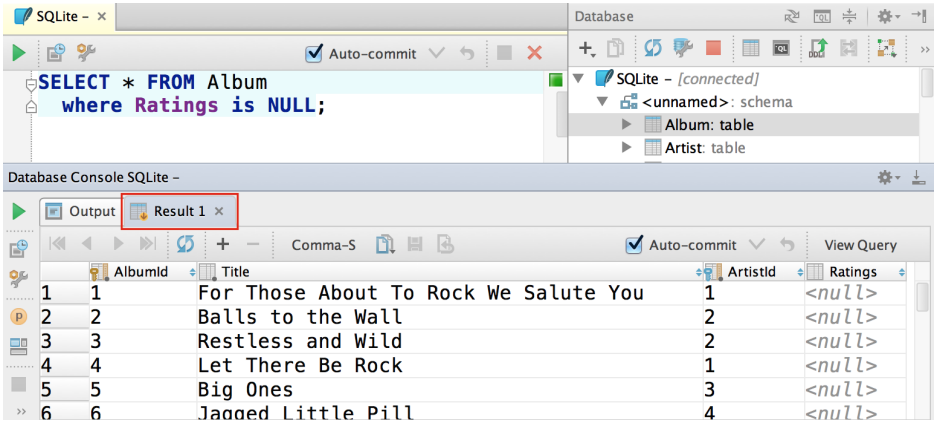
The Database Console behaves like an editor window with completion.



To execute an SQL statement, place the caret somewhere in the statement and click on the first icon in the toolbar (⌘-Enter, C-Enter). To execute more than one statement, select the statements, open a popup menu, and click on "Execute Selection."

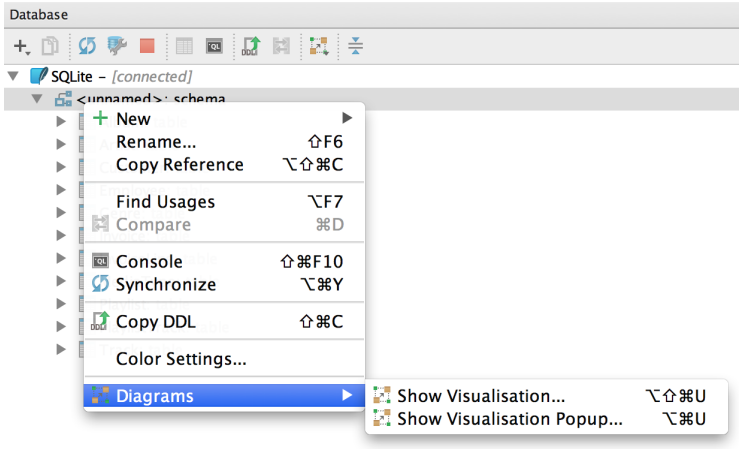


We can access the results in the Result tab in the Database Console tool window.

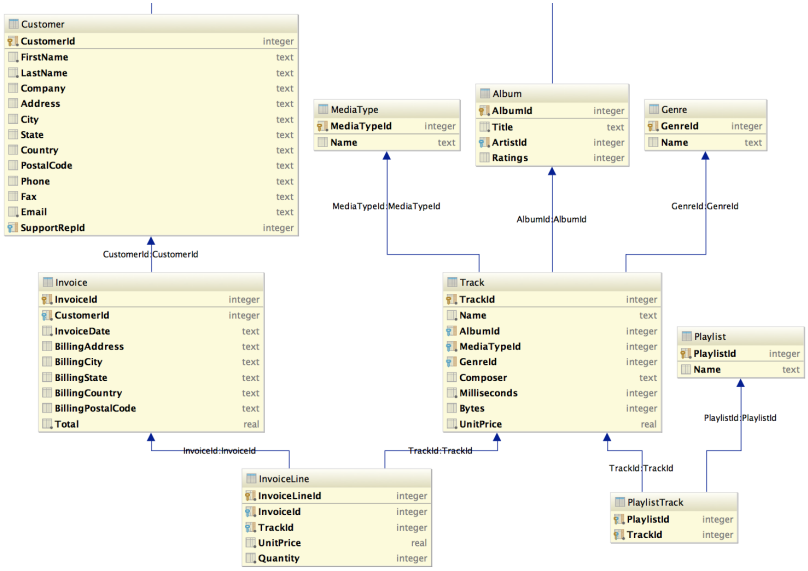


Diagrams

PyCharm can display the structure of a database as a diagram that show us the relationship between tables. This can be especially useful when working with a new and unfamiliar code base. Select the database or tables to visualize, open the popup menu, and go to *Diagrams → Show Visualization…*.



In the following image, we can see the structure of the Chinook database:



HTML and CSS

PyCharm has such great support for HTML and CSS that I use it even for my non-Python projects. As expected, code folding (see *Code Folding and Regions*) works as expected and is super useful in a highly nested format such as HTML.

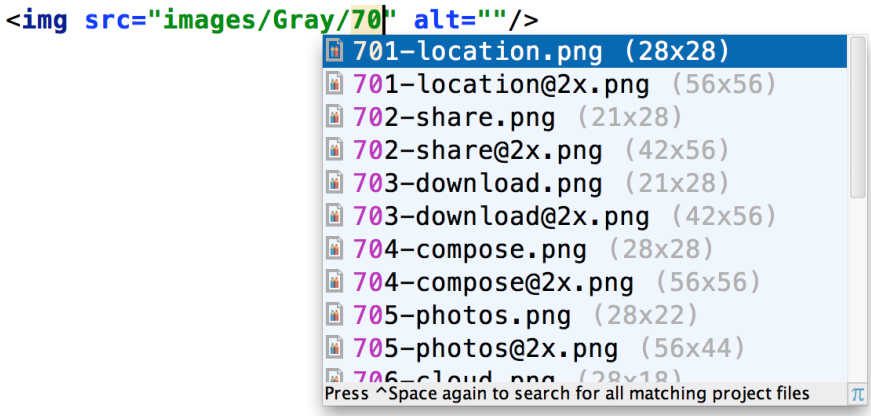
Completion

Completion works as expected and even tries to be useful by adding attributes when necessary or required:

We can add or change how the completion works in *Settings → Editor → Live Templates* (see *Live Templates*).

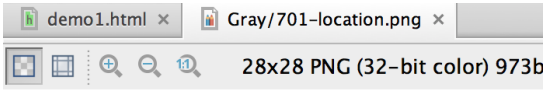
PyCharm will list files in the completion, making it easy to add style sheets.

And it works with images, as well:



Preview

To preview an image, we can use *View → Jump to Source (⌘-↓, F4)*.

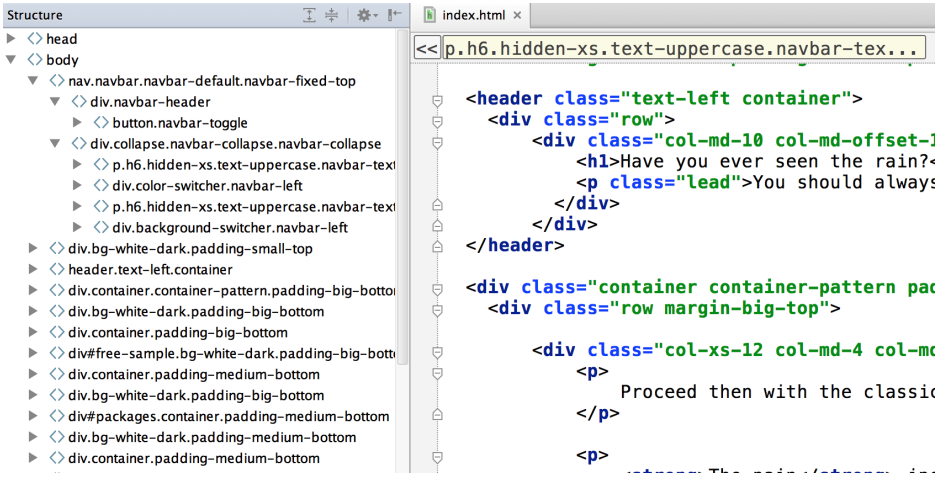


To preview the whole document in a browser, we go to *View → Open in Browser* and choose a browser or click in the small toolbar on the right.

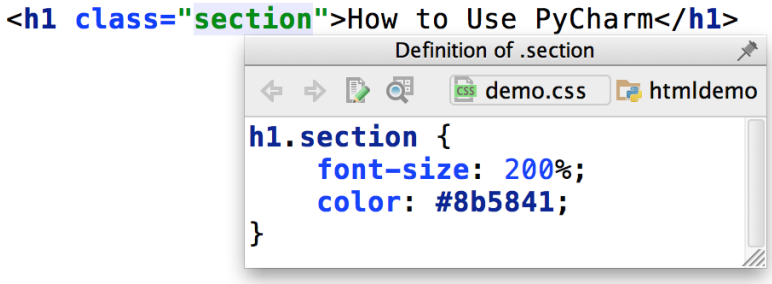
```
<!DOCTYPE html>

<html>

<head>
 <link rel="stylesheet" href="demo.css"/>
</head>
<body...>
</html>
```

Structure

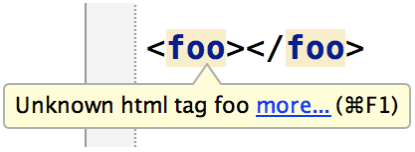
The Structure tool window in *View → Tool Windows → Structure (⌘-7, A-7)* works for all types of file PyCharm handles, but I find it particularly useful to give an overview of a deeply indented HTML file:



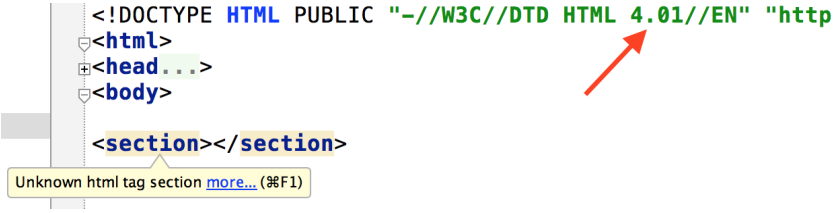
And, as we have seen in *Documentation*, we can see the documentation for a user-defined symbol inside PyCharm in *View → Quick Definition (⌥-Space, S-C-i)*. In an HTML document, it will show how an element is defined:



PyCharm knows about HTML, so if we use a wrong tag or attribute it will let us know:



PyCharm uses the doctype defined in the document (or the default one, if none is declared in the document). In the following example, we get an error message because we are using the `section` tag in an HTML 4 document, whereas this tag is only valid in HTML 5. We can add schemas in *Settings → Languages & Frameworks → Schemas and DTDs*. The default schema used by PyCharm is HTML 5.



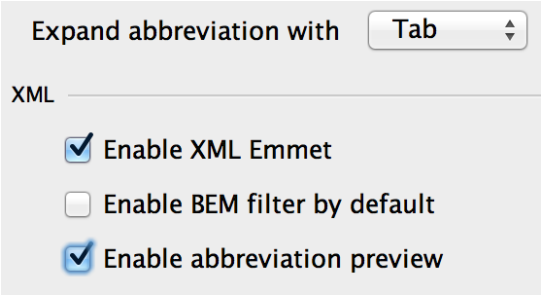
Emmet (Zen Coding)

Emmet, previously known as Zen Coding, is a shorthand toolkit used to help write HTML and CSS faster. In the following example, we can see how the shorthand is expanded:

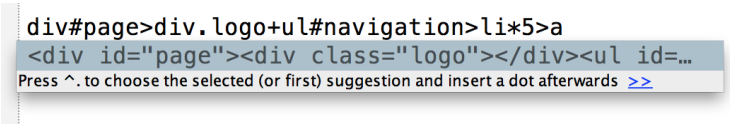
```
div#page>div.logo+ul#navigation>li*5>a
```

```
<div id="page">
    <div class="logo"></div>
    <ul id="navigation">
        <li><a href=""></a></li>
        <li><a href=""></a></li>
        <li><a href=""></a></li>
        <li><a href=""></a></li>
        <li><a href=""></a></li>
    </ul>
</div>
```

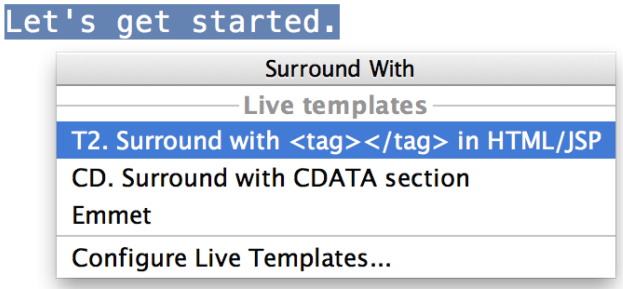
By default the expansion is trigged by the *Tab* key, but we can change it to the *Space* or *Enter* key in *Settings* → *Editor* → *Emmet*.



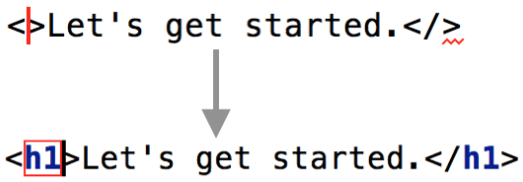
Also, you may want to check the option "Enable abbreviation preview," to preview the expansion while typing.



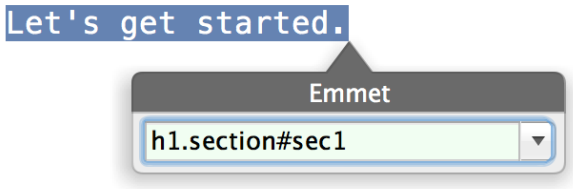
To place a text between tags, we select the text and go to *Code → Surround With… (⌘-⌥-t, A-C-t)*. The "Surround With" popup allows us to choose how to add the tags.



The first option, "Surround with <tag></tag> in HTML/JSP," will let us type an arbitrary tag.



The third option, Emmet, is useful for more complex tags:

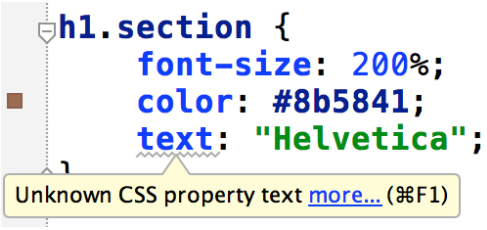


Where this is the result:

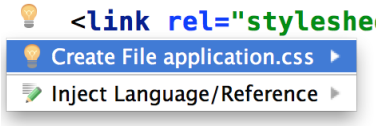
```
<h1 class="section" id="sec1">Let's get started.</h1>
```

CSS

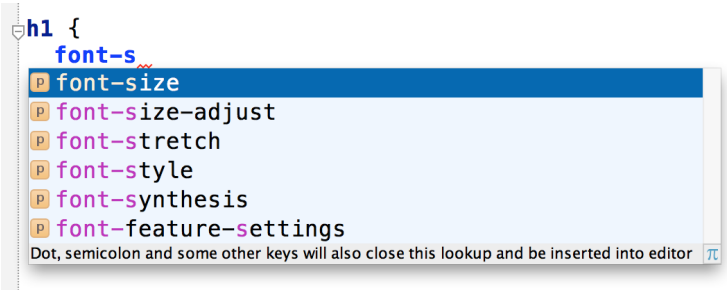
PyCharm knows about CSS and will let us know if we try to use an unknown property:



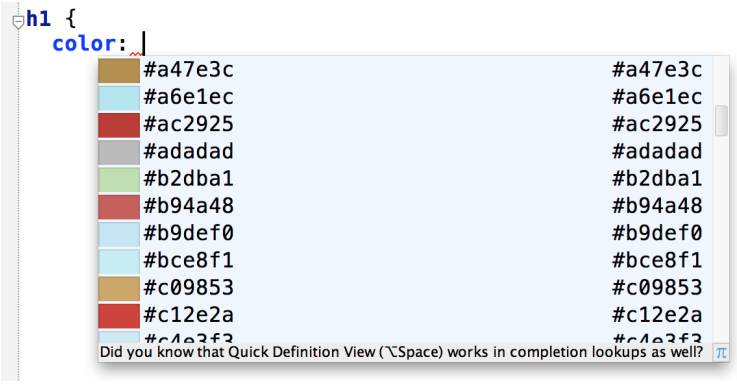
When we refer to an inexistent style sheet in an HTML document, PyCharm will offer to create the file:



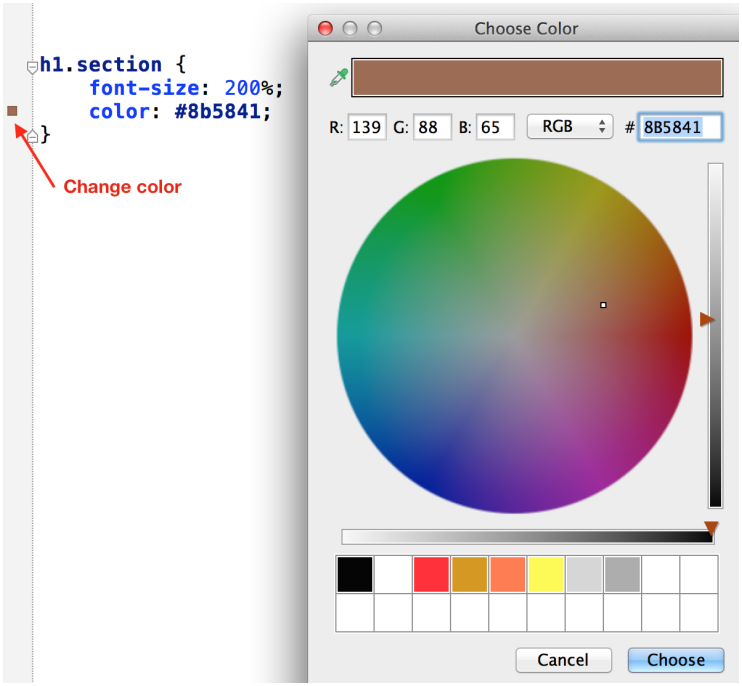
As expected, PyCharm has great completion for CSS:



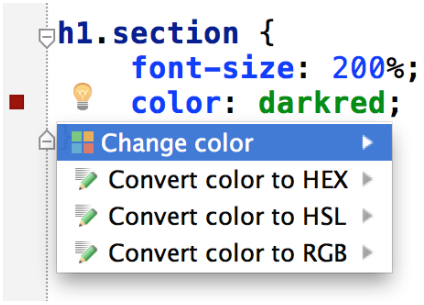
And the completion is even smart about colors:



By clicking on the small square in the left gutter area, we can change the color by using a graphical color wheel.



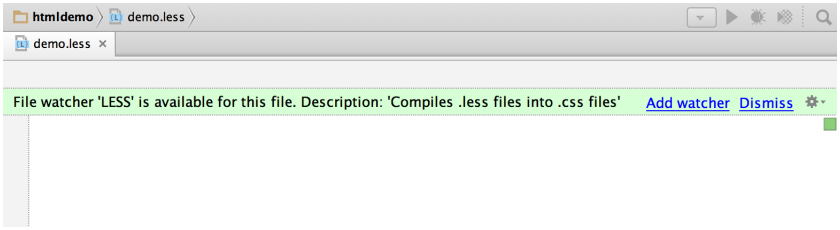
When the cursor is on the color value, PyCharm will offer to change the color or convert it to HEX, HSL, or RGB.



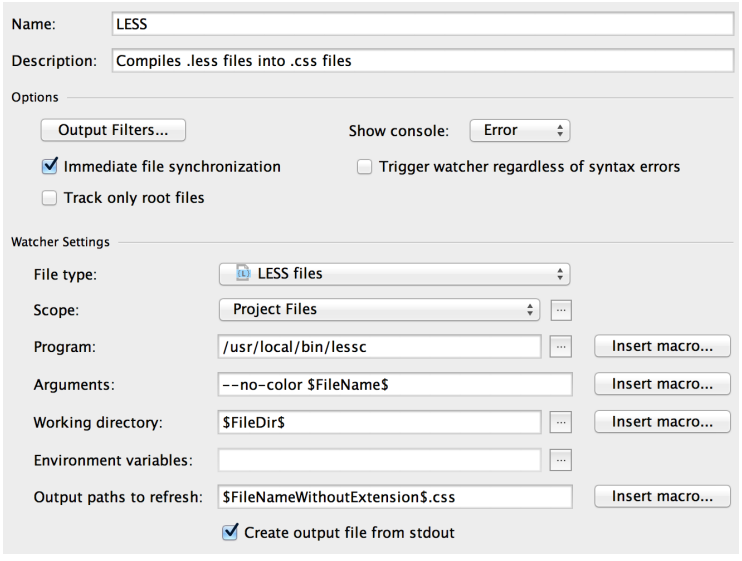
SASS, LESS, and SCSS

PyCharm has some support for SASS, LESS, and SCSS, using *File Watchers* to automatically convert to CSS. In this section I use LESS, but the process is the same for SASS and SCSS.

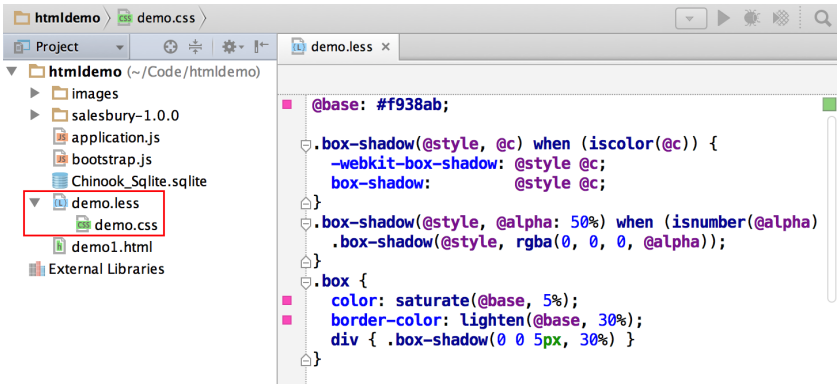
When we open a LESS file, PyCharm will recognize the format and offer to add a watcher for us.



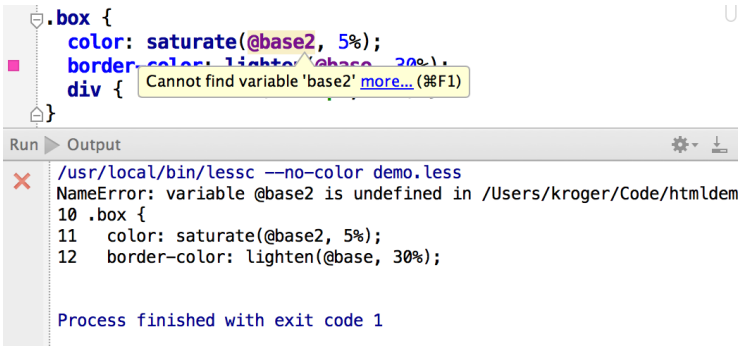
If LESS is installed, PyCharm will recognize it and fill the options automatically:



In the project window on the left, PyCharm shows the generated CSS file coupled with the LESS file. We can still open and change the CSS file, but it will be regenerated every time the LESS file is changed.



PyCharm shows errors as a tooltip and result of running the `lessc` tool as well:

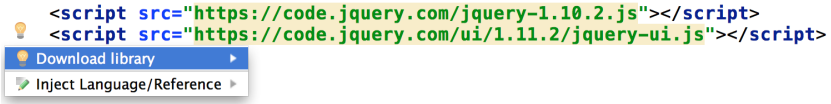


JavaScript

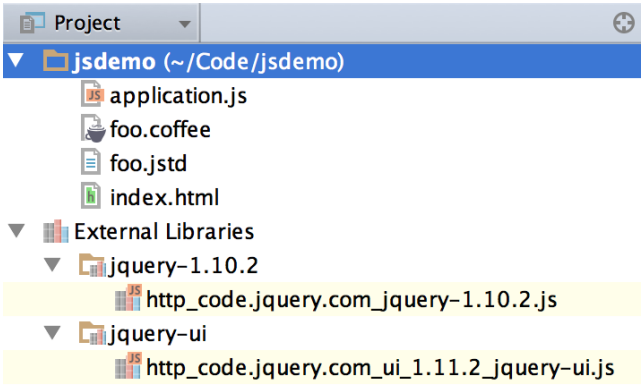
In this section, we will see the main features in using JavaScript in PyCharm. Because we can't cover every feature here, you should also check the manual for more JavaScript-related features such as JSDoc comments, unit testing, and minifying.

When we use a JavaScript library that is not stored locally, PyCharm likes to download a local copy so it can be used for completion and code analysis.

If we click on the icon or use a shortcut (⌥-*Enter, A-Enter*) to download the library, as we have seen in *Intention Actions*, PyCharm will save it in the local cache.



The downloaded library is easily accessible in the project window in the External Libraries section:

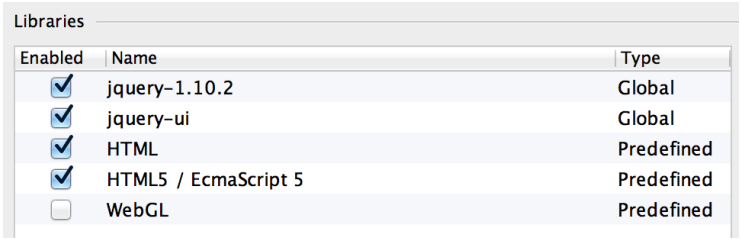


If PyCharm doesn't recognize an external library, we may need to activate it. In the following example, the method `datepicker` from jQuery UI is not recognized because jQuery UI is deactivated.

```
$(function() {
    $( "#datepicker" ).datepicker({
        changeMonth: true
        changeYear: true
    });
});
```

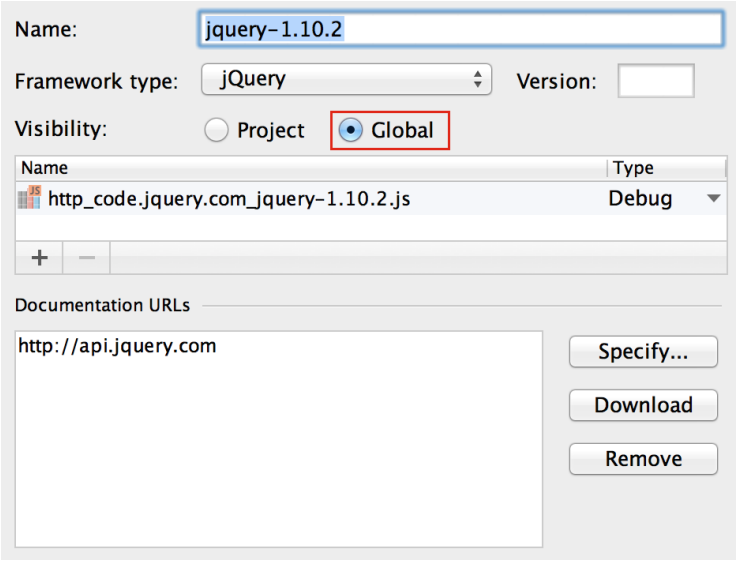
Unresolved function or method datepicker()

We can enable or disable a library by going to *Settings → Languages & Frameworks → JavaScript → Libraries*.



An important configuration is the visibility of the library; it can be visible

only in the current project or globally.

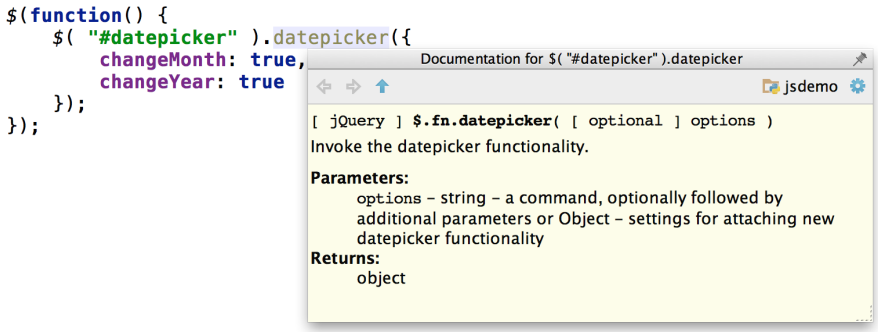


After setting up the library, completion works as usual:

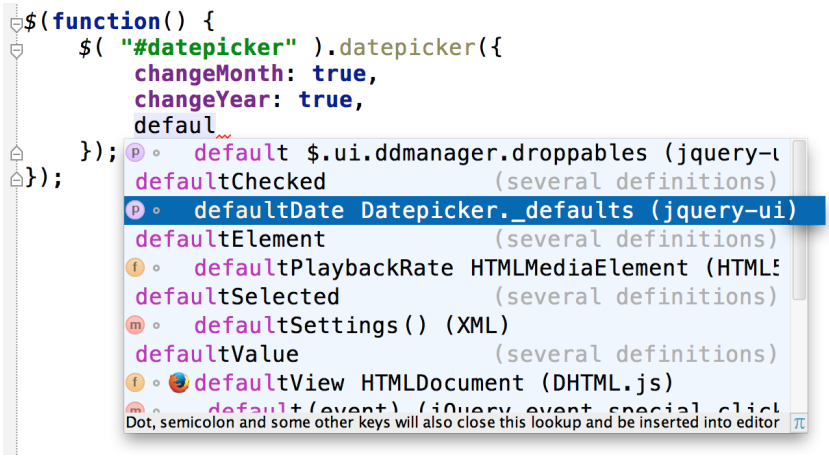
```
$(function() {
    $( "#datepicker" ).datepicker({
        changeMonth: true,
        changeYear: true
    });
});
```

Choose Declaration
datepicker $.ui (jquery-ui)
datepicker $ (jquery-ui)
datepicker $ (jquery-ui)

So do external documentation and quick documentation (see *Documentation*):



And completion works for external libraries as expected:

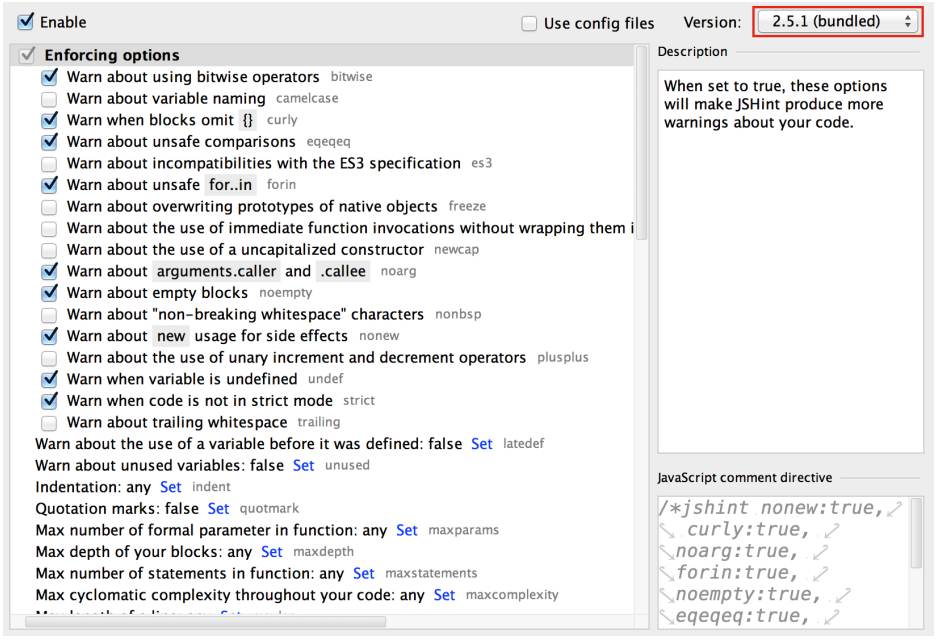


Code Quality

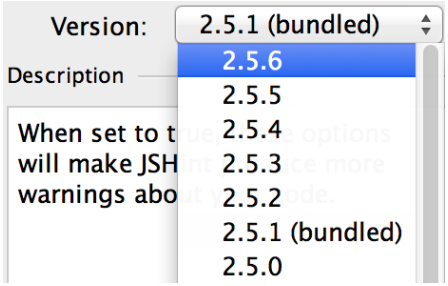
We can expand PyCharm's JavaScript knowledge by using a linter to perform static code analysis. PyCharm has support for five linters: JSLint, JSHint, Closure Linter, JSCS, and ESLint. JSLint and JSHint are bundled with PyCharm, whereas the others need to be downloaded. We can enable the linter in *Settings → Languages & Frameworks → JavaScript → Code Quality*

Tools. In this section, we will see how to work with JSHint. The process will be similar for the other linters.

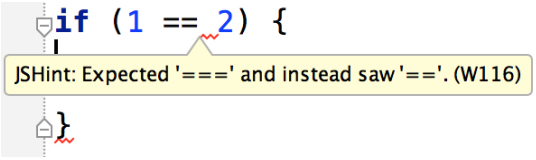
We have lots of options to enable or disable in the configuration.



Also, we can choose a specific version of JSHint to use. If the version is not available locally, PyCharm will download it automatically.



After being enabled, JSHint will show potential problems in a tooltip window.



We can use a configuration file instead of the graphical interface. It's useful to use the same configuration across different projects, to ensure consistency.

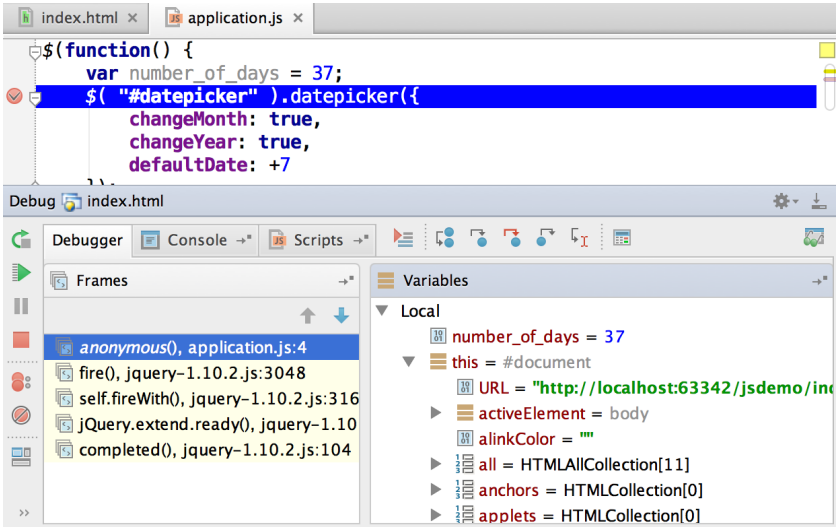
Debugging JavaScript

In PyCharm, debugging JavaScript only works with Firefox and Google Chrome. For Chrome you need to install a plugin. There are some issues with some versions of Firefox, Chrome, and PyCharm. As of this writing, these issues have been solved with Chrome in the latest PyCharm version (4.0.4) but debugging with Firefox doesn't seem to be working.

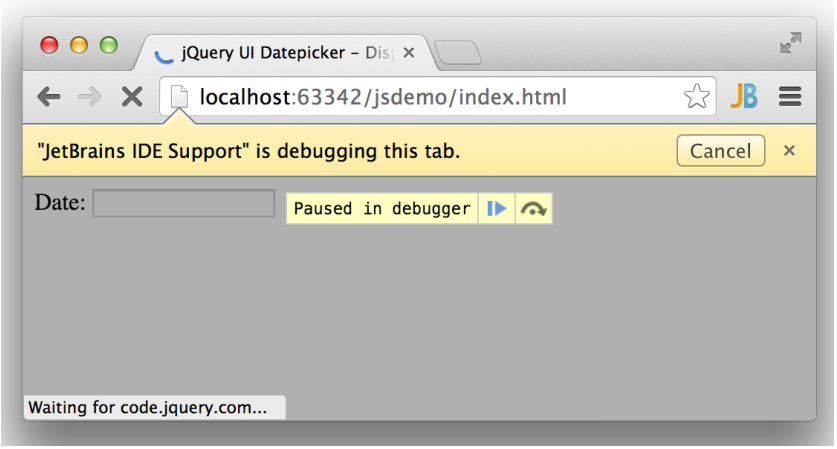
Debugging JavaScript in PyCharm used to take multiple steps, but with the latest PyCharm version, it's as simple as setting a breakpoint and clicking on the debug icon.

```
$(function() {
    var number_of_days = 37;
    $( "#datepicker" ).datepicker({
        changeMonth: true,
        changeYear: true,
        defaultDate: +7
    });
});
```

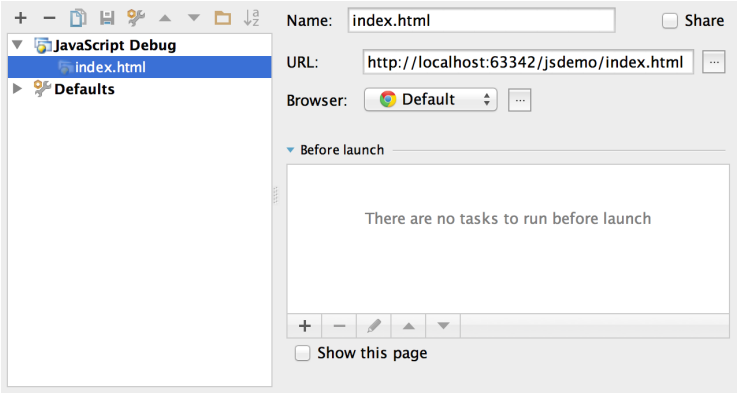
Once the debugger starts, we can perform the usual actions such as inspecting variables and stepping through code.



PyCharm will automatically open the browser and inject the code to be debugged. The browser opened will not necessarily be the default browser in your system.



We can change the browser to run the JavaScript code in *Run → Edit Configurations…*.



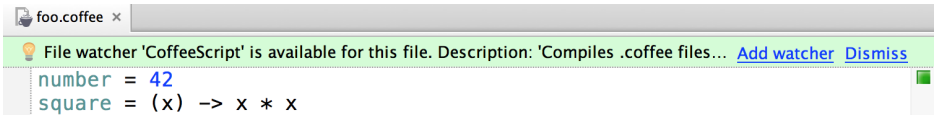
CoffeeScript

To use CoffeeScript in PyCharm, we need to install Node.js and CoffeeScript.

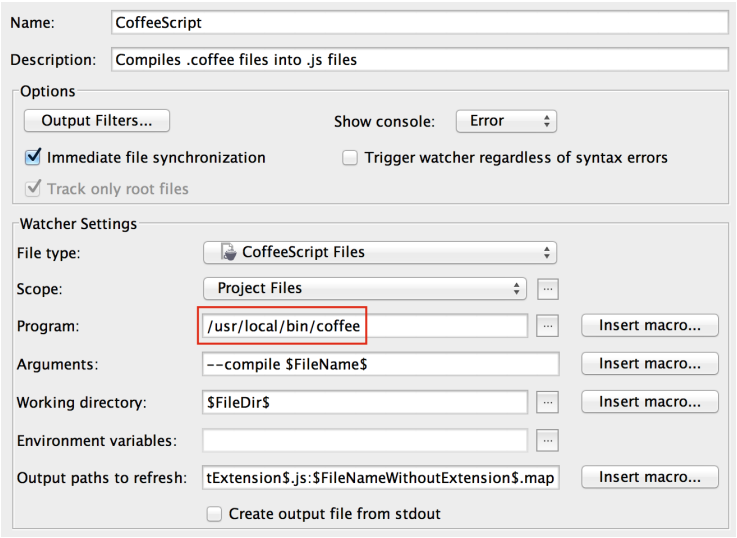
PyCharm has full support for CoffeeScript, including debugging. It's

necessary to install the `NodeJS` plugin by going to *Settings* → *Plugins*. To set it up, go to *Settings* → *Languages & Frameworks* → *Node.js and NPM* and select the node interpreter.

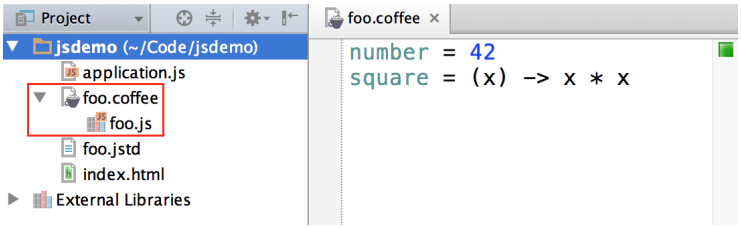
When we open a CoffeeScript file for the first time, PyCharm asks if we want to add a file watcher (see *File Watchers*), which we do.



If we have installed Node and CoffeeScript correctly, PyCharm should detect the `coffee` program automatically:



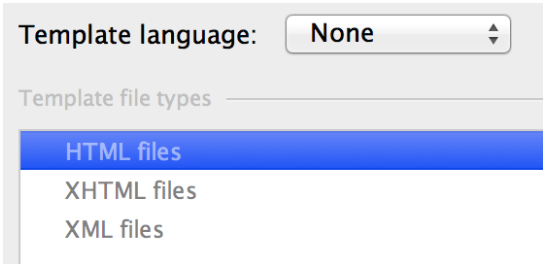
Every change to a CoffeeScript file will generate a corresponding JavaScript file. The two files are coupled in the project window. We can open and edit the generated JavaScript file, but it will be regenerated every time the CoffeeScript file is changed.



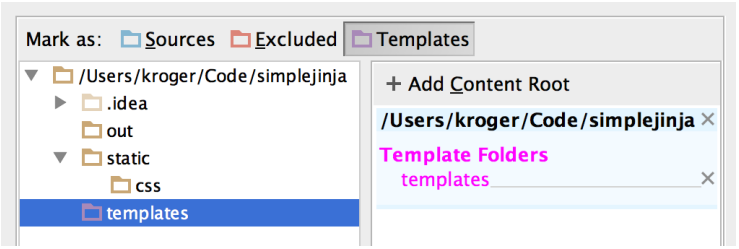
Static HTML with Jinja

Although we often need the full power of a framework like Django or Flask, sometimes we just need to show data using a static web page. An easy way to do so is to make a simple Python script to send the data to some Jinja templates.

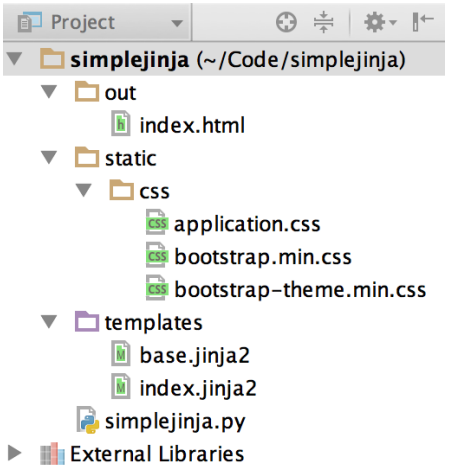
In this project, it's not necessary to select a template language in *Settings → Languages & Frameworks → Python Template Languages*; we can leave it as "None":



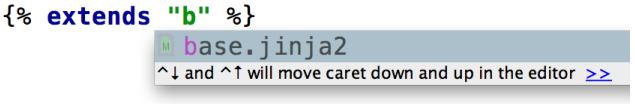
We need to tell PyCharm where to find the templates for the current project. In *Settings → Project: <project name> → Project Structure* select the "templates" folder and mark it as "Templates" in the top.



In this project, the script will be in the the root folder (`simplejinja.py`), the generated files in the `out` folder, and the templates and CSS files in the `templates` and `static` folders, respectively.



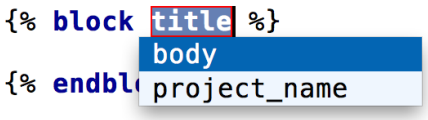
Now PyCharm knows where to find the templates and will complete the code accordingly.



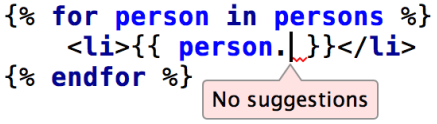
As usual, we can define a base layout using blocks:

```
<!DOCTYPE html>
<html lang="en">
  <head>
...
    <title>{% block title %}{% endblock %}</title>
    <link href="{{ STATIC_URL }}css/bootstrap.min.css" rel="stylesheet">
    <link href="{{ STATIC_URL }}css/bootstrap-theme.min.css" rel="stylesheet">
    <link href="{{ STATIC_URL }}css/application.css" rel="stylesheet">
  </head>

  <body>
    <div class="navbar navbar-inverse navbar-fixed-top" role="navigation"...>
    <div class="container">
      <div class="starter-template">
          {% block body %}
          {% endblock %}
      </div>
    </div>
  </body>
</html>
```

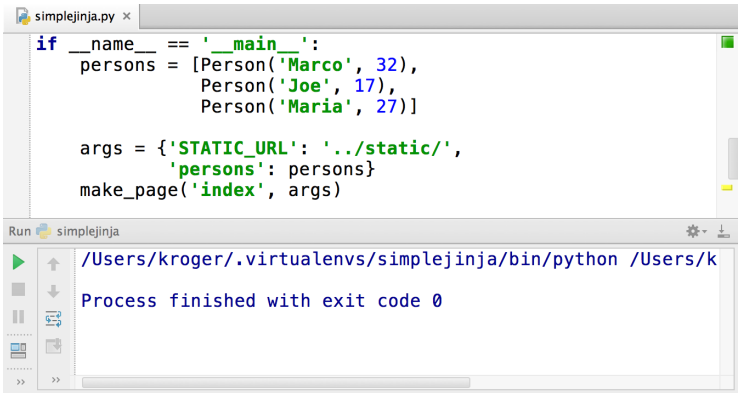
And inherit the base layout in a template file. PyCharm understands Jinja's template inheritance and will show the defined blocks for completion:



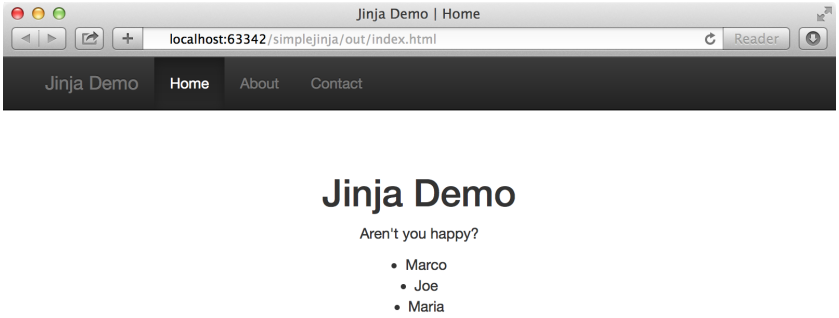
However, PyCharm won't complete variables and objects passed to the template from the script. (This kind of completion works with Django, Flask, and Pyramid.)



We can run the script from the *Run* menu, as we have seen in *Running Code*. In the following example, I'm passing a list with `Person` objects to the template.



And this is the result:



Django

Basic Usage

PyCharm has superb support for Django. The Django-specific features with the features we have seen in the rest of this book, such as *Editing and Navigating*; *Running, Debugging, and Testing*; *Version Control*; access to *External Tools*; *File Watchers*; and full support for *HTML and CSS* and

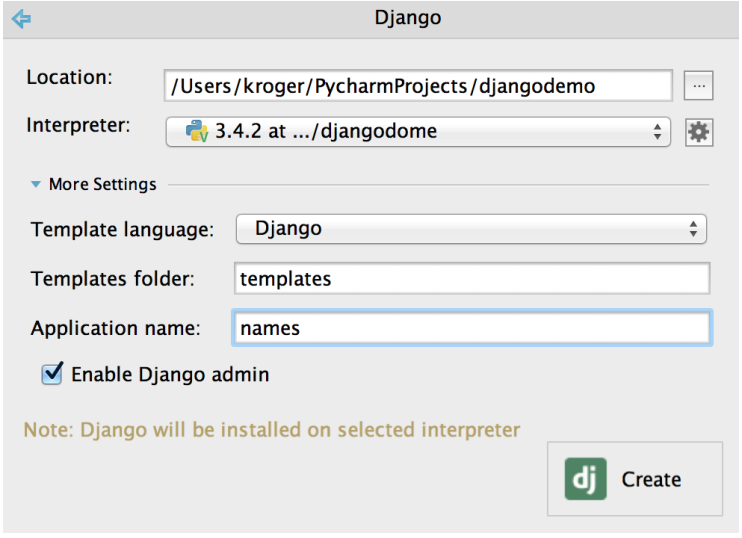
JavaScript, make PyCharm the most complete IDE for Django.

An introduction to Django is out of the scope of this book. The quickest way to get started is to follow the Django tutorial. If you already know Django, Two Scoops of Django: Best Practices for Django by Daniel Greenfeld and Audrey Roy is a great way to learn the best practices.

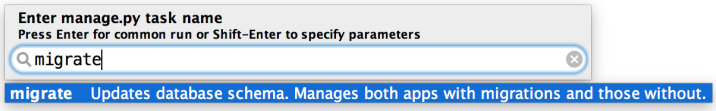
In this chapter I'm using Django 1.7 with Python 3.4, but the process is similar for other versions.

The simplest way to start a new Django project is by selecting *Create New Project → Django* in the welcome window.

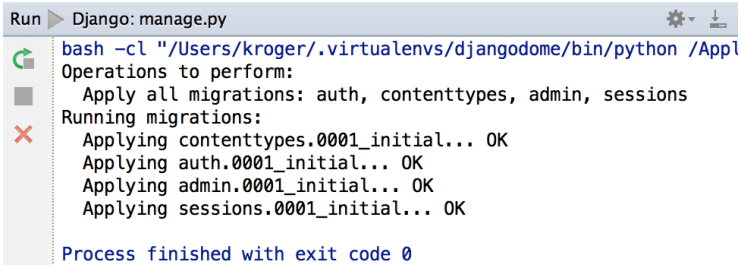
We can set the project's location and Python interpreter (don't forget to create a new *Virtualenv and Packages*!), and choose a template language and the location where the HTML templates will be stored. If you add an application name, PyCharm will create the application and add it to the settings file. If you leave the application name empty, only the project will be created. If the "Enable Django admin" option is enabled, PyCharm will set up the admin site for you.



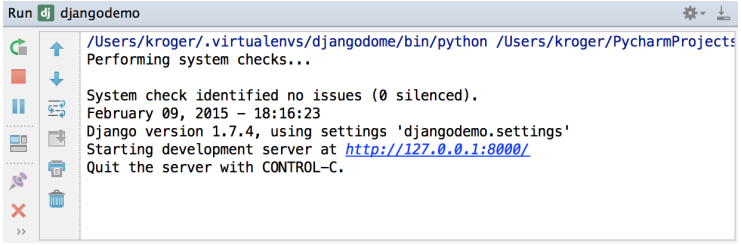
Since Django 1.7 has migration built-in (yay!) and the `startproject` command now creates a SQLite file by default, we need to run the `migrate` command by going to *Tools → Run manage.py Task…* and selecting "migrate."



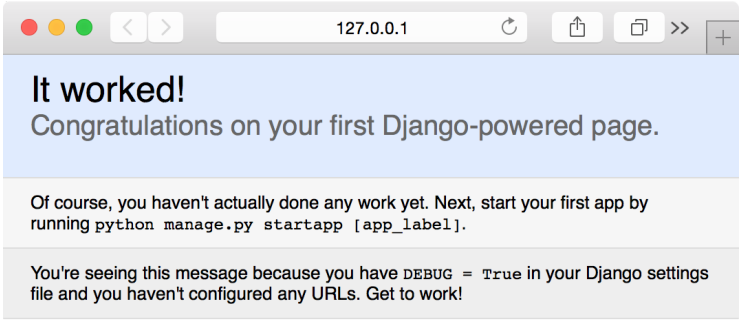
All necessary migrations will be applied, as we can see in the tool window.



Having applied the migration, we can run the basic project as usual (see *Running Code*) and access the resulting web page by clicking on the link.



And this is the result, as expected:



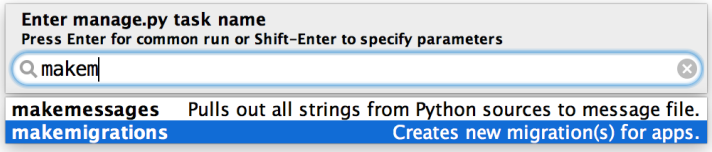
To see how the new migration feature works, let's add a class to the model. If we haven't set up PyCharm to access the database, we may see a warning message. It's fine to dismiss the message and not configure the database access. But, as we have seen in *Databases*, it's quite handy to be able to access the app's data inside PyCharm.

Our class is just a person with a name:

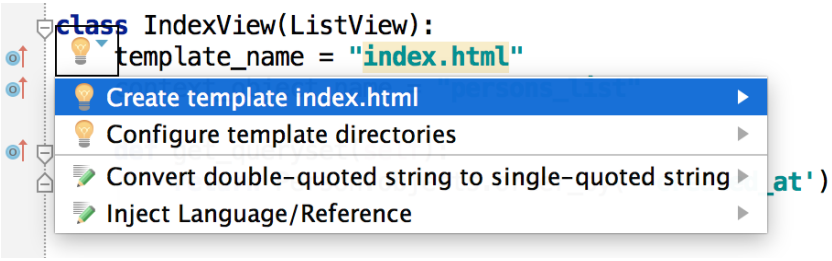
models.py

```
from django.db import models


class Person(models.Model):
    name = models.CharField(max_length=200)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.name
```

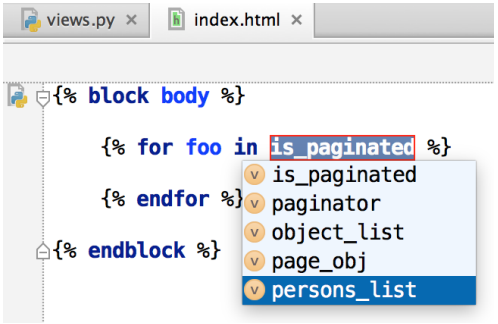
When making changes to a model in Django 1.7, we need to create a migration with the command `makemigration` and apply it with `migrate`. Again, we can perform these commands by going to *Tools → Run manage.py Task...*.



Now let's display a list of people in a page by creating a simple class-based view. PyCharm offers to create the `index.html` template, since it doesn't exist yet:



If we accept the suggestion to create the template (see *Intention Actions*), PyCharm will create and open the template file. As expected, we have completion in the template, and it recognizes the object name of the context (`context_object_name` in the view).

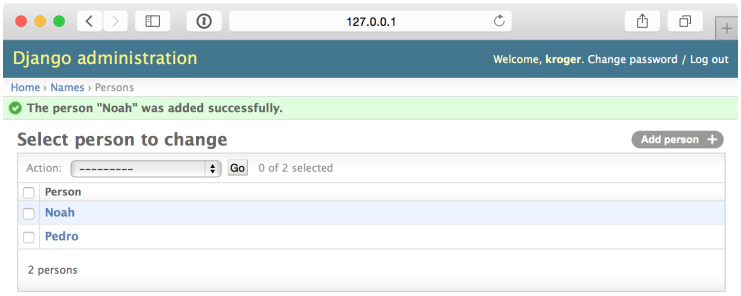


We can jump to the template from the view easily by clicking the icon on

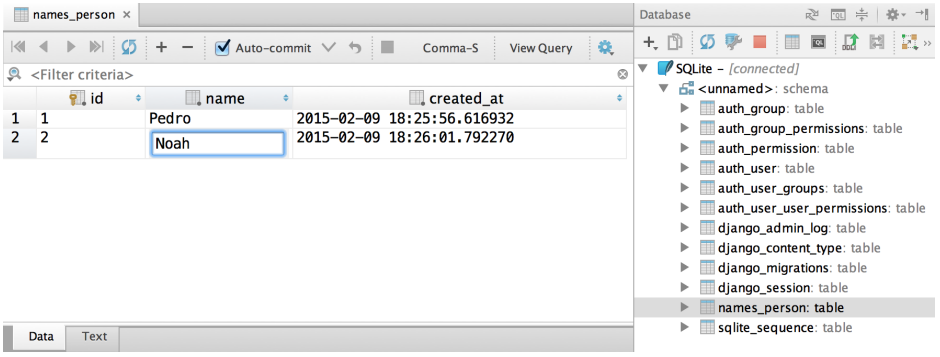
the left gutter (see *Finding Your Way in the Source Code* for more ways to navigate code).

```
class IndexView(ListView):
    template_name = "index.html"
    context_object_name = "persons_list"

    def get_queryset(self):
        return Person.objects.order_by('-created_at')
```

Since the Django admin site is enabled by default when we create a new Django project with PyCharm, we can easily add objects by using the admin:



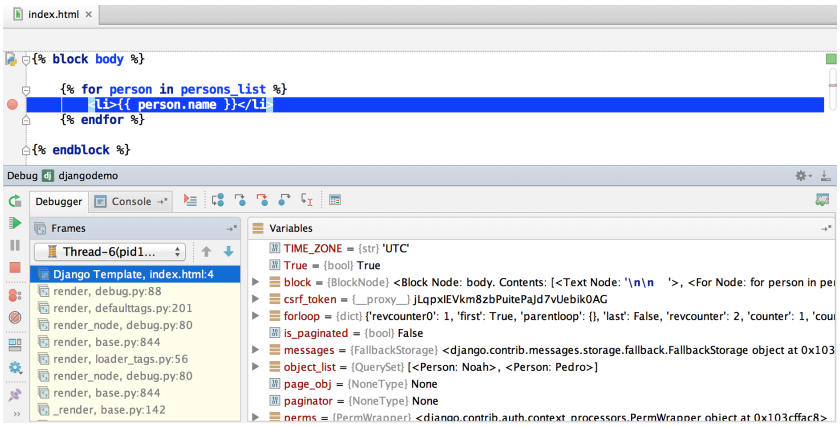
Also, if we have configured access to the database, we can make changes to the data from PyCharm itself:



As we can see, we can have a functional, database-backed site in only a few steps.

Template Debugging

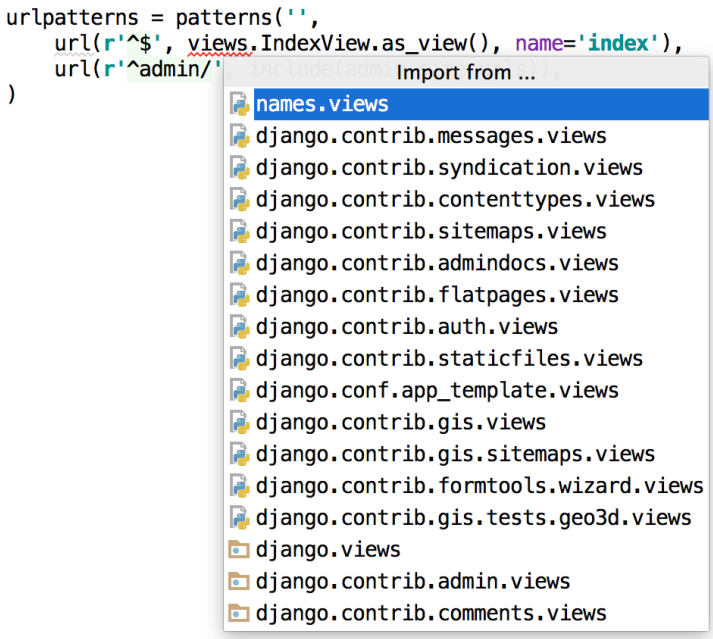
Debugging works with Django and Jinja2 templates. We can inspect variables, step through code, and do what we expect in a debugger (see *Debugging Code*).



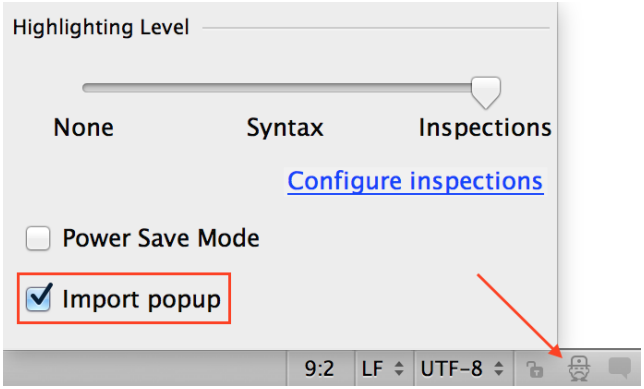
Importing Code

I have the memory of a brainless goldfish (yes, I know it's a misconception—I need to find another animal to pick on), so I can never remember which symbol corresponds to which package. Luckily, PyCharm offers to import symbols automatically. For instance, we can just type `ListView` and PyCharm will offer to import it from `django.views.generic`, which is what we want:

This feature is also useful when importing a symbol whose name is present in multiple packages. In the following example, the symbol `views` is available is many different packages, but we can easily choose the one we want:



If you don't like the automatic import option, click on the Hector icon on the status bar (see more about the Hector icon and the status bar in the manual) and uncheck "Import popup" (see also *Disabling Inspections and Intentions*).



Finally, another useful option is *Code → Optimize Imports…* (⌥-*C-o*, *A-C-o*), which removes unused import statements and sorts them according to PEP-8.

Django Console

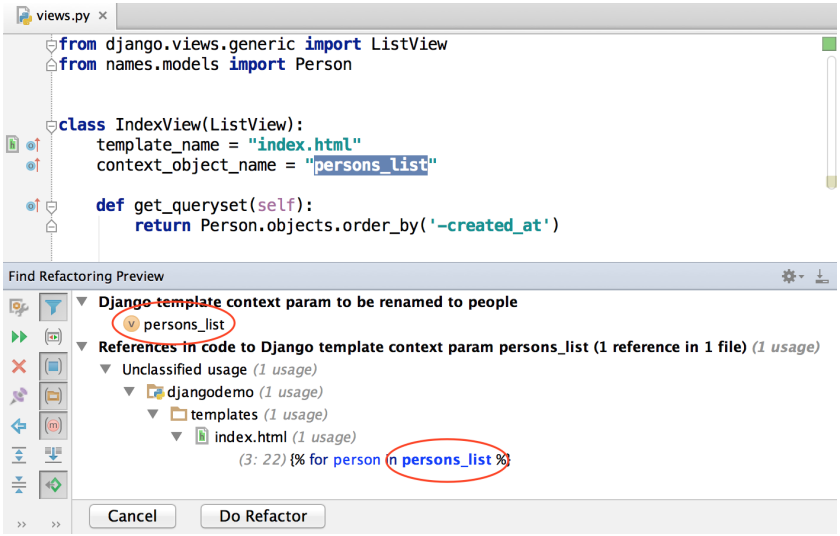
Older versions of PyCharm used to have separate consoles for Python and Django. Since PyCharm 4, they have been merged into a single Python console in *Tools → Python Console…* (see *Console*). If your project was created with an earlier version of PyCharm, you may have to add the following code in *Settings → Build, Execution, Deployment → Console → Django Console* in the "Starting script" text box:

```
import sys;
print('Python %s on %s' % (sys.version, sys.platform))
import django;
print('Django %s' % django.get_version())
sys.path.extend([WORKING_DIR_AND_PYTHON_PATHS])
if 'setup' in dir(django): django.setup()
import django_manage_shell
django_manage_shell.run(PROJECT_ROOT)
```

And since the options in *Settings → Build, Execution, Deployment → Console → Django Console* are valid only for the current project, we can import other packages relevant to the current project. For instance, in the following example, I'm importing all classes from the model, so they will be readily available:

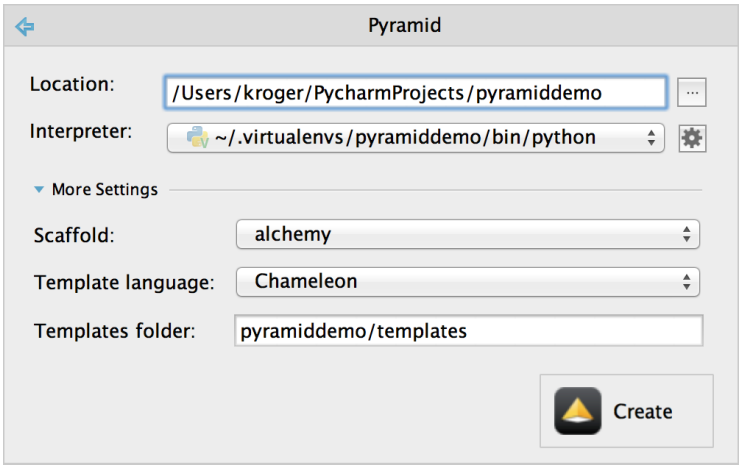
Refactoring

Refactoring is one of the defining features of a good IDE. We have already seen how it works in PyCharm (in *Refactoring*), so we won't go over it again. One big advantage of using a proper refactoring tool in a framework like Django is that it guarantees the consistency of the code while refactoring. In the following example, I'm renaming the `context_object_name` in `views.py`. PyCharm will also rename it in the template file.

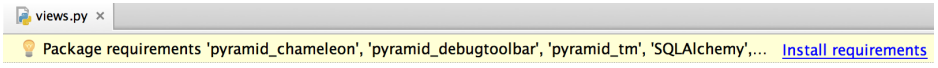


Pyramid

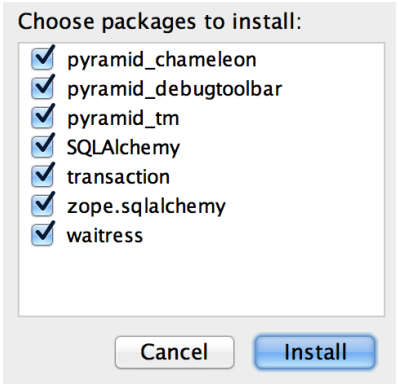
The easiest way to start a new Pyramid project is by going to *Create New Project → Pyramid* on the welcome window. We can set the project's location and Python interpreter (don't forget to create a new *Virtualenv and Packages*!), and choose a scaffold (starter, zodb, or alchemy), a template language (Chameleon is the default), and a location where the HTML templates will be stored. In this section, I'll use the alchemy scaffold with the Chameleon template language. This scaffold uses URL Dispatch to map URLs to view code and SQLAlchemy for persistence.



PyCharm will list the required packages in the `setup.py` file and ask if you want to install them.

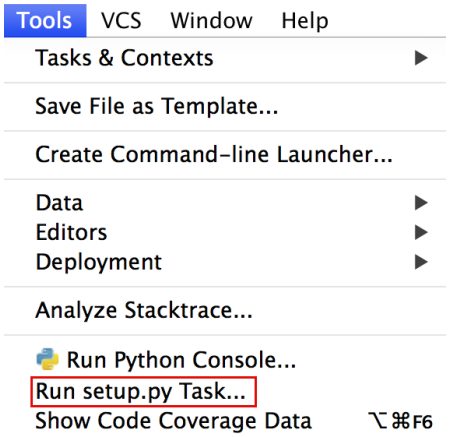


After clicking on "Install requirements," we just need to click on the Install button:

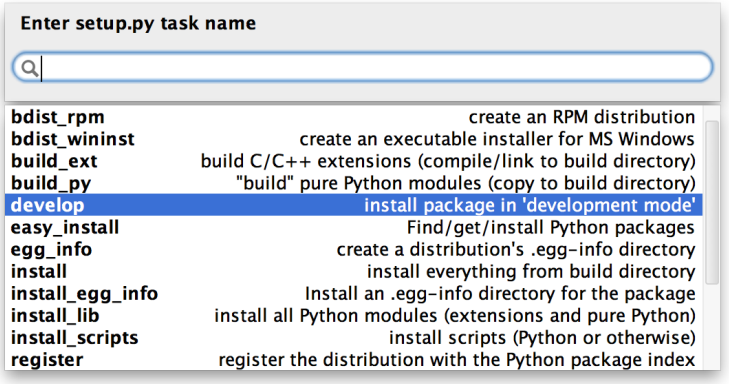


Install the project in development mode (see Pyramid's documentation). We

can run `python setup.py develop` in the terminal or run `setup.py` commands from *Tools → Run setup.py Task...*.

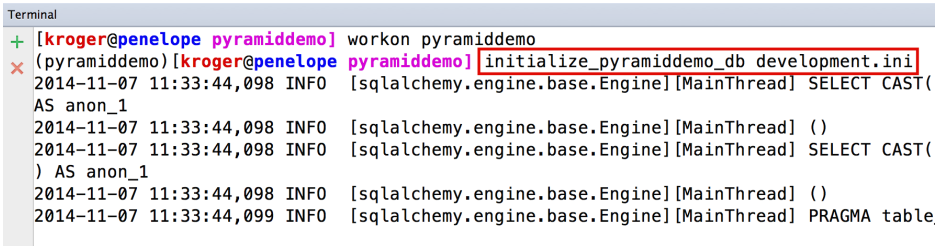


Select the "develop" task in the window:

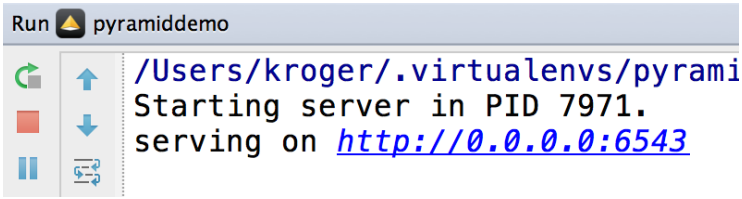


Initialize the database by using a console script named `initialize_<project name>_db`, for instance:

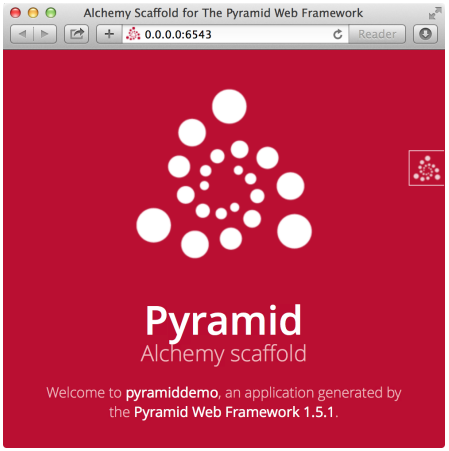
```
initialize_pyramiddemo_db development.ini
```



Start the server by running the project in *Run → Run '<project name>' (C-r, S-F10)*, as we have seen in *Running Code*. We can access the resulting web page by clicking on the link.

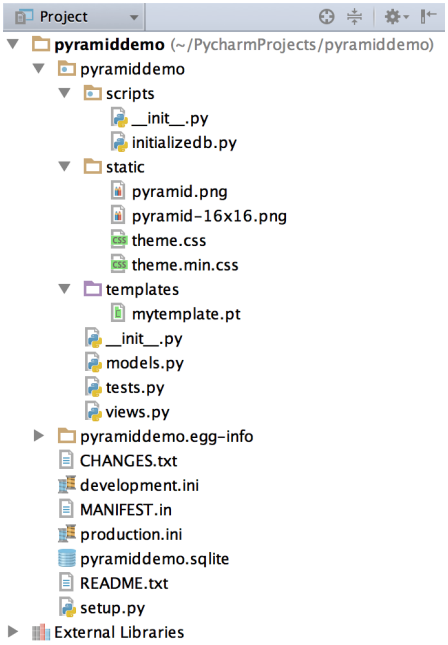


This is the result:



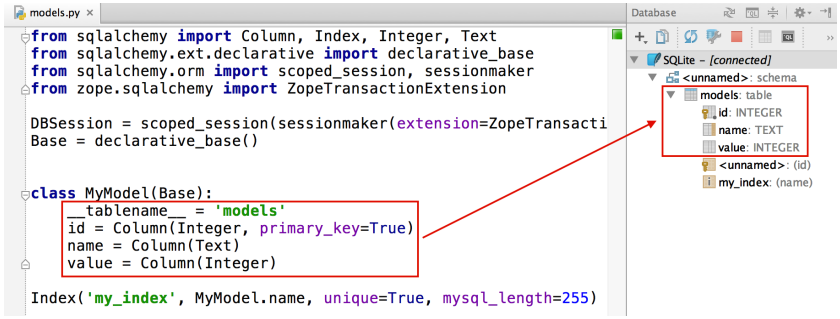
As we can see, the scaffold will create all of the files we need to get started,

including setup and configuration files:

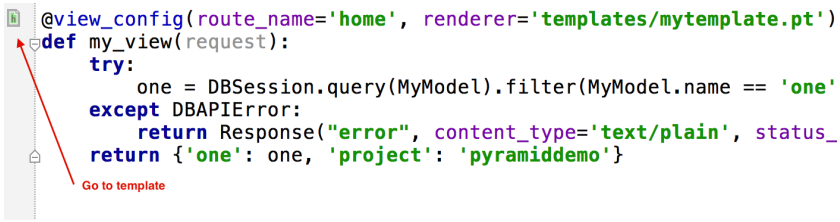


Data Source

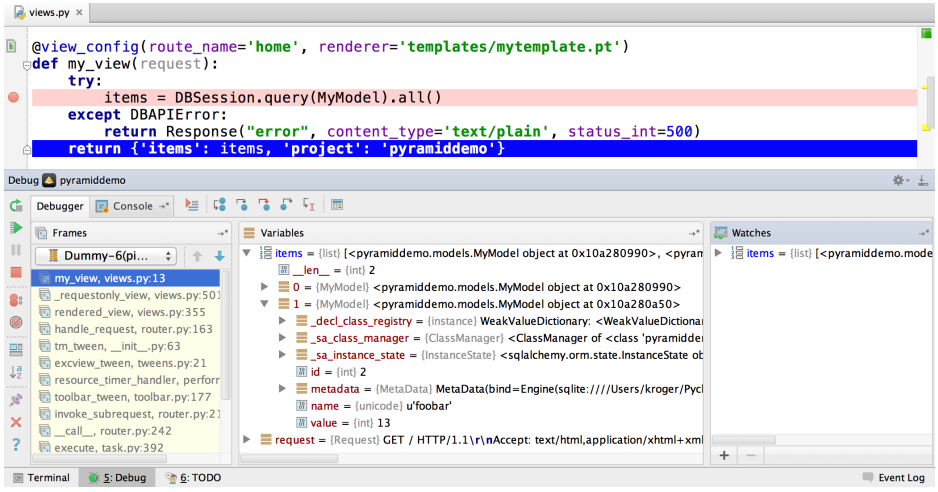
After having configured the database access from PyCharm (see *Databases*), we can see it matches the model defined in `models.py`.



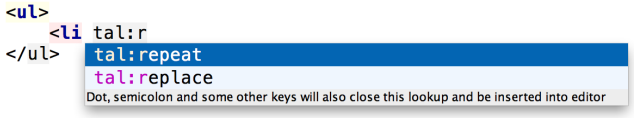
As usual, we can jump from a view to a template by clicking on an icon on the left:



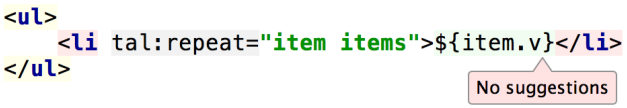
Debugging works as expected (see *Debugging Code*):



And so does completion for Pyramid and the Chameleon template:

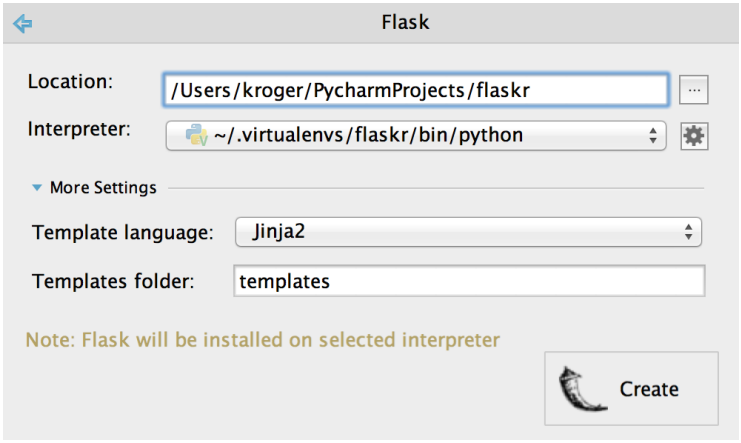


But completion fails to complete attributes from the model in some cases. Also, the debugger doesn't work with the Chameleon template.



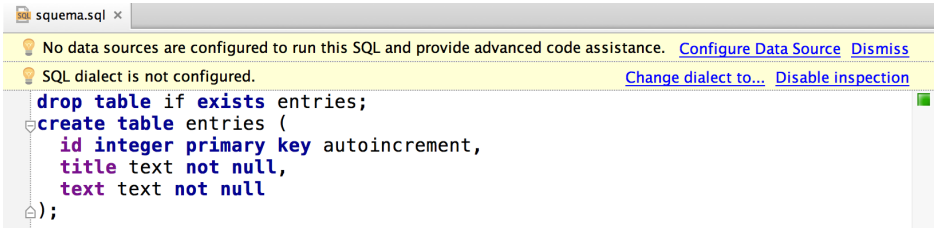
Flask

The easiest way to start a new Flask project is by going to *Create New Project* → *Flask* on the welcome window. We can set the project's location and Python interpreter (don't forget to create a new *Virtualenv and Packages*!) and choose a template language and where the templates will be located. In this section, I'll use the Jinja2 template language.

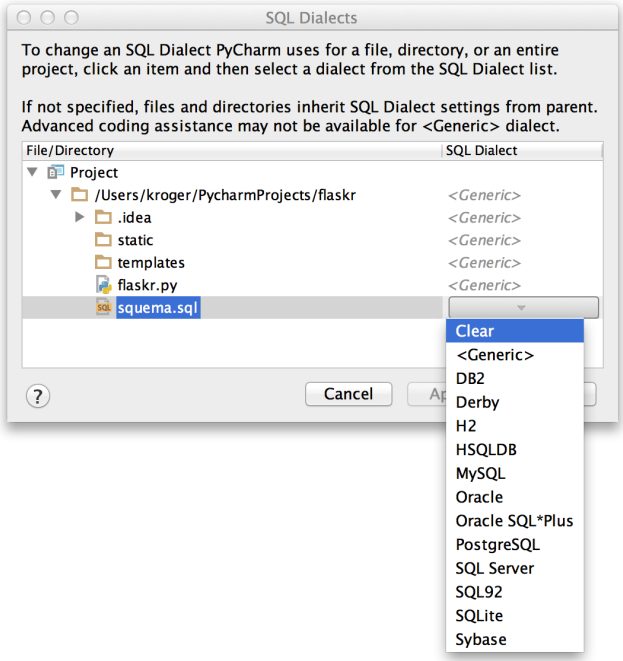


PyCharm will create a basic "Hello World!" project that can be run by going to *Run → Run '<project name>' (C-r, S-F10)*, as we have seen in *Running Code*.

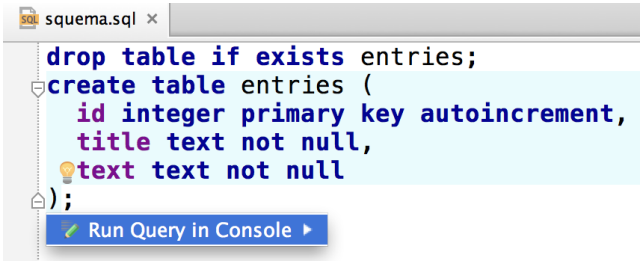
To add a database, we create a file named `squema.sql` and add SQL code to create some tables. PyCharm will recognize the file and ask us to configure a data source (see *Databases*) and to set up the database dialect (in this section, I'm using SQLite).



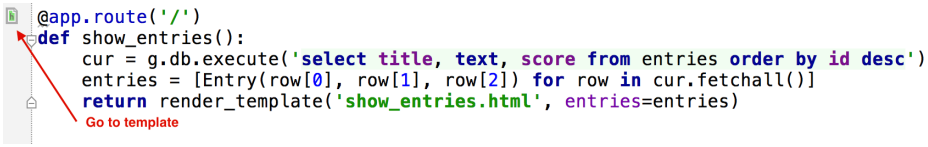
When we click on "Change dialect to…," PyCharm will let us choose the SQL dialect we want to use. We can also change the SQL dialect by going to *Settings → Language & Frameworks → SQL Dialects*.



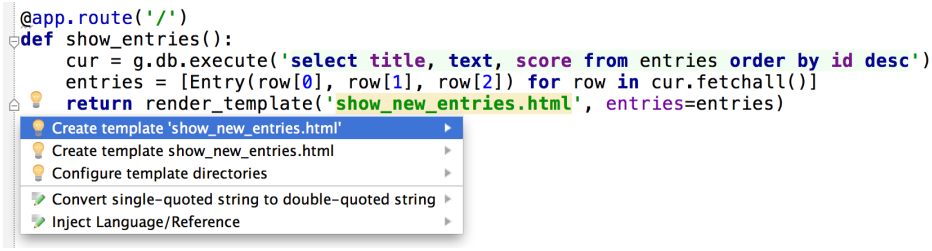
The easiest way to run the SQL query is to click somewhere in the query, click the inspection icon, and click on "Run Query in Console." (It is much easier than it sounds.)



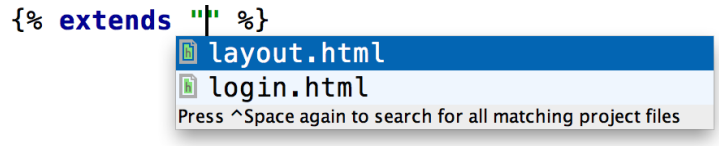
As expected, we can jump to a template from a view:



If a view refers to an inexistent template, PyCharm will offer to create it.

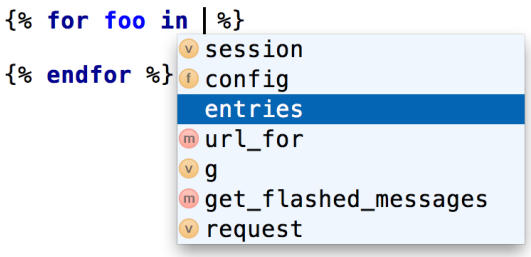


PyCharm has great completion for Jinja2 templates, including listing the templates to extend,

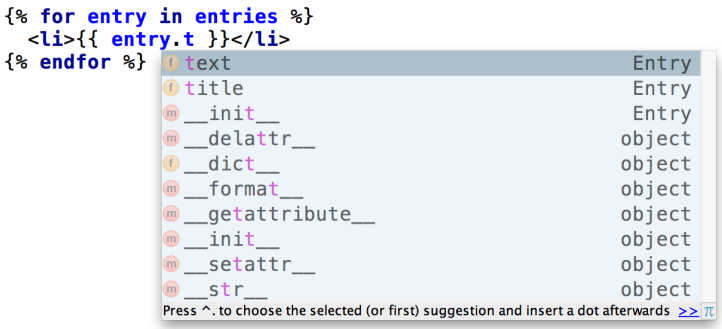


completion of blocks,

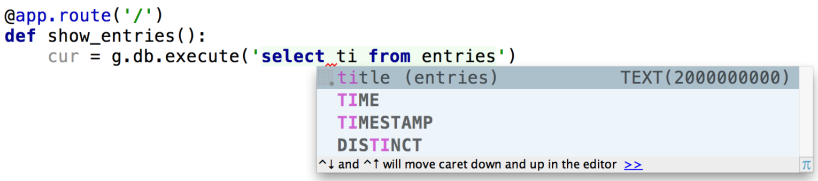
and completion of sequences:



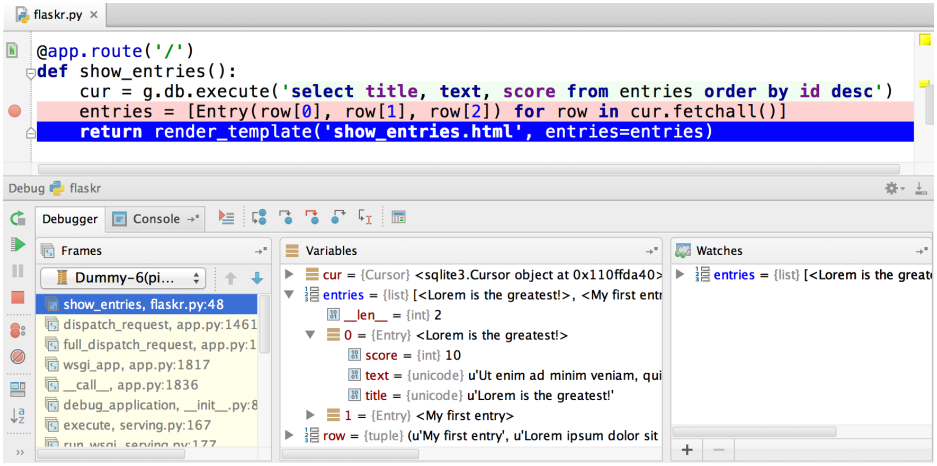
Objects' attributes that are used in templates have completion, as well:



We also have completion for SQL inside strings:



Finally, debugging works as usual, including Jinja2 templates.


flaskr.py
@app.route('/')
def show_entries():
cur = g.db.execute('select title, text, score from entries order by id desc')
entries = [Entry(row[0], row[1], row[2]) for row in cur.fetchall()]
return render_template('show_entries.html', entries=entries)
Debug flaskr
Debugger
Console
Frames
Variables
Watches
Dummy-6(pi...
show_entries, flaskr.py:48
dispatch_request, app.py:1461
full_dispatch_request, app.py:1
wsgi_app, app.py:1817
__call__, app.py:1836
debug_application, __init__.py:8
execute, serving.py:167
cur = {Cursor} <sqlite3.Cursor object at 0x110ffda40>
entries = {list} [<Lorem is the greatest!>, <My first entr
__len__ = {int} 2
0 = {Entry} <Lorem is the greatest!>
score = {int} 10
text = {unicode} u'Ut enim ad minim veniam, qui
title = {unicode} u'Lorem is the greatest!'
1 = {Entry} <My first entry>
row = {tuple} (u'My first entry', u'Lorem ipsum dolor sit
entries = {list} [<Lorem is the great