Effective PyCharm

Learn the PyCharm IDE with a Hands-on Approach

Effective PyCharm

Learn the PyCharm IDE with a Hands-on Approach

Michael Kennedy Matt Harrison

effectivepycharm.com

Try the accompanying course on PyCharm at https://training.talkpython.fm/ for almost 8 hours of video content.

May 1, 2019

Contents

Foreword

You know what's easy? When people you admire ask you to write a forward for a book you eagerly devoured, about a topic that is your primary job.

When Michael and Matt told us they were working on a PyCharm book, frankly, we on the PyCharm team got quite excited. Talk Python is one of the premier information and training sources for the world of Python, with a unique talent at communicating the right information in the right dose in the right tone. PyCharm is a powerful, big tool, which can be overwhelming. While Michael's Mastering PyCharm course helps, let's face it – people like books.

So we knew PyCharm would get the pro treatment and here it is: Effective PyCharm. 16 chapters covering the breadth of all the angles and corners in PyCharm.

As I reviewed the material, I must embarrassingly confide, I frequently found things I...didn't know, or don't use enough in my own coding. For example Duplicated Code in the Refactoring Chapter. I really enjoy when this happens. PyCharm is an IDE for professionals, with adoption growing like crazy, and "IDE" and "professional" combine to be a tool with a large scope, intended to be continuously learned. Which makes this book so useful and timely. As another good "code smell" about the book: they remind you in Chapter 4 that Local History can save your bacon.

"Effective PyCharm" is well-structured, with code examples, screenshots, and a repository where the material will be updated. This is an important aspect of such a work. Just like software, you want to know it will be refined and improved by people willing to stand behind their work. I also appreciate their commitment to explaining the "why" as much as the "what," which is essential on a learning journey.

Thank you for buying and studying this book. Not just because we on the PyCharm team appreciate that you are using our product. But also, thank you for supporting good folks with good hearts who work hard to independently publish Python material. I give a "Python 1994" conference talk about the first years of the Python community. We could never have predicted that Python 2019 would have an ecosystem able to support folks like Michael and Matt. Of course, it's due to such people that it happened.

Again, thank you and I hope you enjoy the book as much as I did.

Paul Everitt - JetBrains Developer Advocate for PyCharm

Chapter 1

Why PyCharm and IDEs?

1.1 Welcome to Effective PyCharm

Hello and welcome to Effective PyCharm. In this book, we're going to look at all the different features of one of the very best environments for interacting and creating Python code, PyCharm. PyCharm is an IDE (integrated development environment) and this book will teach you how you can make the most of this super powerful editor. Let's jump right into what we're going to cover.

The first thing we are going to talk about is why do we want to use an IDE in the first place? What value does a relatively heavyweight application like PyCharm bring and why would we want to use it? There are many features that make PyCharm valuable. However, let's begin by talking about the various types of editors we can use and what the trade-offs are there.

When we're working with PyCharm, we're focused on *projects*. A project is a group of Python files, other static, and data files that are associated with them. They all work together, and PyCharm has functionality to integrate them, hence the I in IDE.

We're going to start by focusing on creating new projects and working with all the files in them. You'll see there's a bunch of configuration switches we can set to be more effective. Then we're going to jump right into what I would say is the star of the show—the editor.

If you're writing code, you need an editor. You will be writing a lot of code. This includes typing new text and manipulating existing text. The editor has to be awesome and aid you in these tasks. We're going to focus on all the cool features that the PyCharm editor offers.

We'll see that source control in particular, Git and Subversion are deeply integrated into PyCharm. There are all sorts of powerful things we can do beyond git, including actual GitHub integration. We are going to focus on source control and the features right inside the IDE.

PyCharm is great at *refactoring*. Refactoring code is changing our code to restructure it in a different way, to use a slightly different algorithm, while not actually changing the behavior of the code. There are many powerful techniques in PyCharm that you can use to do this. Because it understands all of your files at once, it can safely refactor. It will even refactor doc strings and other items that could be overlooked without a deep understanding of code structures.

There is powerful database tooling in PyCharm. You can interact with most databases including SQLite, MySQL, and Postgres. You can edit the data, edit the schemes, run queries and more. Because PyCharm has a deep understanding of your code, there is even integration between your database schema and the Python text editor. Note that PyCharm has a free version and a professional version. The database features are only available in the professional version.

PyCharm is excellent at building web applications using libraries like Django, Pyramid, or Flask. It also has a full JavaScript editor and environment so you can use TypeScript or CoffeeScript. We'll look into both server-side and client-side features.

PyCharm has a great visual debugger, and we are going to look at all the different features of it. You can use it to debug and understand your application. It has powerful breakpoint operations and data visualization that typically editors don't have.

In Python, we use the concept of packaging to bundle up and share and reuse our libraries. PyCharm has many features to make and understand these packages. Remember, PyCharm understands the file structure of packages. It can do automatic imports and even automatically generate a `setup.py` file, so you can share your project with others and they can easily install it.

Profiling is a common task if you want to understand how your code is running. If your application is slow and you want it to go faster, you shouldn't guess where it is slow. PyCharm makes it easy to look at the code determine what it fast and slow, rather than relying on our intuition which may be flawed. PyCharm has some tremendous built-in visual types of tools for us to fundamentally understand the performance of our app.

PyCharm has built-in test runners for pytest, unittest, and a number of Python testing frameworks. If you are doing any unit testing or integration testing, PyCharm will come to your aid. For example, one feature you can turn on is auto test execution. If you are changing certain parts of your code, PyCharm will automatically re-run the tests.

PyCharm has integrated Jupyter notebooks. It has a special data science view with documentation and tools to help you explore your data. We will explore data science tools and how these work in PyCharm.

There are a couple of additional tools that don't really land in any of the above categories. There is a chapter with the additional tools at the end.

Last but not least, there are hundreds of plugins that you can get for PyCharm to make it do other things. For example, do you want it to have like Vim key bindings—there is a plugin for that. There are many plugins and we will explore a few.

1.2 The spectrum of editors

There's an entire spectrum of editors. On one hand, we have things like Emacs or Vim that are super fast to start, can run anywhere because they run in a terminal or a command prompt, and you can use them if you're ssh-ed into a server. These are powerful editors, but they're very much focused on editing a single file, and usually lack many of the "integrated" features of an IDE. While they are powerful and people can make these tools sing—they are not as powerful as PyCharm. They do not bring all the tooling together in one place nor do they understand all of a project and all the different files and pieces fitting together.

For example, if I was working on an on a Jinja 2 or Chameleon HTML template that renders model data in a web framework, it's unlikely that Emacs is going to be able to understand the static file structure, the CSS files, the JavaScript files, and the model data being passed from the web server.

Emacs and Vim don't seek to have a complete integrated view and are focused on editing a single file. This may inadvertently lead to certain types of programming styles where an organization puts a lot of code into a single Python file.

Somewhere in the middle of the spectrum between IDE's and text editors are tools like Visual Studio Code, Atom, and Sublime Text. These are lightweight editors, but they are GUIs. Visual

Studio Code is a GUI application that runs on Mac, Windows, or Linux. It wouldn't run on the server when you are ssh-ed in.

There are limitations to using these tools, but they understand more of the structure of projects. VS Code does have folders, plugins, and excellent editing features. It is more heavyweight than things like Emacs, and a little more restricted as it is limited where it runs.

Then there's PyCharm. It understands your code and project structure. It's more heavyweight, it takes a little more time to start, a little more space to install, however, it's worth it.

What we should be optimizing for is programmer productivity, not rapidness of tools or efficiency of memory or something silly like that. Memory is cheap and our computers are fast. You should think about where PyCharm lands on the spectrum of editors and IDEs. PyCharm lands on the far right of the spectrum of editors and IDEs, a big heavyweight tool. But the power that this tool provides is a well worth it.

1.3 Why choose an IDE?

Let's expand on this idea of why an IDE generally makes developers much more productive. I'm going to give you a few reasons why I think that's the case.

First of all, the entire project, all of the files and all of the different languages are brought together in an IDE like PyCharm. We could be working on a web application. It could have a JavaScript file. In that JavaScript file there might be some functions or even classes that we're going to use in a script block from our Chameleon HTML templates. PyCharm knows about all those different things. It can give you code completion and even re-factoring help across all these different languages, across hundreds of Python files, and even across different languages.

The ability to understand the whole project, not just a single file and as more than just keywords, but actually as an abstract syntax tree is very powerful. You can navigate between code, different files, and different languages. This enables high developer productivity if you can navigate and understand code really well. If you type a variable name followed by a period, PyCharm will present a list of all of the things that that object can do.

You'll also have smaller files rather than one monolithic Python file which means fewer merge conflicts. Files can be for code that does one thing and not a jumbled collection. PyCharm allows you to easily navigate through your code across these files.

PyCharm has the concept of run configuration. If, for example, a project is for a website it can have a configuration that will run the Pyramid web application powering that site. It also might have unit tests. PyCharm can run those as well and report on code coverage.

Because PyCharm understands the whole project, it can refactor across all of the files including things like docstrings and comments. If you rename a function and there's a comment that talks about the function name, the code comment will be updated as well.

PyCharm understands the code and structure of your project and it gives you better code completion than other tools. PyCharm has a good understanding of virtual environments. We can create many different virtual environments and tie these to our run configurations.

There is an excellent package management UI atop pip. It shows you all the various versions of packages you have installed in the active environment and it shows you whether there is an update for it. The UI even lets you search for packages as well.

One of the great things PyCharm has is **code intentions**. A code intention is a little light bulb that'll come up saying "You know, you called `random.choice`, but you didn't `import random` at the top, so your code is going to fail, but don't worry, I know that if I put `import random` at the top, then your code is going to run, so hit a keystroke and I'll fix it for you."

Another example is "You're trying to call this function on a module that doesn't exist. Would you like me to write the function for you, and then you go fill out the details?"

Alternatively, "I see that you are using an external library however, it's not listed in your `requirements.txt`, shall I add it?"

All these neat little "code intentions" are fantastic even if you know what you're doing. The fact that you don't have to spend time thinking about it means you can stay in the flow and keep on writing.

There is excellent source code integration. Git, GitHub, and various other source control systems are integrated. This is great if you don't want to leave your environment just to do source control. Alternatively, if you just prefer a visual environment, PyCharm has source control in the editor.

The database tooling is super cool in PyCharm. It's straightforward to understand various relational databases. If you're even working with something like SQLAlchemy, there are great tools to help you understand, explore, and change the database once the schema is created.

PyCharm has package support. This is the ability to create new packages and create the setup infrastructure or scaffolding. You can then install and register your package on PyPI.

Finally, the unit testing capabilities are excellent. If you're doing unit tests or even integration tests, you can write either pytest or a regular unittest. PyCharm can run them and give you code detailed coverage reports.

There is a lot more than mentioned above and we will cover it much of it in this book.

1.4 IDEs are crazy fast

Before we get into the details of PyCharm, let's dispel what I consider to be a myth—that IDEs are slow. Yes, Emacs definitely starts faster than PyCharm. However, what are we optimizing for? What is the most important thing when writing software? How fast your program comes to life, or how fast you get your work done? I would say that working with an IDE as you'll see, actually lets you work much faster.

While it might take a second longer to start, PyCharm is going to let you work a lot faster throughout the day. You're going to start up PyCharm and leave it going all day anyway, so once it's up and running, it doesn't go any slower. On a modern machine PyCharm might take 5 seconds to start.

Another concern is power usage. Because PyCharm is doing a lot of analysis, this could limit your ability to create the next internet sensation on a plane flight or if you're down to your last 10% percent of battery. PyCharm has a fix for this. It's called **power mode**.

This power mode lets you turn off some of the real-time analysis and code completion type of operations, letting PyCharm use less energy.

1.5 PyCharm is more than just Python

Something that is not immediately obvious (in fact, I misunderstood this when I first got started) is how the building blocks are all put together to make PyCharm. PyCharm is obviously a Python editor. However, as I've hinted, it also does other things like the database tooling or editing other languages. There are a bunch of other JetBrains tools combined to make up PyCharm.

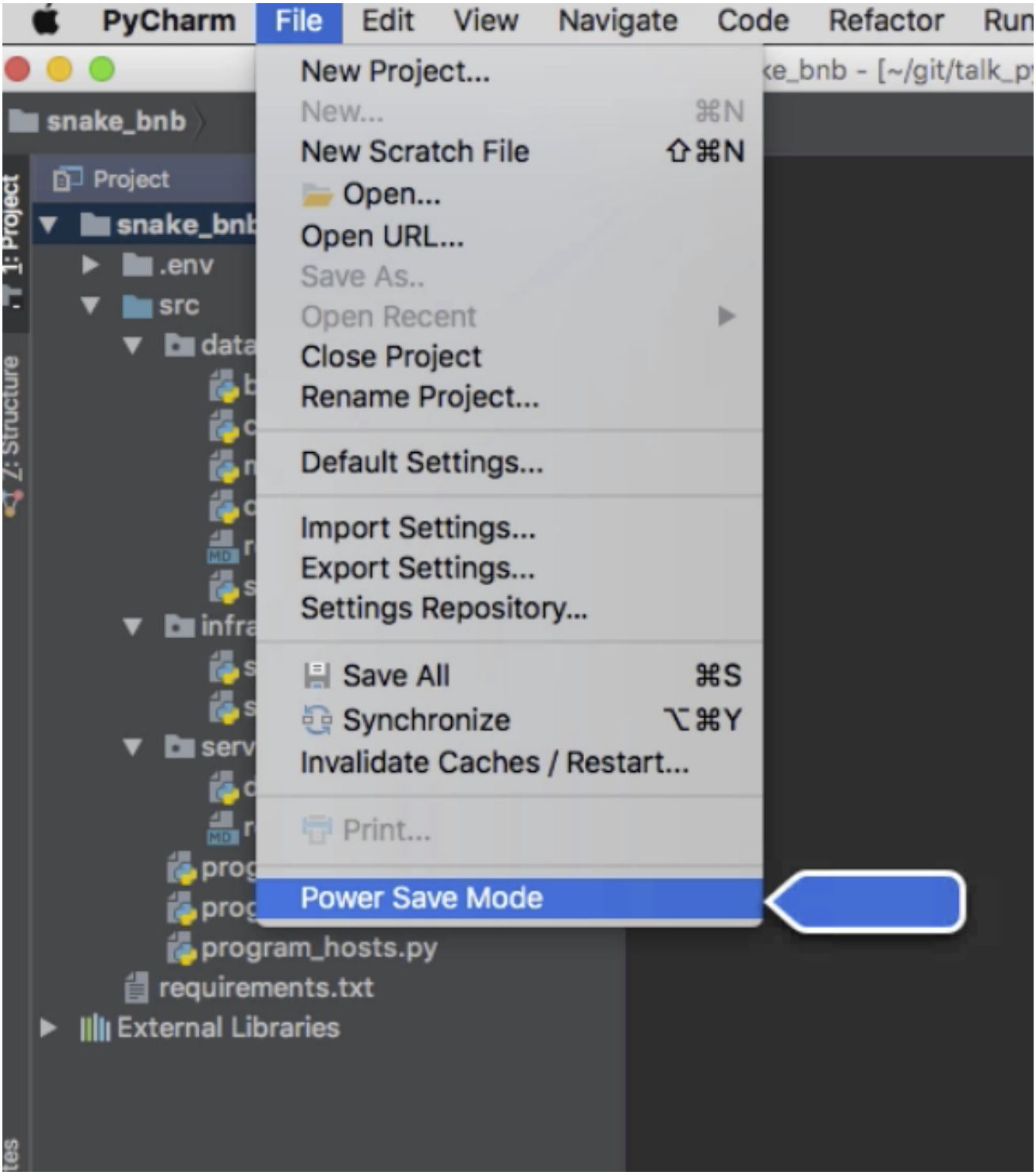


Figure 11: You can turn on *Power Save Mode* for a long flight or if you are out of battery at the end of the day.

JetBrains produces many different IDEs: PyCharm, WebStorm (which is for JavaScript and HTML), AppCode (which is for building IOS apps), DataGrip (for editing databases) and so on. However, all of these IDEs are built upon a common platform called IntelliJ.

IntelliJ initially was just a Java IDE, but it's very extensible and configurable. It has been pulled apart so it can be the foundation of all these different tools. Most things that IntelliJ does, PyCharm or these other editors can also do.

PyCharm professional is a paid version that you should consider if you need broader tooling support. If you are working with web applications, JavaScript, or databases, the professional version has the ability to interface with all of these.

WebStorm is an awesome JavaScript and TypeScript editor. If you're doing AngularJS, Ionic, Cordova, or something like that, it's a great place to work. It understands JavaScript, CSS, HTML super well. In fact, the pro version of PyCharm has all the features of WebStorm in it.

PyCharm also has the features of a tool called DataGrip. DataGrip is a database IDE from JetBrains, and all the database functionality within PyCharm comes from there.

When you think of PyCharm, you should think of PyCharm the Python editor, plus WebStorm, plus DataGrip. This is not a single tool but a holistic, full stack toolset that you can use for anything to do with web development, database work, or even data science.

1.6 Cross-platform

One of the nice features of PyCharm is that you can use it on any operating system. If you use Windows, Mac, or some form of Linux, you can use PyCharm, and it will behave in more or less the same way. (The key bindings might be slightly different-using command on Mac versus control on Windows or Linux).

If you're working in teams, where different people have different platforms, or you want to use it on different machines, you don't have to worry. PyCharm very much follows the Python ethos of running everywhere.

1.7 Versions: Pro and community

If we go over to the PyCharm website and we hit download, you are presented with a choice. Do you want the professional version which you have to pay for or do you want the 100% free and open source community edition?

It's hard to know. The community edition is free, but there might be bonuses in the professional we might want. How do we know?

Some of the things we cover in this book only exist in the Professional version. For example, WebStorm and DataGrip are in the professional version. JetBrains has a page describing the difference .[1]

The free version is pretty full featured if you are only working with Python. There's the code intentions mentioned previously, refactoring, code inspection, source control, and the debugger.

If you are doing web development and need WebStorm capabilities or support for the various Python frameworks (Flask or Pyramid or Django), you should consider the full version. You can edit templates and have access to tools like remote debugging capabilities, and DataGrip.

If features are only available in the professional version, they will be pointed out in the book.

[1] https://jetbrains.com/pycharm/features/editions_comparison_matrix.html

1.8 PyCharm is open source

The Community edition of PyCharm is 100% open source. The JetBrains organization in GitHub[2] has the community editions. If for some reason PyCharm being open source is something that you really care about, you can check it out and build it.

I've never tried, I have no interest in building PyCharm. Maybe I'd build a plugin, but I wouldn't build the whole thing from scratch. Nonetheless, it's nice to know that the code is there in case you want to check it out or modify it.

1.9 Python runtimes

If you're new to Python, you might not realize there are many different versions and flavors of Python. If you just say Python, you probably mean something called "CPython" (that's what you get if you go to python.org and download Python for your OS). However, there are a bunch of different runtimes, sometimes called interpreters, for Python. (I don't like the term interpreters, because not all of them are interpreters, some of them are JIT compilers.)

One of these is PyPy which is a JIT compiler for the Python runtime. In some circumstances, PyPy can be much faster than CPython, but it doesn't have as much coverage for all of the libraries that you might use. Many of the C-based extensions are not supported in PyPy.

Cython is another option. Cython is a tool to take Python-like code and compile it down the machine instructions. This is not even JIT compiled, but compiled like C. It uses a Python-like language where types can be provided to give it a speed boost. PyCharm's code completion engine supports Cython.

Then we have the two plugin flavors of Python: Jython and IronPython. They plug into Java and .NET respectively.

Regardless of how you want to run your Python code, all of these Python variants are supported in PyCharm.

1.10 How do you learn all the features of an IDE?

If you're new to PyCharm, an IDE can be overwhelming. You can see in the figure that there are many windows and many things happening in those windows. Obscured in the middle, there is an editor, project information, database tools, and configuration. There are many menus with lots of options, and you can right click on almost anything to bring up more options.

There's just so many things to learn to be effective with PyCharm and IDEs. These tools do tons of stuff and it can be really hard to remember all the features and operations that you can do with them. If you don't use them to full advantage, then they're not nearly as valuable to you as a simple editor.

You should commit to learning all of these little features (or at least the 80% that you use most of the time). But how do you do it? How do you go through and you learn all of these things?

I'm going to give you a concrete set of techniques that more or less I use for myself, and you can use as well. It starts with discovering what the features are. We are going to present them in this book. Outside of this book, you need to discover even more features and the hotkeys.

[2] https://blog.jetbrains.com/pycharm/2013/10/pycharm-3-0-community-edition-source-code-now-available/

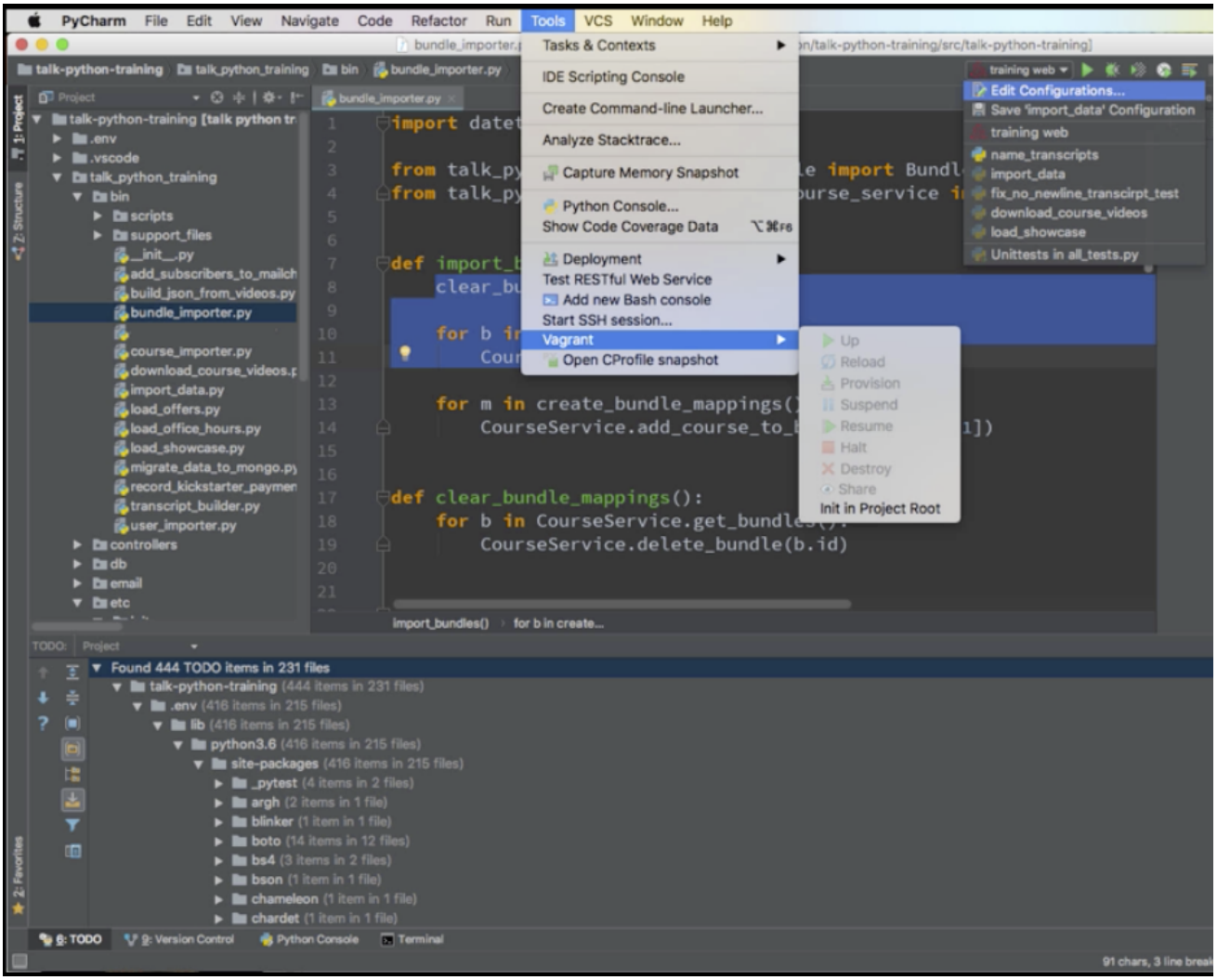


Figure 12: Figure showing many (perhaps overwhelming) features of PyCharm

More importantly, and perhaps more challenging, you need to actually recall the features. You want to keep having this experience: "Oh yeah, there was this feature, and it was awesome and here's the hotkey." How do you do that?

I'm going to propose a couple of things you can try, adapt them to your learning style.

First, reading about PyCharm is only going to get you so far. It will introduce many of the features, but if you never try them out you will probably forget them. Make sure you are not just reading this book, but have a computer nearby where you can try the features of PyCharm.

Next, there is a cheat sheet from JetBrains.[3] It has the hotkeys for the most common features. Print this out and keep it handy.

Learning the hotkeys are a chore, but they pay off. You will be much more productive if you don't need to reach for a mouse.

The final learning hack is to use sticky notes. Keep a stack of them handy while reading the book. As a feature is mentioned, jot the feature and the hotkey from the cheatsheet on the sticky note.

[3]https://blog.jetbrains.com/pycharm/files/2010/07/PyCharm_Reference_Card.pdf

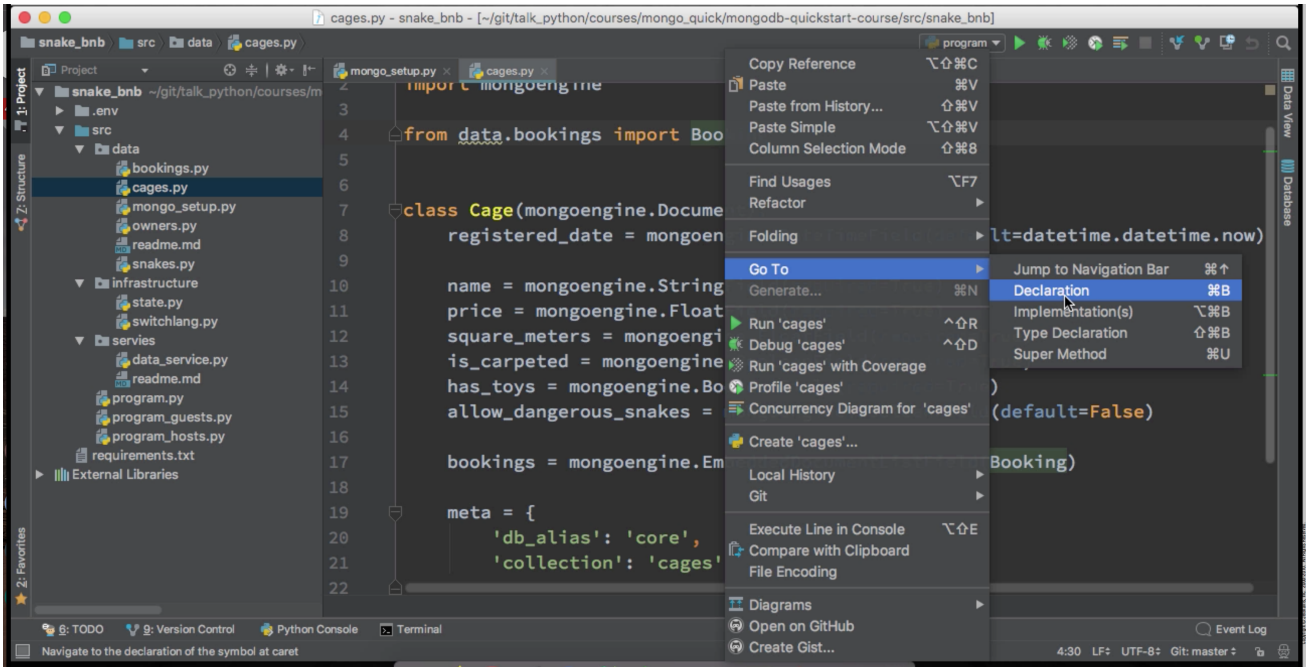


Figure 13: Clicking on the navigate to declaration option when presentation assistant is installed.

You should have a stack of sticky notes. Every week, you should put one (or more) of those on the corner of your monitor. Make a concerted effort to use that feature and hotkey until it becomes second nature. Try not to use the mouse if you don't have to. Once you have it dialed in, put that sticky note away.

It takes a bit of practice. If you do it for a few days, the hotkeys will stick and you will be more productive.

One final hint. There is a presentation assistant. If this plugin is enabled and you click on a command (button or menu item) with your mouse, it will flash the hotkey for a few seconds at the bottom of the screen.

To install the Presentation Assistant, go to the Preferences page, and type in "plugin". From there click on "Browse repositories...". In the search box type "Presentation Assistant" and click on the Install button.

An alternative option is the Key Promoter X[4] plugin. This plugin will show you the keyboard shortcut for commands when you use the mouse. Search for "Key Promoter X" to install the plugin.

1.11 Get the source code on Github

One final thing—head over to GitHub and star the materials for the book[5].

There are hands-on exercises that you can work through to reinforce each chapter. Reading about PyCharm is one thing. Your mastery of PyCharm will go much deeper if you are able to apply what you are learning.

[4]https://plugins.jetbrains.com/plugin/9792-key-promoter-x

[5]https://github.com/talkpython/mastering-pycharm-course

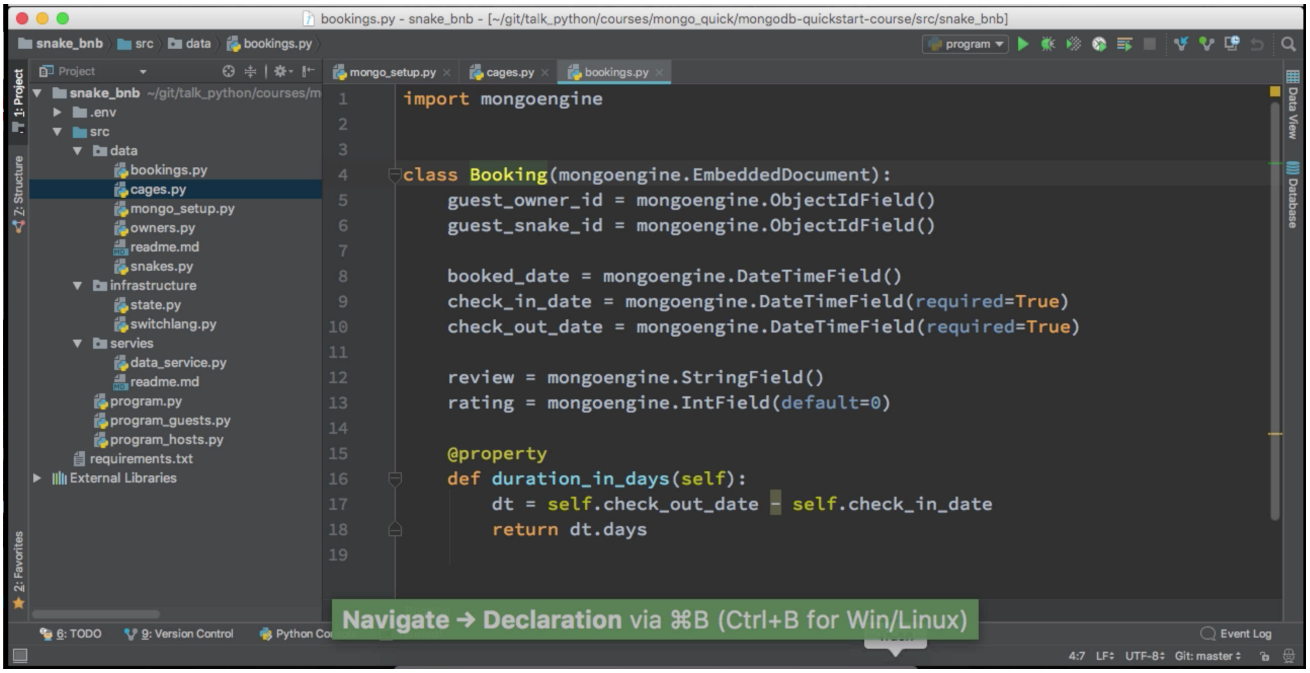


Figure 14: Presentation assistant shows the hotkey for a command.

1.12 Summary

PyCharm is the most powerful IDE for Python. It runs on the most common operating systems, Mac, Windows, and Linux. The open source version allows you to create and debug Python programs. If you are working on web projects or database projects, the professional version has compelling features to aid you.

Learning a feature-packed tool like PyCharm can be an intimidating task. We recommend that you print out the cheat sheet, write shortcuts on sticky notes, use the presentation assistant, and practice using PyCharm with the exercises.

Chapter 2

Machine Setup

2.1 PyCharm Community or PyCharm Professional?

When following these instructions, you're offered the choice of using PyCharm Professional (paid) or Community (free) additions. You can choose the free edition but do be aware that roughly 30% of the topics covered in the book are only available in the Professional edition. The good news is that the free edition covers the remaining 70%! If you want to see which features land where see their comparison matrix[6]. We recommend getting the professional edition as it has powerful tools that make our lives as developers easier. Use the free trial if you want to test it out.

Note

PyCharm Professional is free for open source developers and for education uses.

2.2 macOS Setup

Let's talk about setting up PyCharm on your Mac. First, you want to make sure you have a recent version of Python 3. Open a terminal (press CMD+SPACE and type terminal) and enter the command: python3 –version. You should see something like this if Python is installed:

$ python –version Python 3.7.1

I have Python 3.7.1, which will work great for our examples in this book. If instead, you see this:

$ python3 –version -bash: python3: command not found

You can certainly use PyCharm with Python 2 which macOS bundles, but we will cover some of the Python 3 only features. If you want access to them (and you do), install Python 3.

You have a couple of options for installing Python 3 on macOS. You can go to http://python.org and download the latest package there (be sure to choose Python 3). Another option is to use Homebrew[7], a package manager for Mac similar to Linux. Homebrew is great for installing and

[6] https://www.jetbrains.com/pycharm/features/editions_comparison_matrix.html

updated all sorts of libraries and applications such as Node, OpenSSL, MongoDB, and of course Python.

To install Homebrew, copy the command from the webpage and execute it in a terminal:

```
$ /usr/bin/ruby -e "$(curl -fsSL
  https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

The script explains what it will do and then pauses before it does it. There are more installation options (required for OS X 10.8 Mountain Lion and below). You will need the command line tools for XCode installed (see the details[8]).

After installation, you should have access to the `brew` command. If you have previously installed Homebrew, be sure to run `brew update` before continuing. Now you can run:

```
$ brew install python
```

It will install a recent version of Python 3 for you. I prefer the Homebrew version because if I have to compile a Python library that wraps a C or Fortran library, Homebrew usually "just works".

Next, we want to install PyCharm. Head over to the download page[9] and download the version you prefer.

After installing PyCharm, you can launch it by double clicking on the icon in your Applications folder or typing command-space and then PyCharm. We recommend keeping it in your dock for easy access.

2.3 Windows Setup

If you are using Windows, the first thing you're going to need to do is make sure that you have Python installed. Windows doesn't come with any version of Python. You will need to install it. To check if you have Python installed launch the Command Prompt. You can do that by clicking in the Search Box in the lower left of the screen and typing `cmd`. After the Command Prompt launches, type:

```
C:\> python --version
```

This command will report on the Python version you have or complain that `python is not recognized`. In that case, go to http://python.org and download the latest package for Python 3. Make sure you click on the "Update PATH" checkbox on the first page of the install wizard so that you have access to `python` from the Command Prompt.

Next, we want to install PyCharm. Head over to the download page[10] and download the version you prefer.

You should now be able to run PyCharm from your start menu. We recommend pinning it to your taskbar for easy access.

[7] https://brew.sh

[8] https://docs.brew.sh/Installation

[9] https://www.jetbrains.com/pycharm/download/

[10] https://www.jetbrains.com/pycharm/download/

2.4 Linux Setup

Most Linux machines come with Python installed. Check it Python 3 is installed by opening a terminal and typing:

```
$ python3 --version
Python 3.6.6
```

If it has a recent version, you're good. Otherwise, you can use your package manager to install a newer version.

Next, we want to install PyCharm. Head over to the download page[11] and download the version you prefer.

Then you can pin it to your favorites, so it's easy to access while going through the book.

[11] https://www.jetbrains.com/pycharm/download/

Chapter 3

PyCharm Projects

3.1 Project Introduction

We have talked about PyCharm, some features, why you want it, and how to install it. Now it's time to start using PyCharm to do some cool stuff. We're going to start by focusing on projects. PyCharm is organized around projects - a set of files and the relationships between them. Some features made possible by these projects include navigating through the file relationships, jumping to definitions across multiple modules, and refactoring methods. If you are working with a web application, you can edit static files and get auto-completion for them.

We're going to focus on the critical parts of a project and show how to put a bunch of files together as a single application, navigate through them, and run them.

3.2 Creating Projects

Let's start by creating a new project. We are going to use an existing project. Type:

```
$ git clone https://github.com/talkpython/mastering-pycharm-course.git
```

We could clone this with PyCharm, but you'll see there's one piece that might get missed from the application.

Launch PyCharm and click "Create New Project". We will create a "Pure Python" project. In the location area use a path like `first_project`. It also asks you to select a Python interpreter. PyCharm should find the recent version of Python we installed, if not point the "Interpreter" selection to it.

If you created a project in a folder already controlled by version control like git, PyCharm shows a popup asking if you want to add the project to version control. Most of the time, choosing to register the VCS root is a good idea. It lights up a ton of features around files and source control inside PyCharm.

On the left-hand side of PyCharm is the project area. You can see your folder, the Python environment you selected when creating the project, and the external libraries.

If you right-click on the project, you can add a new file by clicking on New > "Python File".

PyCharm asks you for the name of the file. Call the file `hello`. Again, if you are in a version-controlled directory, it will ask if you want to add that file. In that file put the code:

```
print("hello world")
```

Create another file named `otherfile.py`. In that file add:

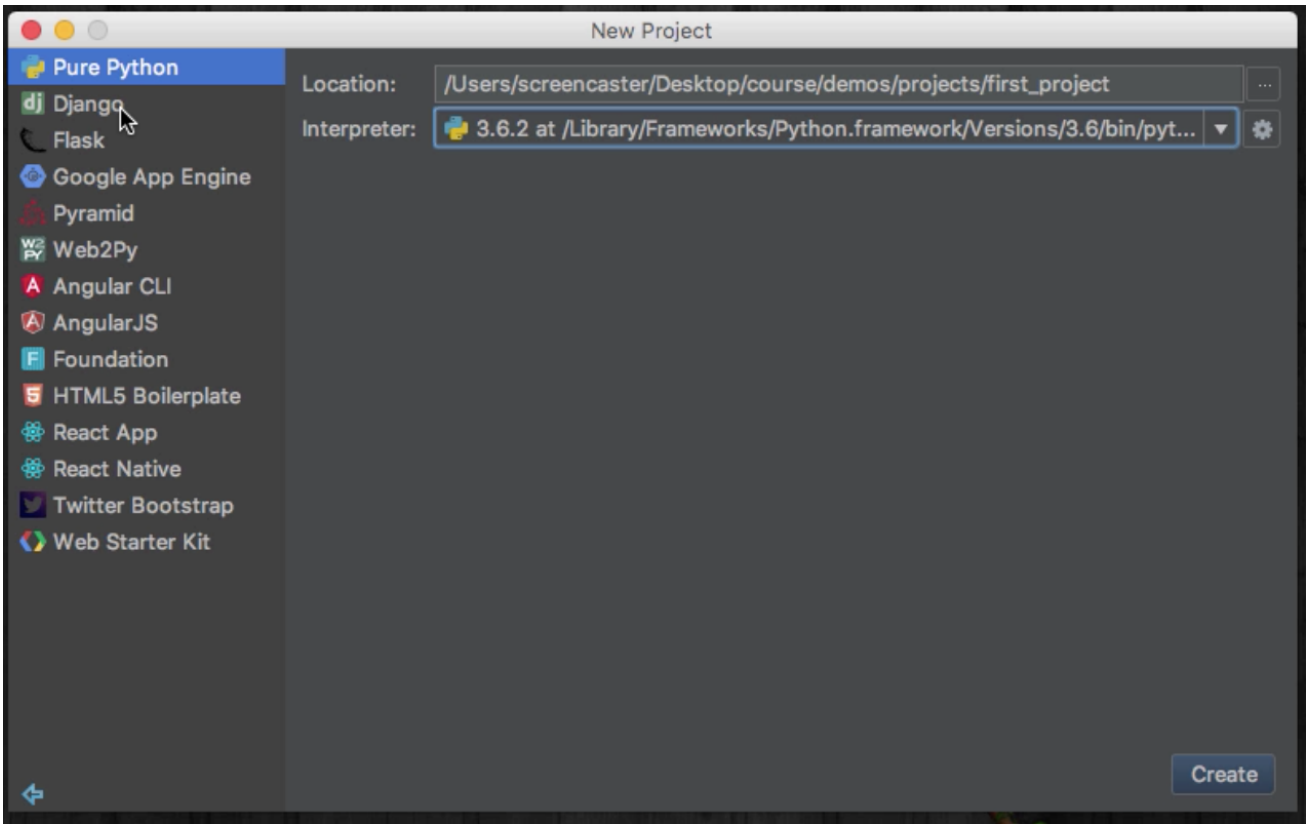


Figure 31: Figure illustrating location and interpreter for the first project.

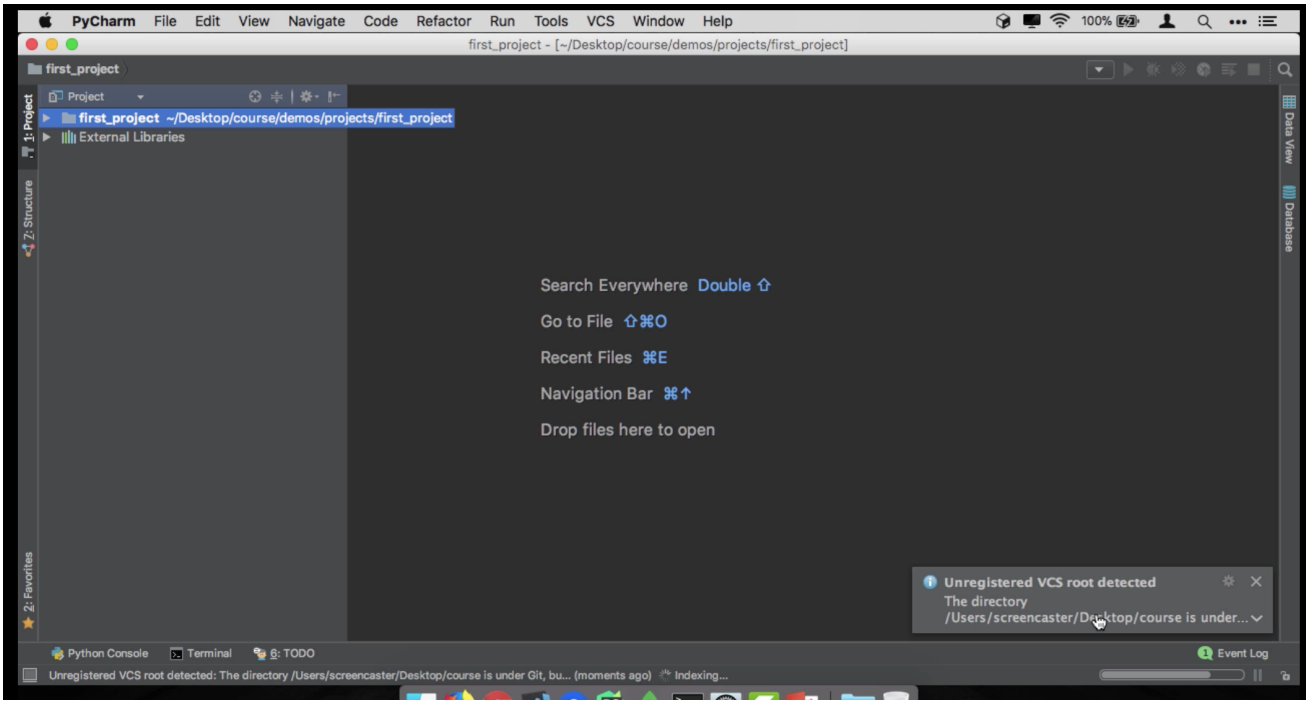


Figure 32: Figure illustrating a new empty project. Notice VCS popup in lower right

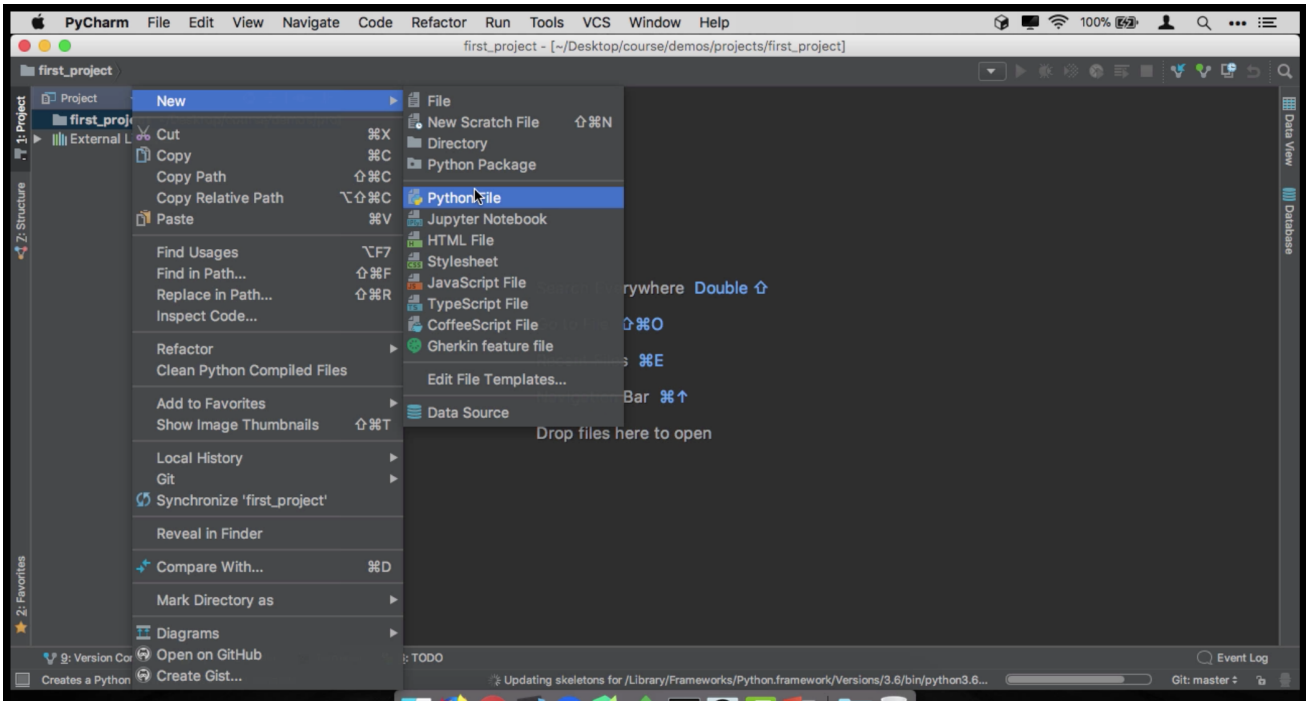


Figure 33: Figure illustrating creating a new Python file

```
def other_method():
    print("This is the other method")
```

To run a file, you need to right-click on the `hello.py` file in the project window and click "Run 'hello'".

Running a file in this manner will generate a *run configuration*. If you look in the upper right-hand side of PyCharm, you will see a drop-down pointing to the "hello" configuration. There are buttons for running, debugging, reporting on code coverage, and profiling the configuration.

Those are the basics are creating a project from scratch. You point PyCharm at a folder, choose a Python interpreter, and setup run configurations for the top-level files that you add.

3.3 Working with Existing Projects

The majority of the time you will be opening existing projects. After all, we work on projects (packages, apps, and websites) for years and we rarely create new ones. Let's go and take an existing project and open it in PyCharm.

This time, let's clone the Talk Python Jumpstart course's repo:

```
$ git clone https://github.com/mikeckennedy/python-jumpstart-course-demos
```

In this repository is an `apps` folder with sorts of different applications or projects. You can have a mega project with sub-projects in it or a bunch of smaller projects based on the various directories here. We'll go for the "mega project". Before we do thought, let's also create a virtual environment for the project. A virtual environment is an isolated Python installation where you can install your packages and 3rd party dependencies into.

There are two main ways to create a virtual environment. One is from the terminal. This is nice because it works everywhere and is not dependent on any tool. It is described below. The other option is to use PyCharm to create the environment.

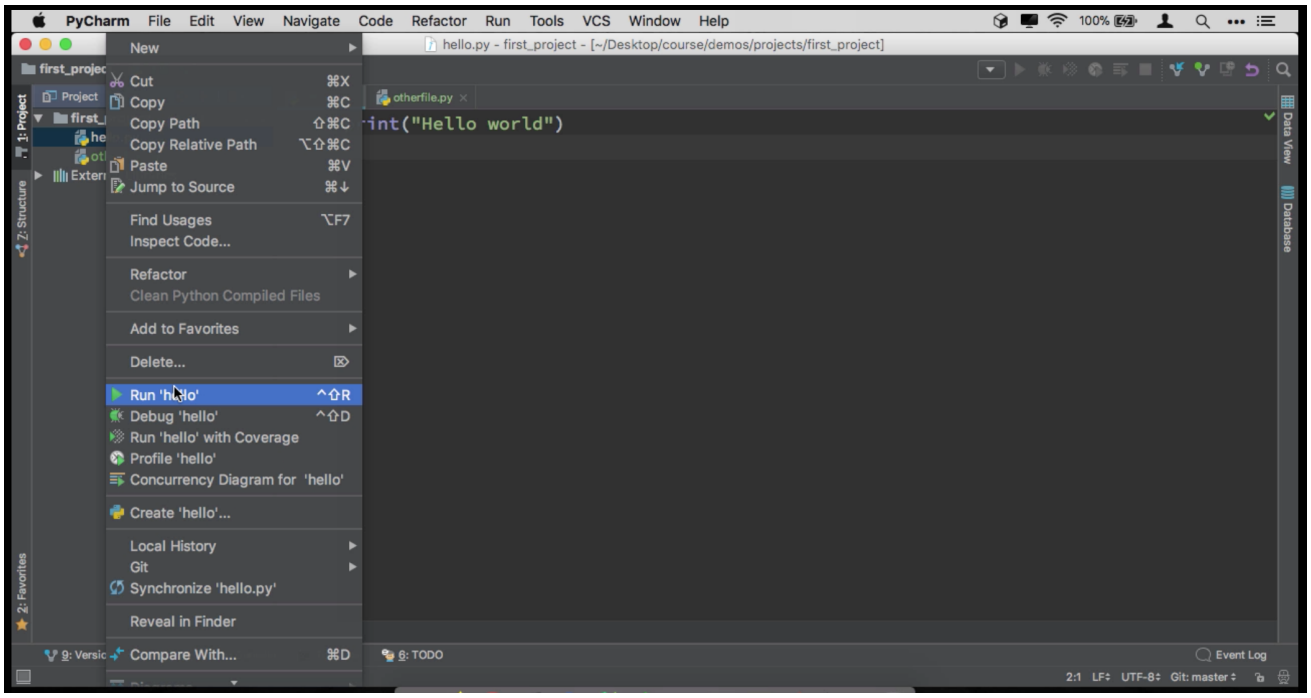


Figure 34: Figure illustrating configuring a file to run

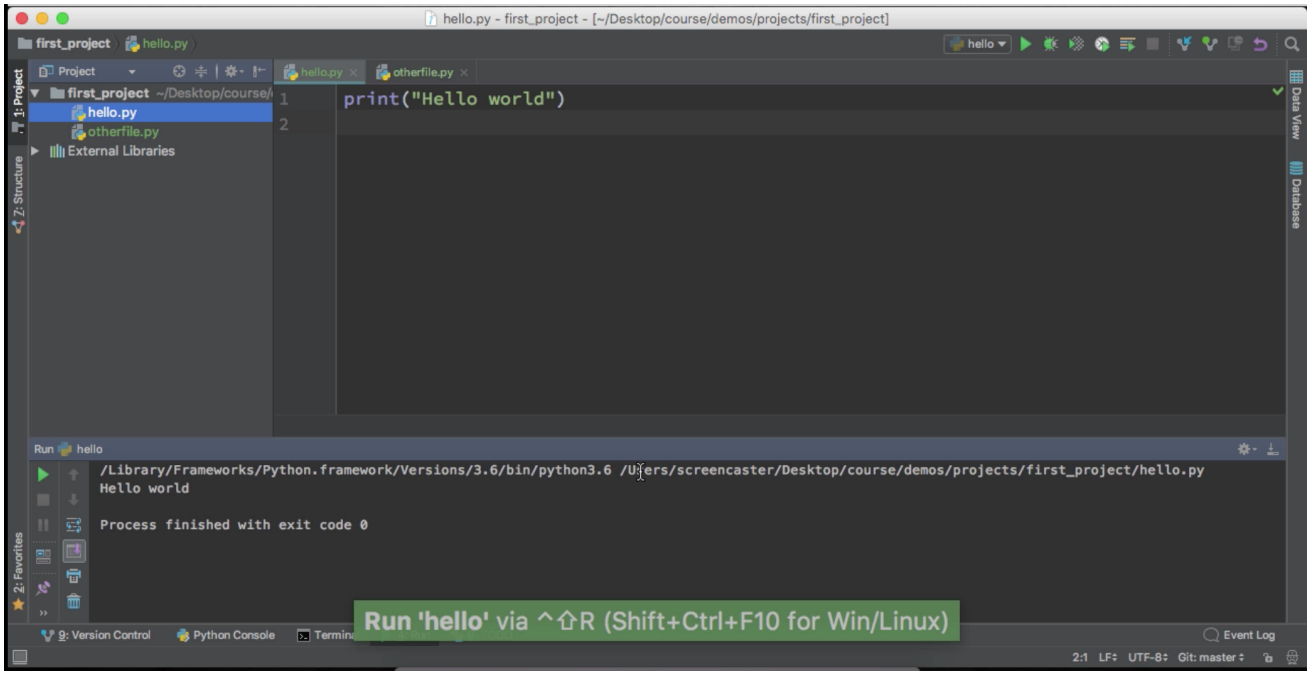


Figure 35: Figure illustrating running a file. Output is shown in the lower pane, and the Presentation Assistant shows the hotkey (shift command R on Mac).

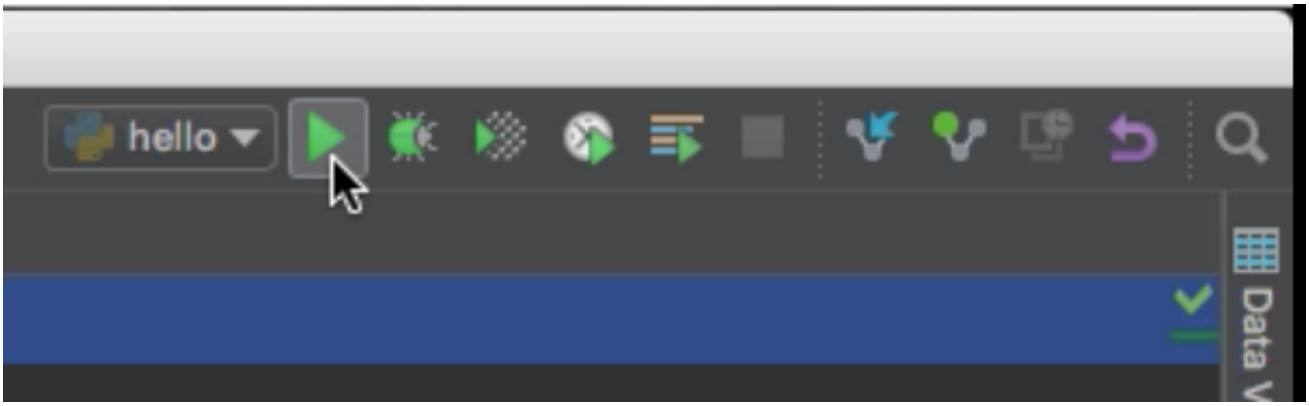


Figure 36: Figure illustrating buttons for running an existing configuration. There are buttons for running, debugging, reporting on code coverage, and profiling the configuration.

To create an environment from a terminal, do the following. In a terminal / command prompt, change into the folder that you just checked out (i.e. `python-jumpstart-course-demos`). We'll create a virtual environment with the command:

```
python3 -m venv .env
```

Note

On commands that work on all platforms (Mac, Windows, and Linux), we will not precede them with a `$` (Unix prompt for Mac or Linux) or a `C:\>` (Windows prompt). When we invoke Python for this platform neutral command, we will specify `python3` although the command on Windows will be `python`. On Windows, because Python does not support the `python3` (vs `python`) command, you'll need to be sure that `python.exe` for Python 3 is first in your path. An easy way to test this is to type `where python` in a command prompt.:

```
$ python3 -m venv .env
```

And the Windows version would be:

```
C:\> python -m venv .env
```

To create a virtual environment with PyCharm, open the *Preferences* window and search for "Project Interpreter". If you click the gear next to the default interpreter, you can click "Add...". From the "Add Python Interpreter" window, you can create a "New environment" or use an "Existing environment". If you have multiple versions of Python installed on your machine, make sure that you have the appropriate "Base interpreter" selected. Note that PyCharm puts environments in the `.virtualenvs` directory in your home folder.

PyCharm lets you open up the `python-jumpstart-course-demos` folder to create this mega project. Alternatively, in Mac, if you have PyCharm in your dock, you can drag the folder over the application icon. Because we have a `.env` folder with a virtual environment, PyCharm recognizes it, and uses it as the default run configuration. The names PyCharm recognizes as virtual environments include `venv` and `.env`.

Let's talk about a concept in PyCharm called *Sources Root*. Because this project has multiple subprojects in it, PyCharm will assume that Python package for the project will be in the root directory. If we navigate to the `apps/07_wizard_battle/final` folder, there are two files: `actors.py` and `program.py`. If you open `program.py`, you will see that it references code found in

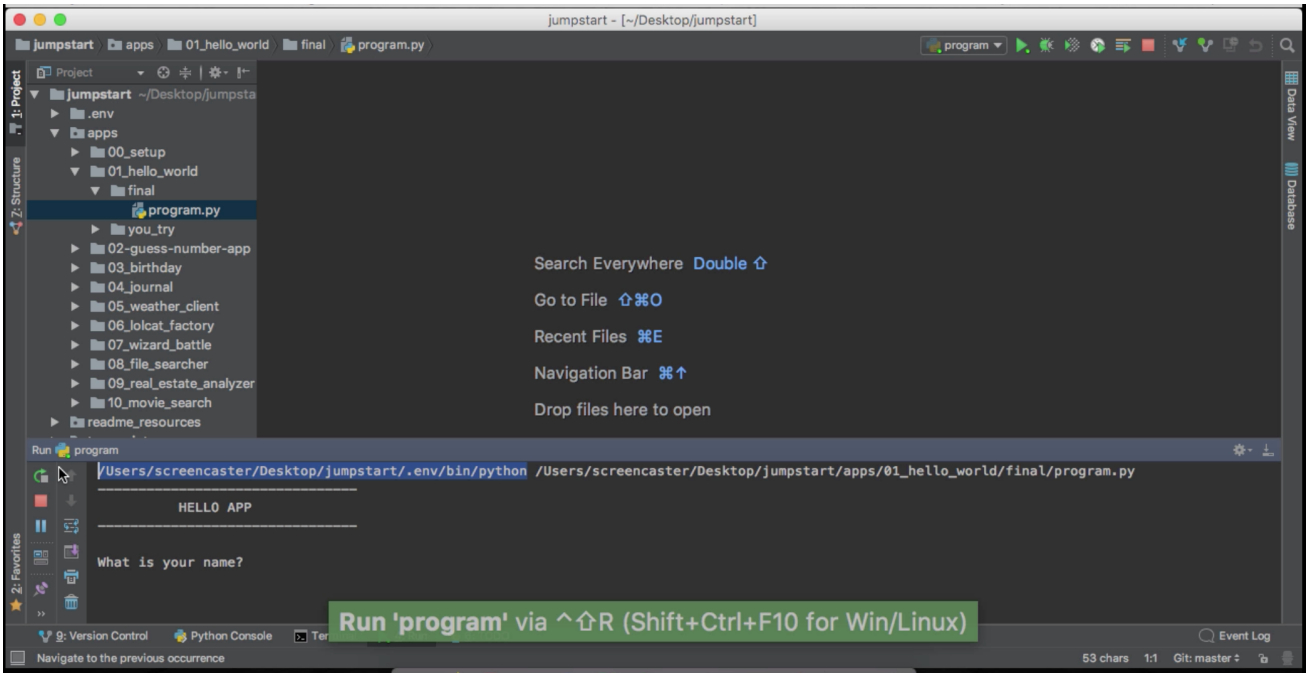


Figure 37: Figure illustrating running a file in a folder with an existing virtual environment (`.env` folder).

`actors.py` and that it has little red squiggly lines under those references. This is because PyCharm is looking in the top-level directory for `actors.py` (and the virtual environment) and not finding it. However, if we run `program.py` from PyCharm, it happily runs. When the program is executed, Python adds the directory containing `program.py` into the list of paths that it uses to find libraries.

To tell PyCharm to recognize the `final` folder as a folder containing source code that is standalone relative to the rest of the project, we can right-click on it and choose "Mark Directory as", then "Sources Root". This lets PyCharm use its smarts and enables features like autocomplete.

Similarly, if you are working on a web application and have a structure like `apps/static/site.css`, PyCharm does not associate `/static/site.css` to URLs in this folder. But you can fix this as well. Right-click and choose "Mark Directory as", then click "Resources Root" (as in website resources). PyCharm also lets you make template folders (for Django, Pyramid, and Flask). You can even tell PyCharm to ignore folders and not analyze or index them. This can improve indexing speed as well as omit that content from autocompletion.

In conclusion, using an existing project in PyCharm requires opening the folder. If there is a virtual environment in that folder, PyCharm will automatically detect it and use that environment to execute the files (only on the initial project creation). We made need to mark folders as source, resource, or template folders to tell PyCharm how to index and integrate them. To run a file, right-click on them and choose "Run" or explicitly create a run configuration.

3.4 Python Environments

PyCharm recognizes virtual environments that are inside of existing projects. Alternatively, you can tell PyCharm to use the system Python (Python 2 on Mac or Linux) or a Python 3 installed for systemwide use. You can even point PyCharm at PyPy, IronPython, or Jython. For this book we use version 3 of Python that comes from python.org (also referred to as CPython as it is implemented in C).

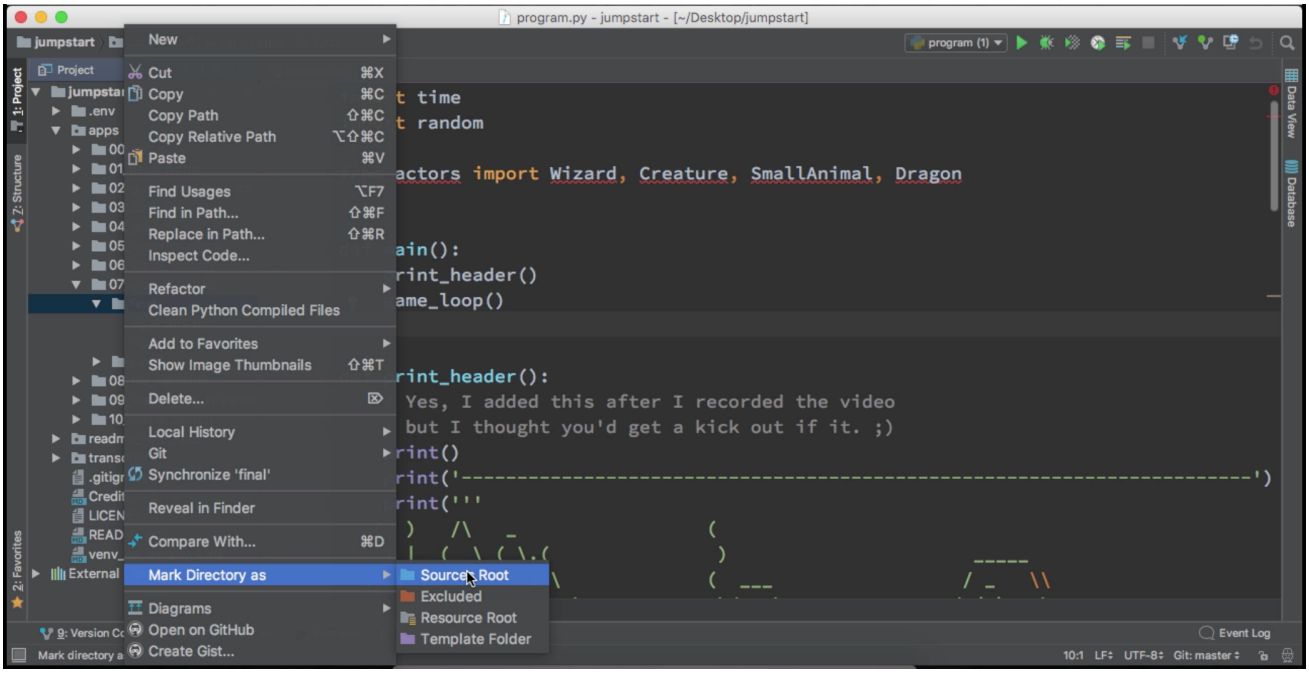


Figure 38: Figure illustrating marking a folder as a source root, so PyCharm knows to index the code in them.

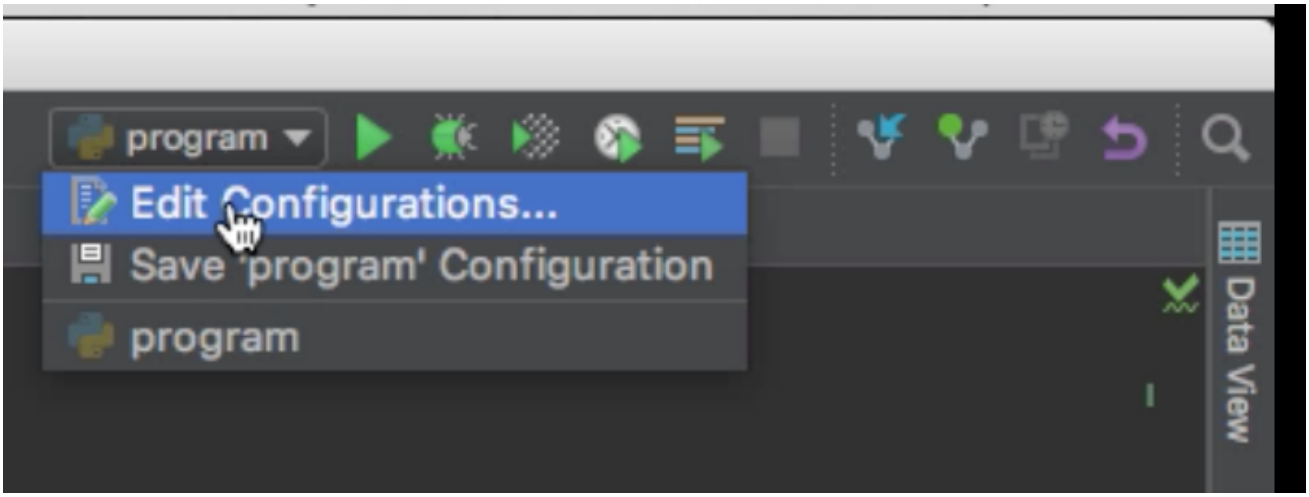


Figure 39: Figure illustrating launching the configuration editor.

If you want to change the environment for a run configuration, you can click on the run configuration button in the top of the editor. It will either say "Add Configuration" or the name the a recently executed file with a dropdown option. Click the dropdown option and click "Edit Configurations...". Note that this will only change the interpreter for the run configuration and not for the project. (If you want to change the interpreter for the project, you need to do it from the Preferences -> "Project Interpreter" page.)

With the "Run/Debug Configurations" window open you can change the "Python interpreter". For basic scripts using the system Python might be sufficient. For any substantial project, we highly recommend that you create a virtual environment and use it to manage dependencies.

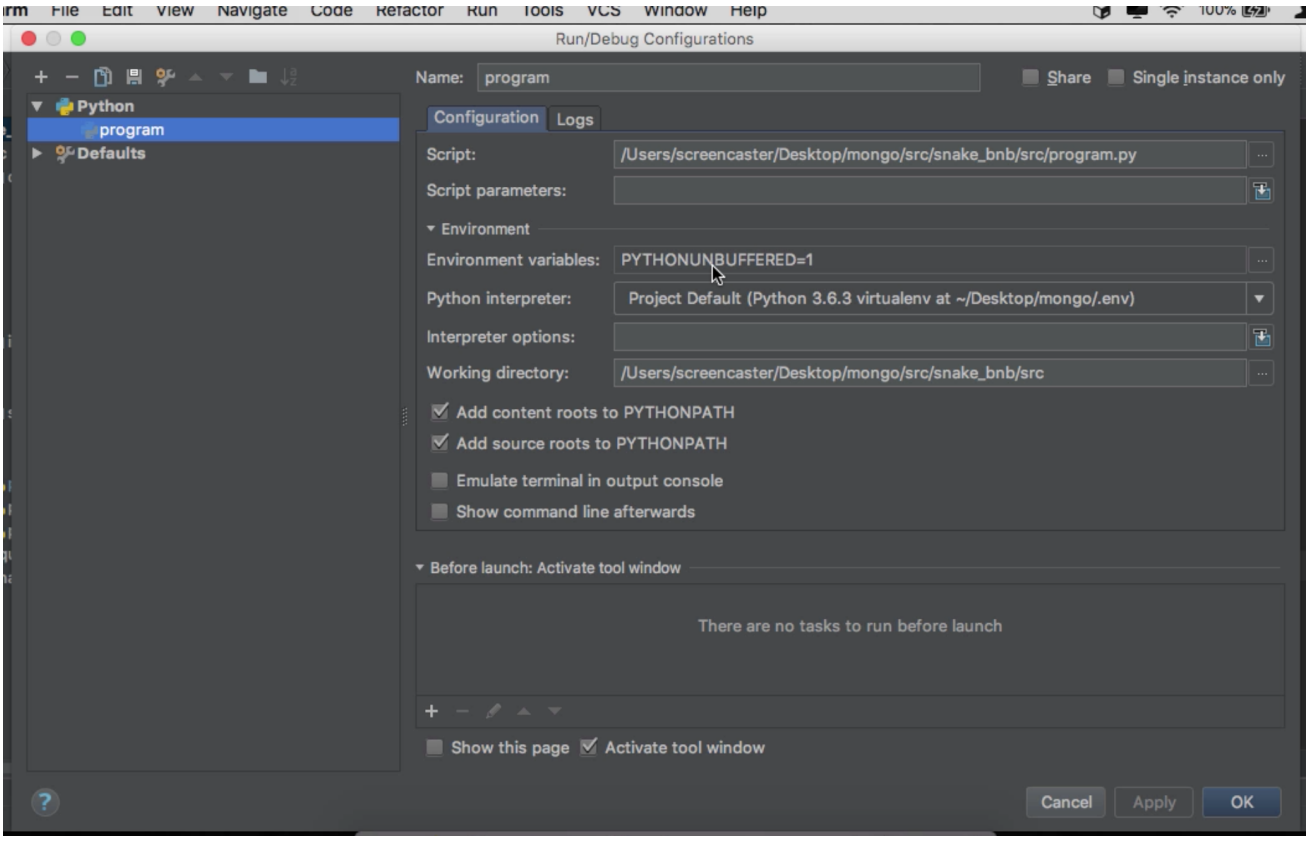


Figure 310: Figure illustrating launching the configuration editor. From here we can select the Python interpreter used for our configuration. Note that each file can have its own configuration.

If you had a virtual environment named `.env` within the checked out repository before adding it as a project in PyCharm, PyCharm recognizes that. If you need to add an environment after the fact, you will need to create the environment and add the environment as a "Project Interpreter". Then you can use the environment from the run configuration window.

You can create an environment using the command line as we have been. But, you can also create these virtual environments from within PyCharm. To do that, open the *"Settings"* (Windows/Linux Ctrl+Alt+S) or *"Preferences"* (Mac) screen (command+",") by hitting the wrench icon. In the search area type "Interpreter".

From this window you can see all of the packages in the project interpreter. You can also add, remove, update, or refresh the packages by clicking on the plus, minus, up arrow, or eye icons respectively. These icons are found at the bottom of the window. If you click on the gear icon on the right, there is an option to "Add..." a new environment. From there you can create a virtual environment using Python's venv command, pipenv, conda (if you have Anaconda installed), and more.

If you are familiar with `pip` and working from the command line, on the main editor window there is a button at the bottom labeled *"Terminal"* (alt-F12) that launches a terminal.

In this window, you can run all the standard pip commands (e.g. `pip install -r requirements.txt` if you have created a requirements file). Whatever interpreter / environment you have set as the project default will be automatically activated in this window.

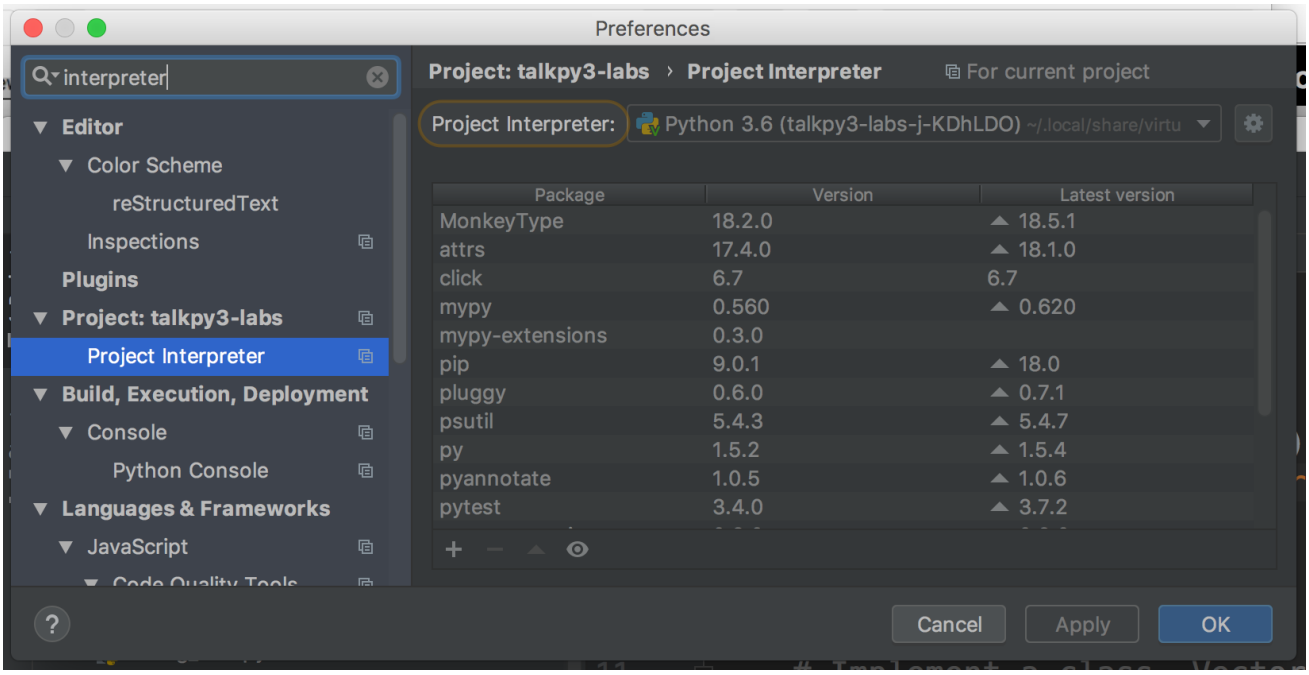


Figure 311: Figure illustrating Interpreter configuration. In this screen we can manage packages in an interpreter. We can also click on the gear to add new interpreters.

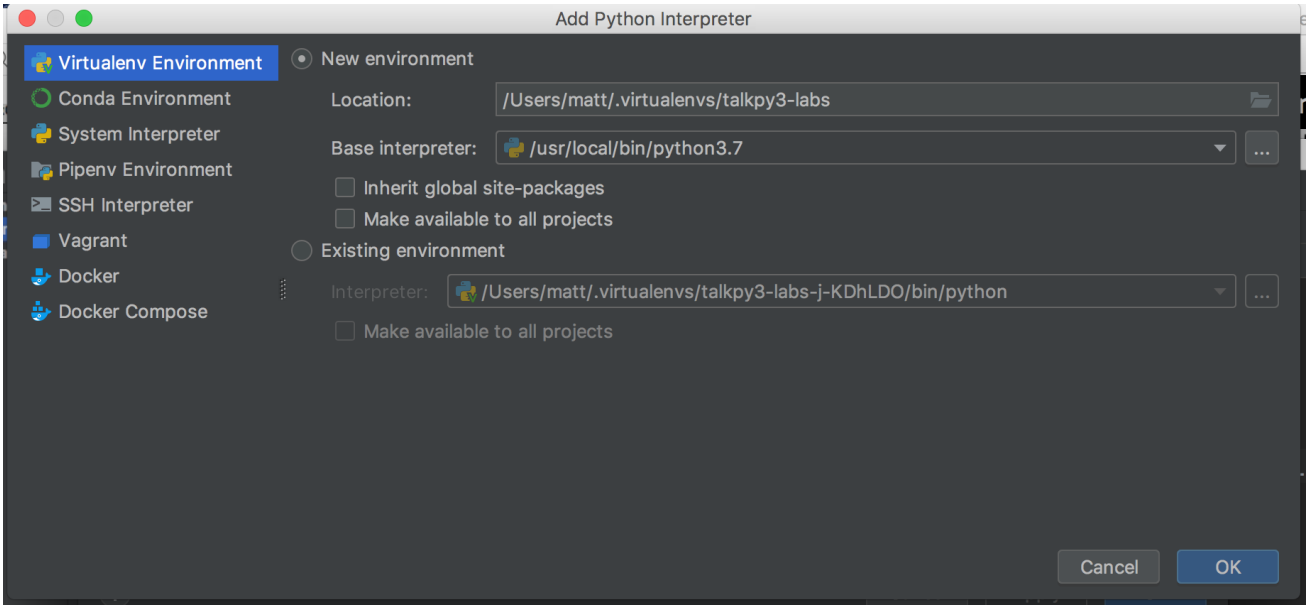


Figure 312: Figure illustrating adding a new environment.

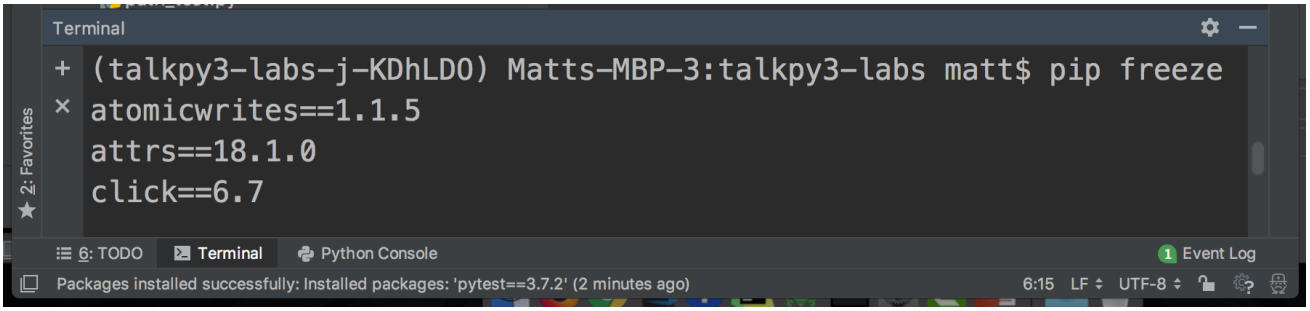


Figure 313: Figure illustrating using the terminal to gain access to `pip` and the virtual environment. Notice the "Terminal" button at the bottom of the window.

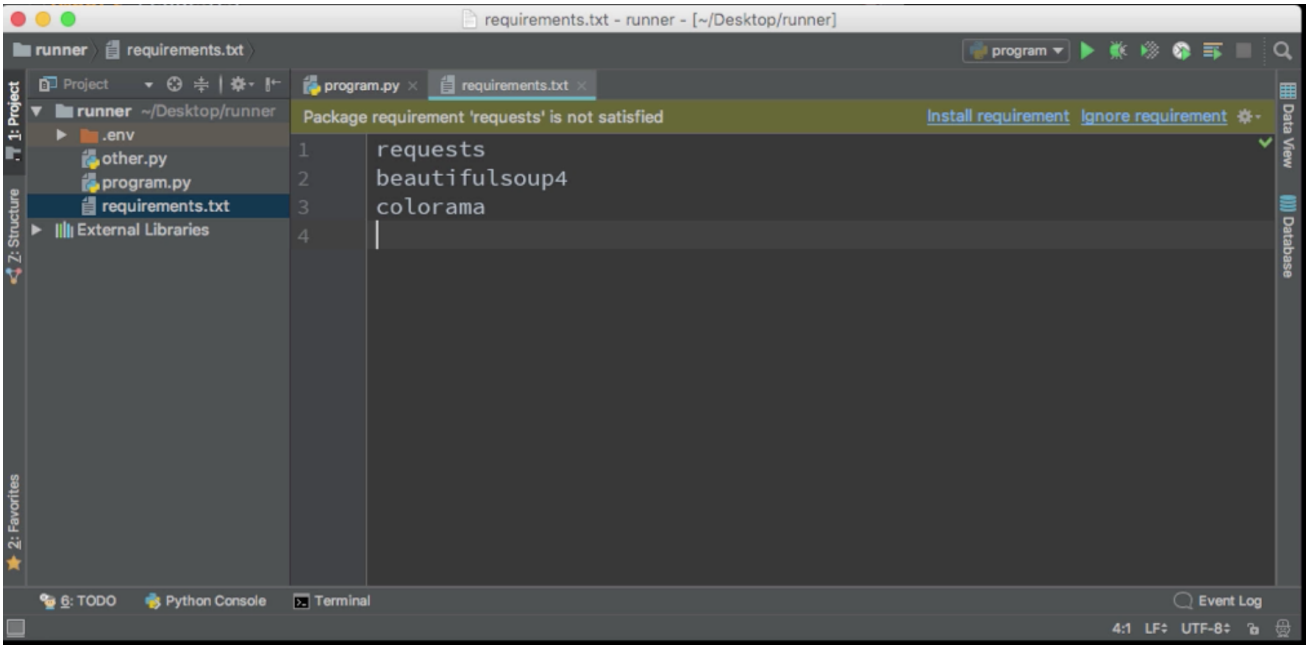


Figure 314: Figure illustrating opening a `requirements.txt` file and PyCharm prompting to install a missing dependency

3.5 Managing Requirements

We saw that PyCharm manages existing packages installed for a given interpreter. PyCharm will also help us install dependencies found in `requirements.txt` files. These `requirements.txt` files are a common method of declaring dependencies in Python applications. A requirements file can have only the package name or it can also specify a version condition. An example might have contents like this:

```
pyOpenSSL==0.13.1
pyparsing==2.0.1
python-dateutil==2.4.2
```

Think of each line of the `requirements.txt` file being fed to `pip install VALUE` one after another.

If you open this file in PyCharm, it will recognize if all of the dependencies are met. It will prompt you to install missing packages.

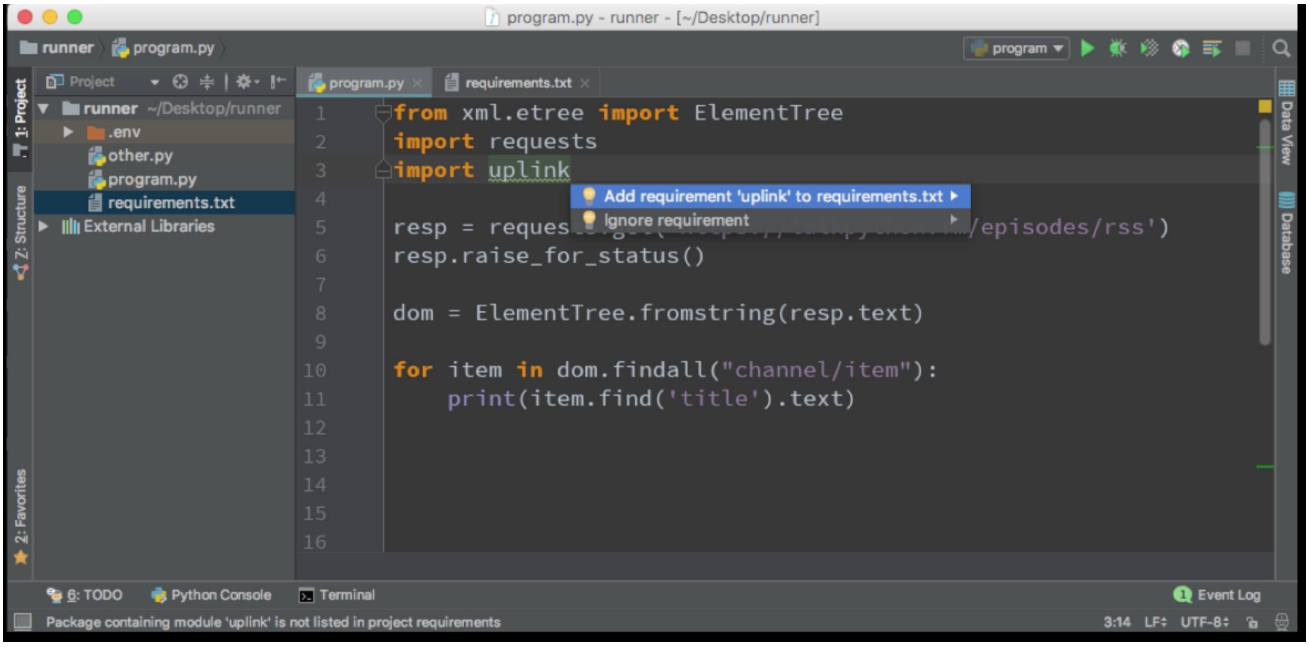


Figure 315: Figure illustrating updating a `requirements.txt` file after an import for a new package is added to a Python source file

Conversely, if you install and import a new package and it isn't installed in your active environment, PyCharm will prompt you if you want to update the `requirements.txt` declaring the new package as a dependency.

3.6 Classifying Directories

In large applications or large directory structures, the concept of marking a directory is really helpful. This is especially helpful when developing web applications. If you right-click on the directory there are four options for marking it.

The first one is *sources root*. PyCharm uses a sources root directory as the point for resolving imports. We saw in our previous example that `program.py` referenced code in `actors.py`. PyCharm claimed that the actors module did not exist. By marking the `final` directory as "Sources Root", PyCharm will index the code found there.

We also have *resource roots*. These directories are for static things like style sheets, images, Javascript, and so on. This is helpful for web development because when you create Pyramid Web Apps there's a whole folder structure that is customizable. Not every project has the exact same structure. However, all you've got to do is right-click and mark it as a resource root, and PyCharm takes care of the rest.

Excluded roots are for folders that PyCharm ignores. They are not indexed, so PyCharm will consume less memory and perform faster by using them. If you have documentation folders, virtual environment folders, or resources that don't have to do with the code in your project hierarchy, use this option. Folders that have build output that frequently changes (like webpack) will benefit from this.

Finally, there are *templates roots*. These are where web templates like Jinja2 or Chameleon files are located. Because PyCharm understands these files, you get autocomplete inside of them.

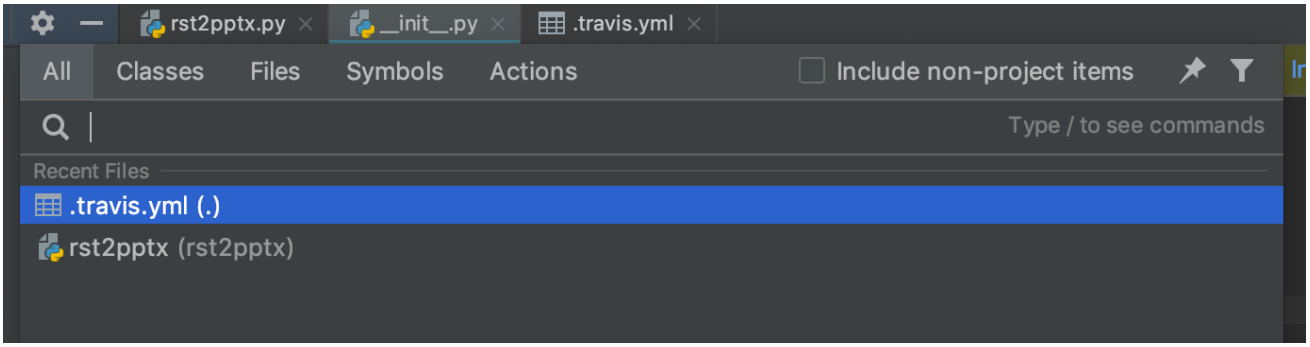


Figure 316: Figure showing "Search Everywhere" dialog with no search terms entered. In this case, it shows the most recent files.

3.7 Search Everywhere

After you have loaded a project, set up the interpreter, installed dependencies, and classified directories, you should be ready to code away. This might seem tedious, but you typically only have to set up a project once. After you have set up a project, you'll reap the benefits of the full power of PyCharm. Before we discuss the mechanics of typing and editing code in PyCharm, let's explore a feature you would use to understand the existing code: code navigation.

Here's a shortcut (that happens to not be in the PyCharm keymap cheatsheet), shift-shift (alternatively you can click on the search icon in the upper right), that brings up the *"Search Everywhere" dialog*. This name is a misnomer as it does more than searching. When you pull up "Search Everywhere", you will first see tabs for All, Classes, Files, Symbols, and Actions. The default tab is All, and if you type into the search box, PyCharm will show all the search results. You can filter them down by clicking on a tab. If you don't type anything into the serIn addition to remembering recent files, it also remembers the last location of the cursor in said file and will navigate there by clicking or arrowing over (and hitting enter) on a filename.

If you type text into the search box, the window shows classes, files, symbols, and actions. You can navigate to those areas of code, or perform those actions, by clicking on the results.

In addition, the *"Search Everywhere"* tool is not limited to working with only Python code. It can search in JavaScript files, CSV files, or any other file that is indexed by PyCharm. Remember the shortcut for this command (shift-shift), because you will be using it often. There is also an option for searching in "non-project items", which if checked allows you to search in system and third-party libraries as well. This can be very handy when you want to read the code for a package you are using (which we encourage as Python has an emphasis on readability and you will learn much by reading others' code).

Tip

The "Search Everywhere" command is one of the first you should memorize. Should you forget another keyboard shortcut or action, you can use "Search Everywhere" to find it.

3.8 Navigating within a project

In addition to search everywhere, PyCharm has another cool feature up its' sleeve. If you hold down the command (or control on Windows and Linux) key, your source code will become navigable like a web page. If you click on something such as a function call, the editor will *"Navigate*

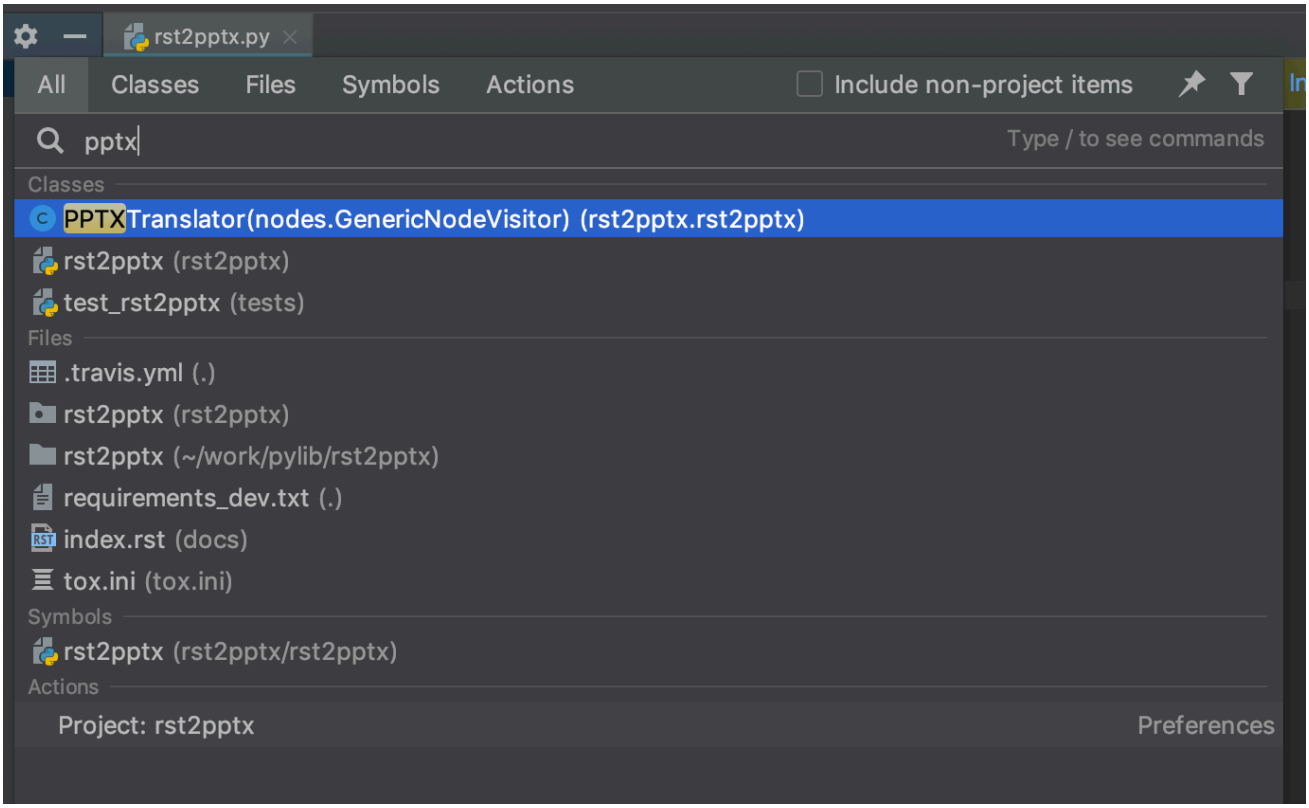


Figure 317: Figure showing "Search Everywhere" dialog with `pptx` entered as the search term. Notice that it shows classes, files, symbols (functions or methods) and actions.

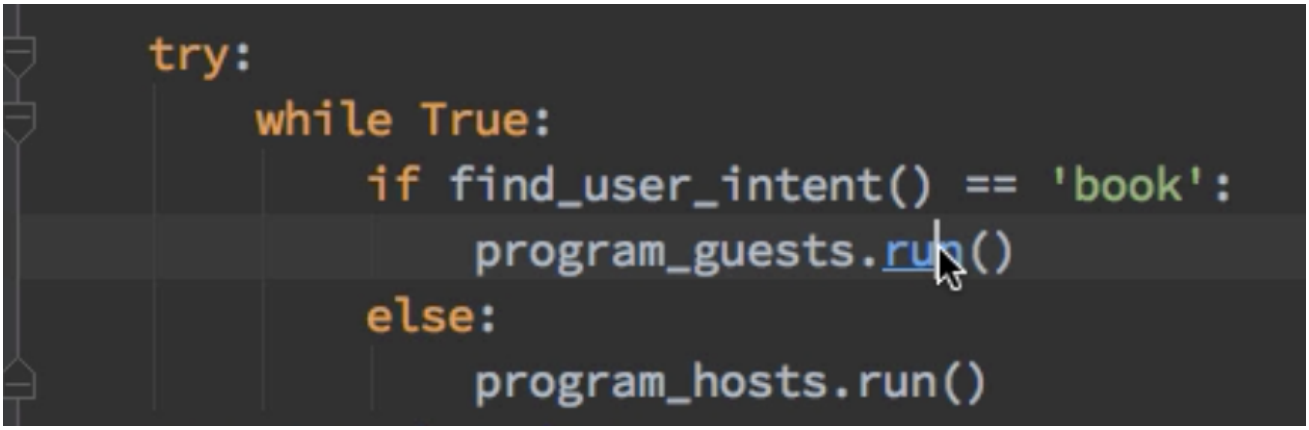


Figure 318: Figure illustrating holding down command to enable hyperlinking of code and right before clicking on `run` function.

to declaration". PyCharm even has stubs for parts of the standard library that are implemented in C. When you navigate to those, you won't see the implementation, but you will see the parameters and docstring.

The reverse operation, *"Find Usages"*, is also available. You can right-click on variable, function, method, or class and click "Find Usages" (command + F7 or control + F7).

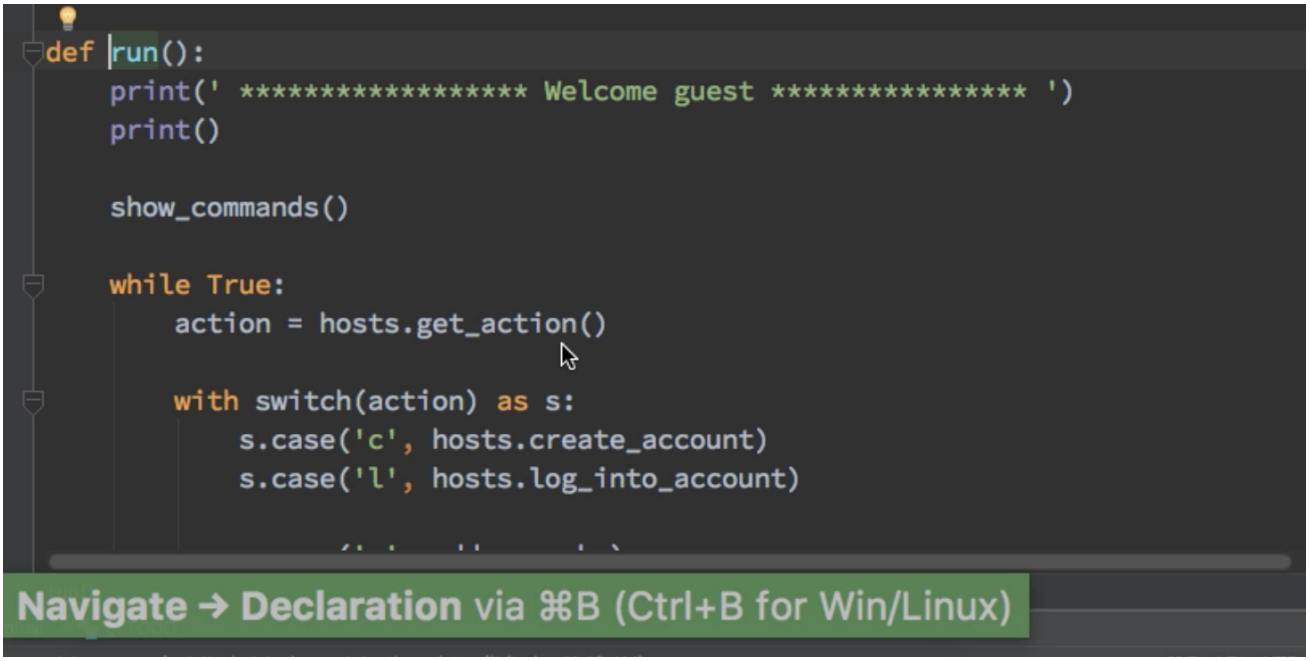


Figure 319: Figure illustrating the result of navigating to `run` declaration.

3.9 Summary

To get the most out of PyCharm, you need to configure it for new projects. This enables power tools like autocompletion and code navigation. We saw that PyCharm recognizes existing virtual environments if they are present in the top-level folder of a project. Alternatively, you can create them from a terminal or inside of PyCharm. PyCharm will also manage installing third party dependencies if you don't want to use `pip` on the command line.

In some cases, we will need to provide PyCharm with some hints about how to treat directories. After we have configured our project, we can take advantage of powerful tooling. In this chapter we also saw some of the search and navigation functionality, but PyCharm has much more to explore. We will see that in the coming chapters.

3.10 Commands

- *"Run File"* - (ctrl-shift-F10)
- *"Search Everywhere"* - (shift-shift)
- *"Navigation to declaration"* - (hold down command or control and click)
- *"Find Usages"* - (command-F7 or ctrl-F7)
- *"Mark Directory as"* - (right-click on folder)
- *"Settings/Preferences"* - (command-, or ctrl-alt-S)
- *"Terminal"* - (alt-F12)

3.11 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/1-projects` create a new project, run it, and verify the environment in it.

2. Create a new virtual environment for the previous exercise and set up a run configuration to use it.

3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the complex project in `tree/master/your-turn/1-projects`. You will import an existing project, mark directories, validate that PyCharm is indexing the code, and run the code.

4. Try out the "navigate to declaration" and "find usages" commands using the project from the previous exercise.

Chapter 4

The Editor

4.1 The PyCharm editor

In this chapter we will look into the editor a little more. It is a powerful tool for creating Python code. In addition, it can work with other file types, like HTML, JavaScript, and CSS. We will focus mostly on Python in this book and introduce some fantastic editor features including some that might not be immediately obvious.

4.2 Simple Project

We will work through coding up a simple project to retrieve information about Talk Python podcasts. We'll start by creating a project named `podcast` with a virtual environment in it and add two files. The `program.py` will contain our main logic and `service.py` will have some supporting functionality.

The `program.py` file should have:

```
def main():
    print("Welcome to the talk python info downloader")
    print()
```

The `service.py` file should have a simple stub. We can fill in the details later, but the point is that this logic will retrieve data about podcasts. Put this content in it:

```
def download_info():
    pass

def get_episode(show_id):
    pass
```

Now let's show how PyCharm helps you out. We will try and call the `download_info` method from inside of `program.py`. If you simply put `download_info()` into the body of the `main` function, PyCharm will underline the call. This red underline indicates that PyCharm has found a problem with your code, and possibly some suggestions.

If you move your mouse over the underlined code, a light bulb appears to the left. This indicates PyCharm has a *Code Intention* (an automatic fix) for you to activate. The red light bulb color indicates it can fix an error. If the light bulb were yellow/orange, it would be a suggestion or an improvement to code that already works.

When you *"Show Intention Actions"*, (alt-enter or clicking on the light bulb), you will be presented with a drop-down of options. These have suggestions such as import

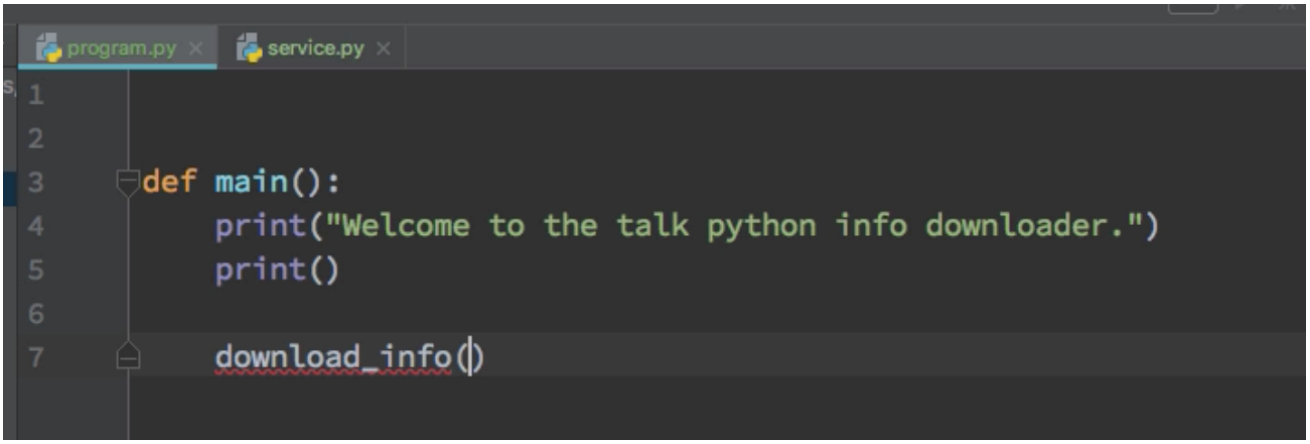


Figure 41: Figure illustrating that when we insert code that a module doesn't have in its namespace, it underlines it, indicating there are problems and potential solutions.

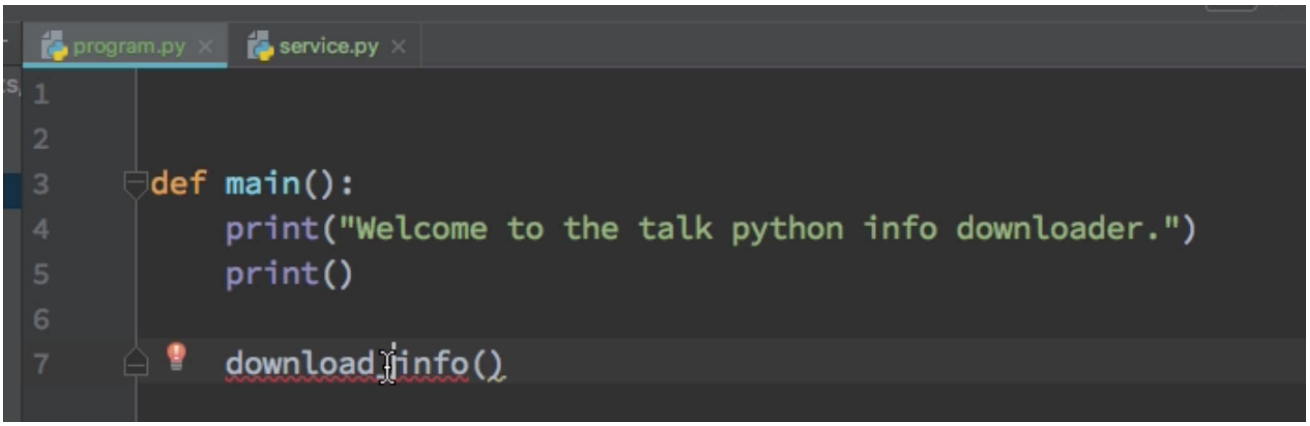


Figure 42: Figure illustrating the light bulb that appears when you move your cursor near the underlined code

`service.download_info`, create function `download_info`, rename reference, or ignore this class of errors altogether (careful here, this one is a global change in the editor).

The first option, import `service.download_info`, is what we want. If you select that option, it will insert the appropriate import statement at the top of the code:

```
from service import download_info
```

Remember we can now navigate between files using "navigate to declaration", or hitting shift-shift to jump to a previous file. You will see that PyCharm has syntax highlighting. Keywords, built-ins, strings, and user-defined code have their own color. Also if you navigate back to `service.py`, you will see that parameter `show_id` in the `get_episode` function is underlined because it is not used.

Unused code is a code smell and may indicate that there is a bug in your logic. Addressing this bug is a feature PyCharm gives you for free. Some other editors feature this, but more commonly Python programmers will use a tool like pyflakes to check for these types of issues. By having this functionality right in your editor, you save time and don't have to worry about breaking flow to leave your editor and jump into a terminal or other tool.

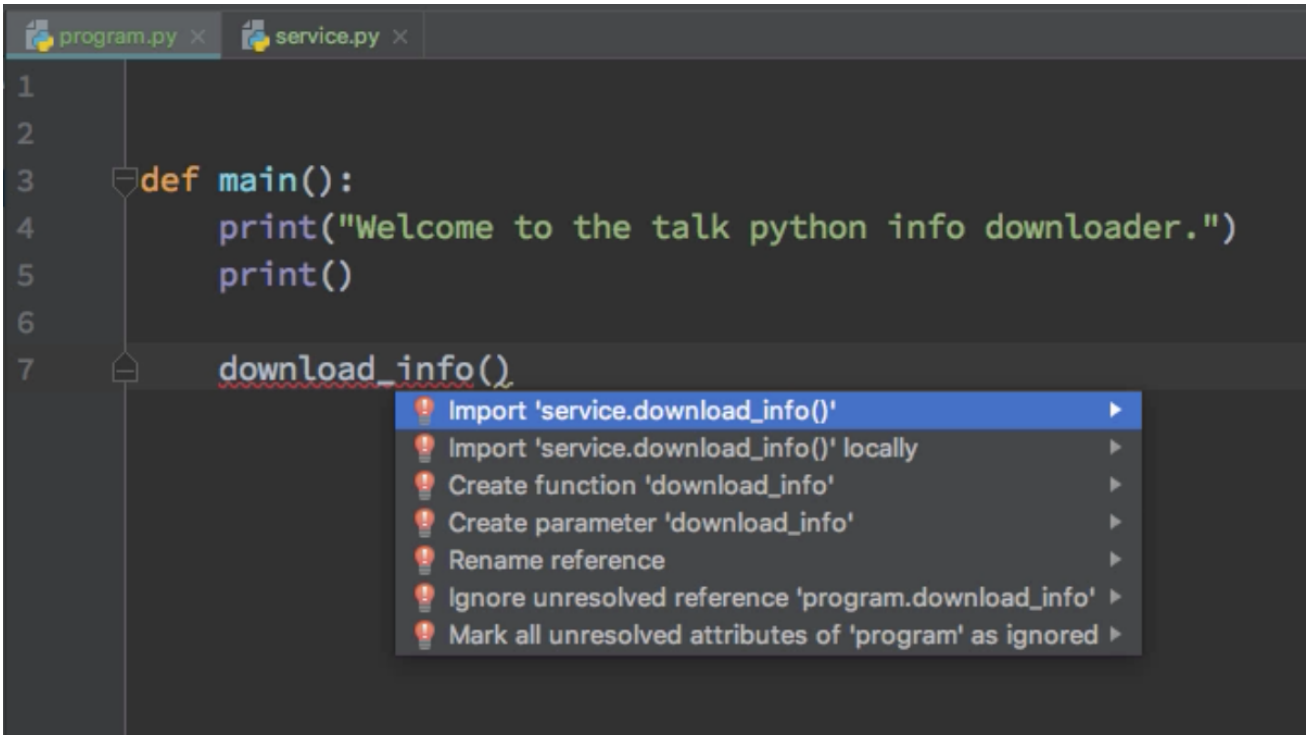


Figure 43: Figure illustrating intention actions (hitting alt-enter) when the cursor is over underlined code

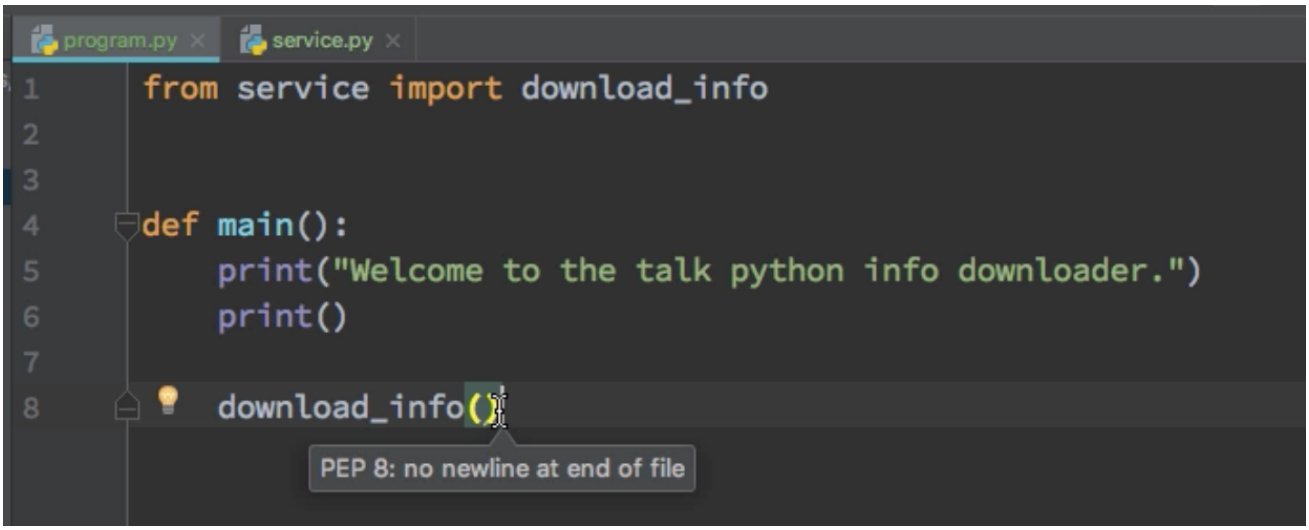


Figure 44: Figure illustrating PEP 8 violations. Use reformat code to clean these up.

PyCharm also warns of PEP 8 violations as well. The nice thing about PyCharm is that it doesn't just complain, but it also gives you the solution (or multiple solutions) to your problem. By running the *"Reformat Code"* command (alt-command-L or ctrl-alt-L), PyCharm will clean this up for you.

Another option would be to change the import statement from:

```
from service import download_info
```

to:

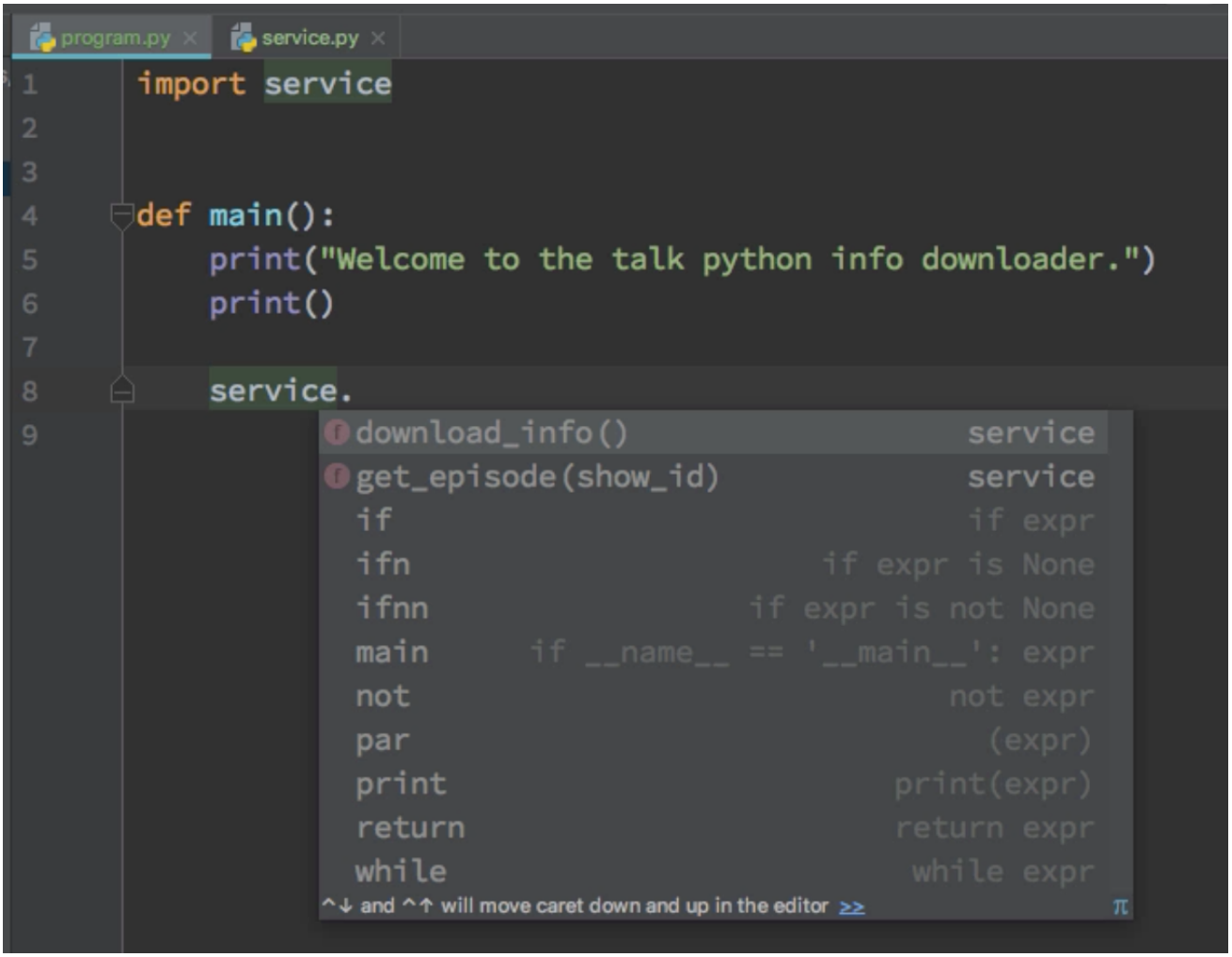


Figure 45: Figure illustrating intellisense complete on a module.

```
import service
```

Now if you go into the `main` function and type `service.` (notice the period at the end), PyCharm will present options from the `service` namespace. Among these options is `download_info`.

Inside of the `main` function let's add a loop to get information for show ids from 100 to 129. Update the `main` function to the code below:

```
def main()
    print("Welcome to the talk python info downloader")
    print()

    service.download_info()

    for show_id in range(100, 130):
        info = service.get_episode(show_id)
```

Now, we will fill in the implementation of `download_info`. You can nagivate to declaration (by holding down command or control) and clicking on `download_info`. This will open up `service.py`. We want this function to get information from the RSS feed using the requests[12] library. Change the implementation of `download_info` to this:

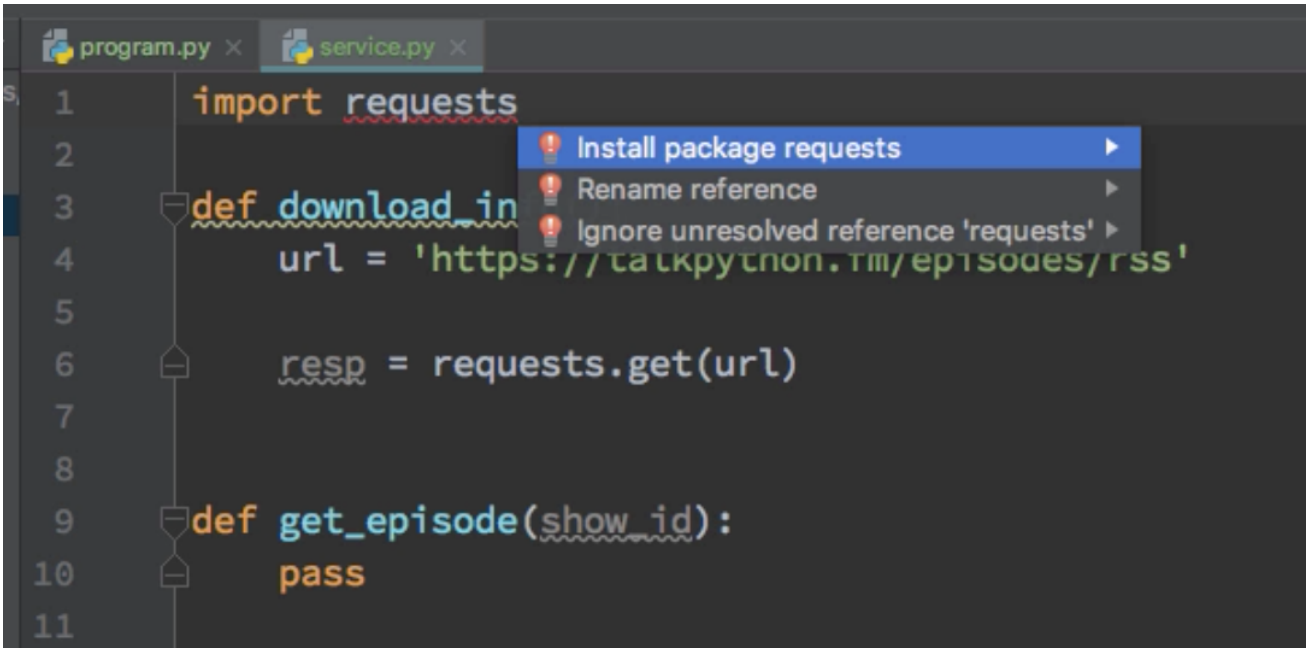


Figure 46: Figure illustrating installing the requests library from an intention action

```
def download_info():
    url = 'https://talkpython.fm/episodes/rss'

    resp = requests.get(url)
```

PyCharm will underline `requests`, as this is not in the namespace of the file. In this case, the options for intention actions will not do the correct thing (which is to import the requests library). Instead, pop up to the top of the file and add the line:

```
import requests
```

In the intention actions for this line is an option to "Install package requests". Selecting that will install the dependency for us. If there was a `requirements.txt` file, we would have the option of updating that as well.

How cool is this? We go from attempting to use an uninstalled, not imported, external package to having it installed, imported, and (potentially) added to our requirements. All of that with the slightest bit of a hint to PyCharm.

Now we are going to loop over the results of the RSS feed and pull items out using the element tree library. Update the `download_info` function to print out the length of items like this:

```
def download_info():
    url = 'https://talkpython.fm/episodes/rss'

    resp = requests.get(url)
    resp.raise_for_status()

    dom = ElementTree.fromstring(resp.text)

    items = dom.findall('channel/item')
    print(len(items))
```

[12]https://python-requests.org/

You should be able to use the intention actions to create the correct import for `ElementTree` as that is in the standard library. If you jump back to the `program.py` file and try to run it, nothing happens because we are not calling the main function. We could type out the common code pattern below that enables a Python module to operate as both an importable module and an application:

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

But, let's have PyCharm do it for us. If you have used Python for a while, this code is commonly used to indicate that we are executing a file (versus importing it for use as a library). PyCharm has a feature called Live Templates. If you type `main`, PyCharm will present a live template for you that has that conditional in it as the first intention action. Hit TAB while it is selected, and PyCharm will insert that code for you.

Tip

When you create a main function, PyCharm will add a green gutter icon next to it. This makes it really easy to run your code.

Live templates are awesome to save time, avoid mistakes, and even help prevent carpal tunnel issues. You can take advantage of the built-in live templates and later we will show how to create your own.

If you run the code now, it should tell you how many episodes there are. Let's continue updating our code to flush it out and also explore more of PyCharm. In the `service.py` file add an import for `namedtuple`, define a `namedtuple` to store episode information, and update the `download_info` logic, so the code looks like this:

```
from xml.etree import ElementTree
from collections import namedtuple
import requests

Episode = namedtuple('Episode', 'title link pubdate show_id')
episode_data = {}

def download_info():
    url = 'https://talkpython.fm/episodes/rss'

    resp = requests.get(url)
    resp.raise_for_status()

    dom = ElementTree.fromstring(resp.text)

    items = dom.findall('channel/item')
    episode_count = len(items)

    for item in items:
        episode = Episode()


def get_episode(show_id):
    pass
```

Put the cursor inside of the call to `Episode` in the `for` loop. If you hit *Parameter Info* (command-p or ctrl-p), PyCharm understands named tuples and will show you the parameters to call it with.

Having parameter information is very useful when calling class initializers, methods, or functions.

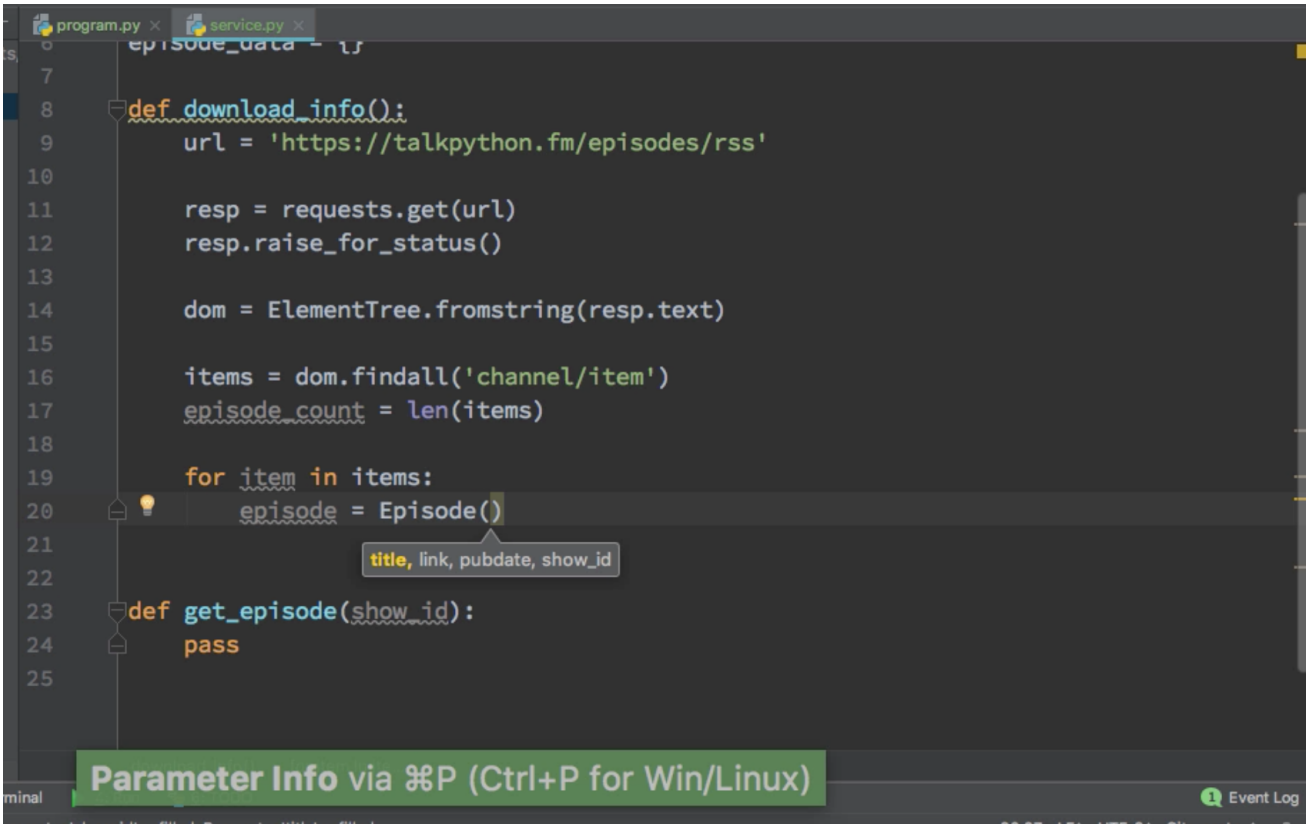


Figure 47: Figure illustrating parameter information when the cursor is inside of a named tuple.

Our next step in our little downloader app is to populate these `Episode` namedtuples from the RSS feed. We will need to change the for loop to something like this:

```
for idx, item in enumerate(items):
    episode = Episode(
        item.find('title').text,
        item.find('link').text,
        item.find('pubDate').text,
        episode_count - idx - 1
    )
    episode_data[episode.show_id] = episode
```

Down in the `get_episode` function add the following implementation:

```
def get_episode(show_id):
    return episode_data.get(show_id)
```

Update the `program.py` `main` function to:

```
def main()
    print("Welcome to the talk python info downloader")
    print()

    service.download_info()

    for show_id in range(100, 130):
        info = service.get_episode(show_id)
        print(f"{info.show_id}. {info.title}")
```

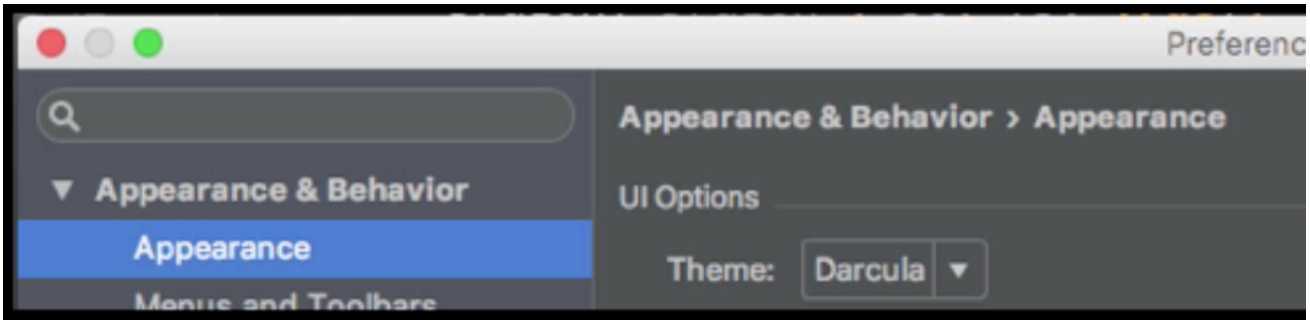


Figure 48: Figure illustrating theme selection.

Note that PyCharm supports f-strings in Python 3.6 and above. It also has completions inside of them. Sometimes, though PyCharm isn't smart enough to give us completions. In the above code, it would not give us completion on the `info` object, even though we know it is an `Episode` named tuple. To help PyCharm, we can take advantage of another Python 3 feature, annotations. Update the `service.py` `get_episode` function to:

```
def get_episode(show_id: int) -> Episode:
    return episode_data.get(show_id)
```

With the above code, PyCharm gives us completion on the result of calling `get_episode`!

Note

Since the `.get` method can return `None`, the return result of `get_episode` should really be `typing.Optional[Episode]`.

A couple of things to note as we were working on this project. You don't really need to save files. Whenever you run them PyCharm takes care of that for you. Also, if there is an error during running, the stacktrace in the terminal window has hyperlinks and you can click them to go to the source of the error.

This overview should have given you a glimpse into of the power of PyCharm. While amazing, this is just the surface of what it can do.

4.3 Syntax Highlighting

As we saw, the editor has Syntax Highlighting. PyCharm colors (and treats) keywords, built-ins, variables, strings, and user-defined code differently. This is configurable. In the figures in this book, the screenshots are using the theme "Darcula". You can change the theme if you go to *Preferences* (command+, on macOS) or *Settings* (command-, or ctrl-alt-S on Windows and Linux) and search for "Appearance". Under "UI Options" there is a "Theme" selection box.

If you want to install your own theme you can go to the "Color Scheme" preferences page. On that page, next to the scheme name is a gear where you can import a new theme. The Color Themes[13] website has a bunch of themes to download.

Another option is to customize an existing theme. In the "Color Scheme" preferences page, you can customize the font and color for many different file types. You can also export that scheme to share with others (or other computers).

[13] http://color-themes.com

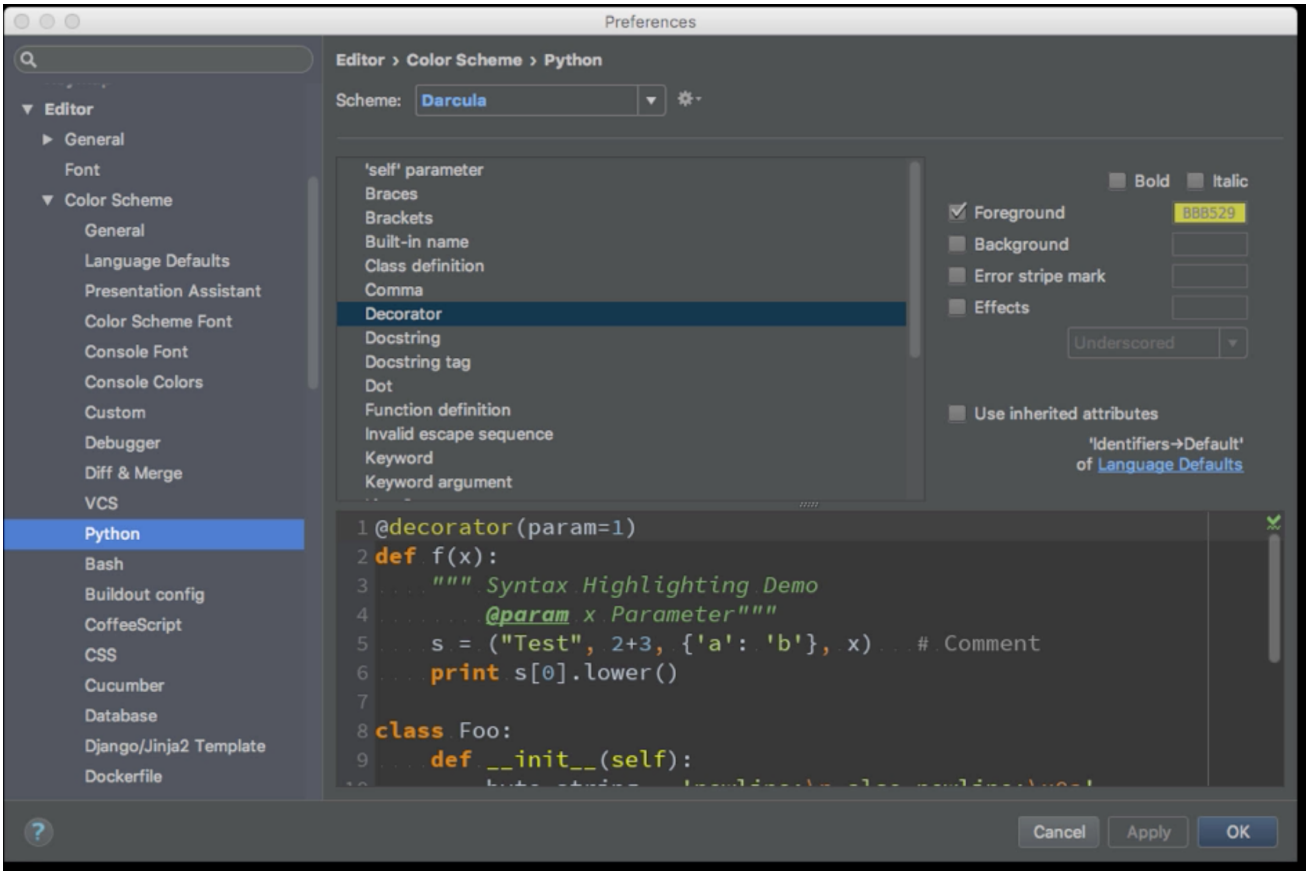


Figure 49: Figure illustrating customizing the decorator options for Python code.

There is also an "Appearance" preference page to future adjust the editor. You can tweak accessibility, UI options, Window options, and more.

4.4 Code Completion

PyCharm has *code completion* for code that it understands. As we saw in the example, sometimes we need to provide annotations to aid PyCharm with suggestions. Let's explore code completion.

If you have a variable called `name` and you type `na` later in your code, PyCharm will suggest `name` as a *basic completion*. If you hit TAB it will fill in the missing characters for you. Alternatively, you can type CTRL + Space to bring up code completion if it was dismissed or not active for some reason.

If you type CTRL + space a second time, PyCharm will present you with more options. It will show classes, functions, and modules with that name.

PyCharm will also offer suggestions for completions on object attributes (anything following a period). Normally, the options will pop us automatically after you type `.`, but if you want to go back and perform a new lookup, type CTRL + Space again. The letter in the icon on the left indicates the type of the completion: class (c), field (f), function (F), method (m), parameter (P), property (p), and variable (v). On object instances, the right side of the completion indicates the class where the attribute is found.

Sometimes PyCharm is not able to provide intelligent completions. This is due to the dynamic nature of Python. If the completions in PyCharm don't make sense, it is probably because PyCharm

Figure 410: Figure illustrating basic completion of a variable name.

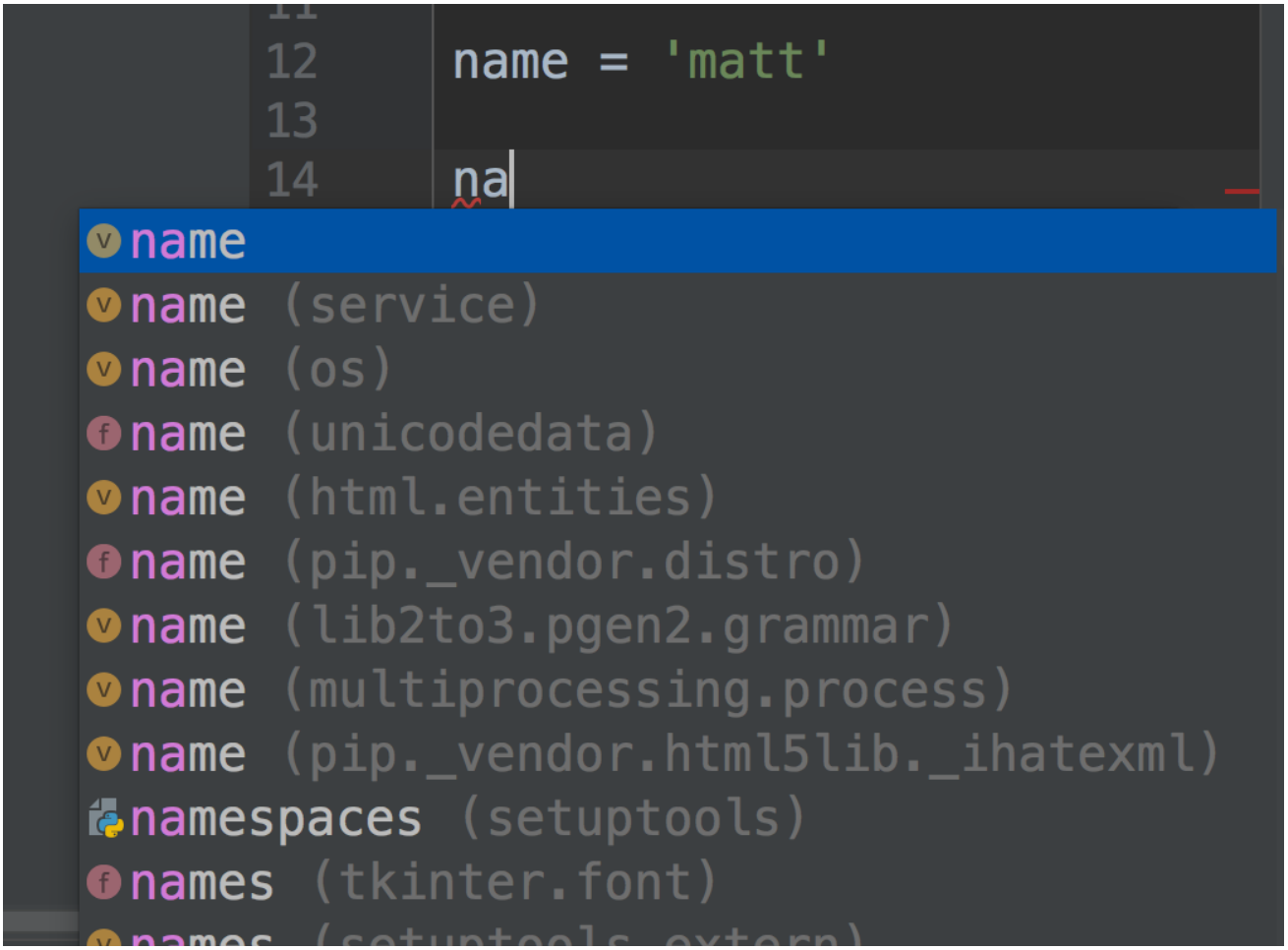


Figure 411: Figure illustrating hitting control space a second time to get more code complete options.

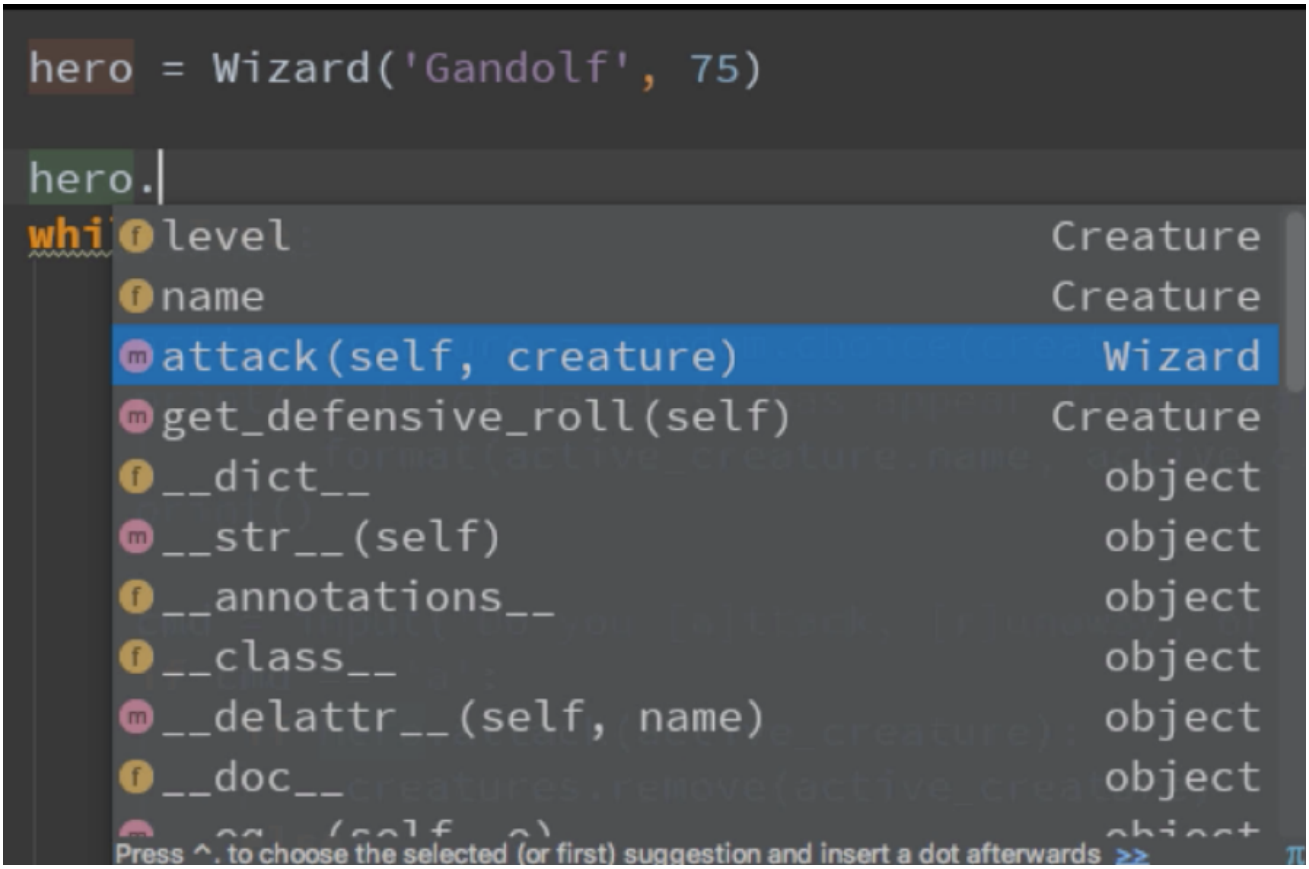


Figure 412: Figure illustrating the icons for completion on the left. "f" is for field and "m" is for method. On the right are the classes where the attribute is defined. A `Wizard` is a subclass of `Creature` which is a subclass of `object`.

isn't able to divine a type for a variable. You have a couple options, you can live without code completion, add a type annotation to the variable, or add a type annotation for the return type on the function. The *"Show Intention"* command (alt + return) will include an option to "Add a type hint".

4.5 Intention Actions

When PyCharm shows a bulb icon, it is indicating that there is a way to change your code. If the bulb is red with an exclamation point, it indicates there is an error and PyCharm has an associated fix. By clicking on the bulb (or hitting alt + return), you bring up the *"Show Intention"* drop-down. Common intentions include: creating code from usage, quick fixes (add missing import for module), micro-refactorings, removing dead code, PEP 8 cleanup, helping performance issues, fix Python 2 or 3 mismatches, package maintenance, and more.

Don't ignore these options. Take your time to read them and even try them out to see what they do. You can always undo an action later. We will be discussing many of them in this book. If you find that there are some warnings that are false positives or are getting in your way, you can always suppress them. Click on the "Settings..." dropdown from the intention and select the suppress option.

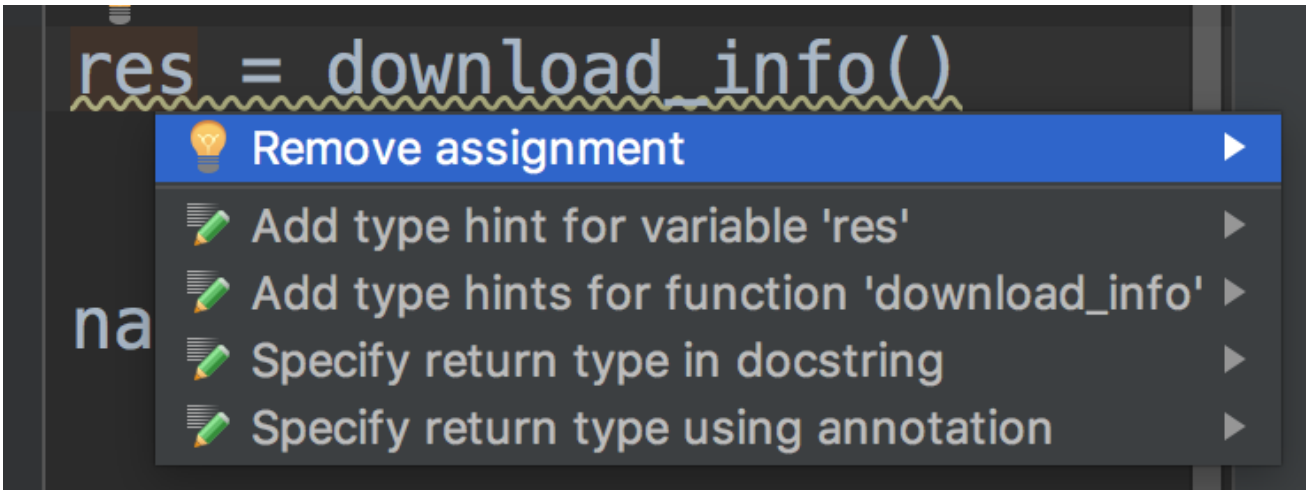


Figure 413: Figure illustrating the result of Show Intention (alt + return) on the `res` variable. We can add a type hint to the variable, or to the `download_info` function. This will enable intelligent code completion.

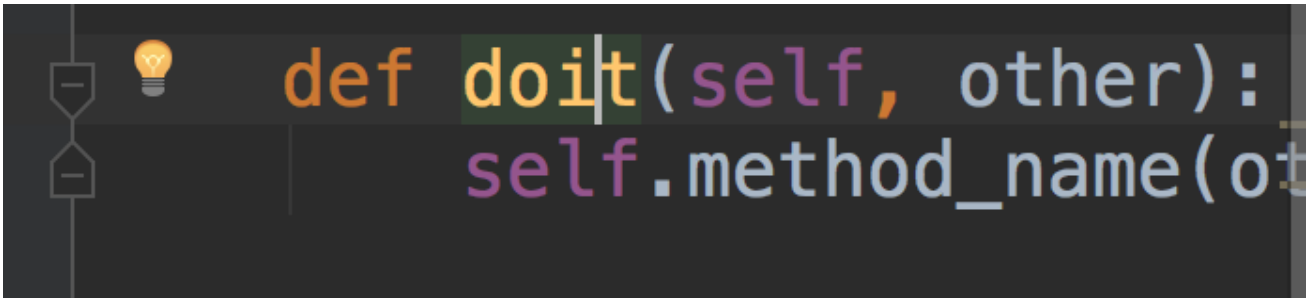


Figure 414: Figure illustrating the bulb icon. You can click on it or type alt + return to bring up the intention options.

4.6 Discovering New Features

PyCharm includes extensive documentation and help topics. If you forgot how to do a command, you can search for it in the help menu. For example, if you saw a bunch underlines because you were not PEP 8 compliant and wanted to reformat your code but forgot how to, use the help menu. In there is a *"Find Action..."* (command-shift-a, or ctrl-shift-a). If you type "reformat" in there it will bring a list of suggestions. Near the top is the *"Reformat Code"* command with the shortcut (command-alt-l, or ctrl-alt-l).

Note

Mac users have another option. In the "Help" menu is a search box. If you enter "Reformat" in there, it brings up a list of menu items and if you select the menu item, it will bring up the menu and point to where that item is. This works ok if you prefer to use the mouse, but it is better to remember the hotkeys for frequently used commands.

"Find action" is a powerful command and one of the first you should memorize. For example, if you wanted to refactor some code into a function, you can highlight that code, open up the find action window, and type "refactor". You will see that you can open up the *"Refactor"* menu itself, or call *"Refactor Method"* (command-alt-m, or ctrl-alt-m) directly.

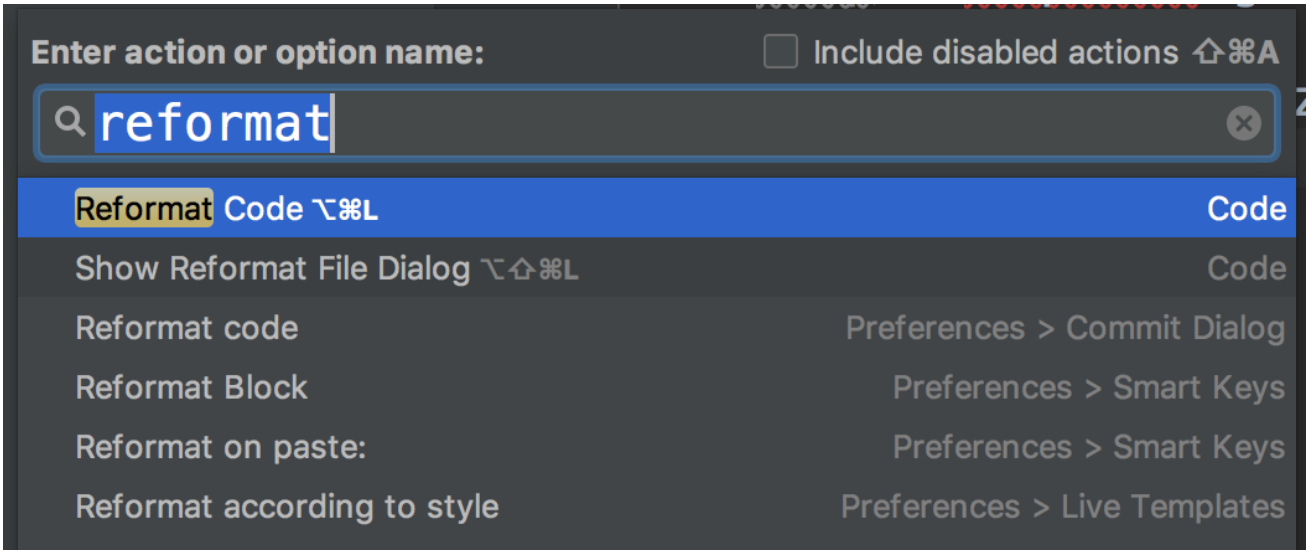


Figure 415: Figure illustrating using the find action popup to search for the command to "Reformat Code".

Note

PyCharm uses the term "method" for refactoring a function (code at the global level or in a function) or a method (a function bound to in a class).

4.7 Keymaps

All of the actions in PyCharm are configurable. Out of the box these are preconfigured for you and use the mappings found in the cheat sheet referenced earlier. If you want to customize any keycode, you can in the Preference window. Search for "keymap" to open up the screen to adjust them. From there you can customize individual commands or you can tell PyCharm to use a different keymap. For example, if you are used to Emacs, there is a Emacs option the "Keymap" drop-down that has pretty good emulation for Emacs.

Alternatively, if you are a Vim user, there is a plugin, IdeaVim, that supports many of the Vim features. You will need to go to the "Plugins" screen on the preferences page and search for "vim". From there you can click the "Install" button.

Note

If you change the keymap many of the books in this command will have a different command. For example Emacs uses control space to mark a selection, so it won't work for code completion.

4.8 Formatting and Code Cleanup

You might have code that has various PEP 8 issues. Some of these are trivial fixes. Others, like using *camel-case* naming conventions (capital letters between words) instead of *snake-case* (underscores between words) are not always easy to change. If you are using a third-party library (or even a module in the standard library like `unittest`), you can't just rename the violating functions or methods.

Consider the code below:

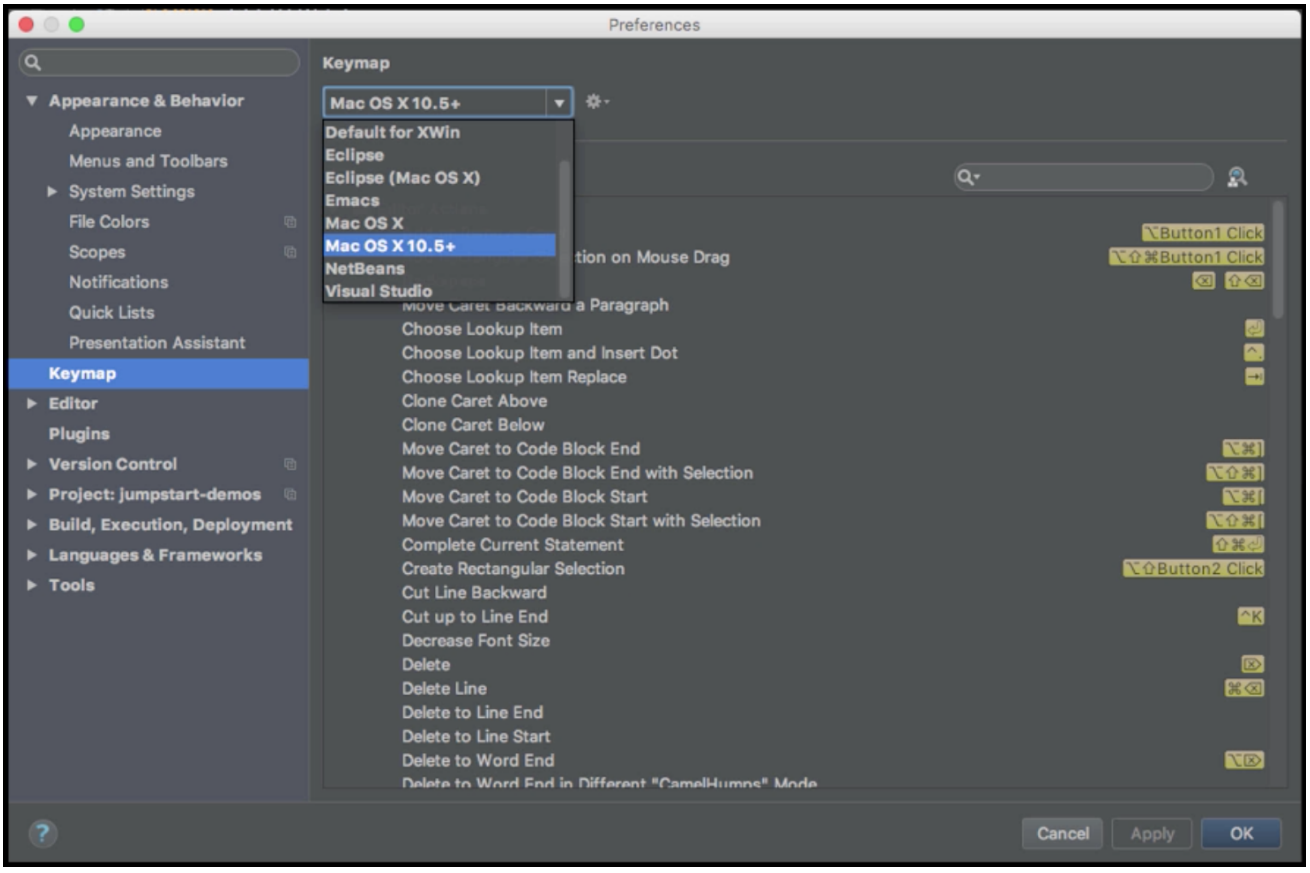


Figure 416: Figure illustrating changing hotkeys or the global keymap.

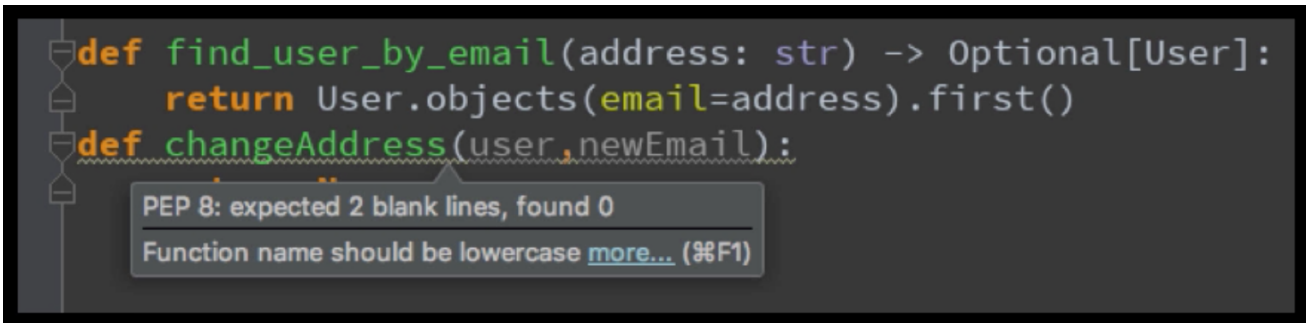


Figure 417: Figure illustrating issues found in the code. There are no blank lines and the function name is not lowercase.

```
def find_user(username):
    return User.objects(name=username).first()
def changeAddress(user,newEmail):
    return None
```

It has PEP 8 violations because there are no spaces between the functions. Also, `changeAddress` and `newEmail` are camel-case, and there is no space before `newEmail`.

There is a command *"Reformat Code"* (cmd + L, or ctrl + L) that will address some of these issues. The code will look like this following that command:

```
def find_user(username):
    return User.objects(name=username).first()


def changeAddress(user, newEmail):
    return None
```

Note

If you want to format a whole project, you can run the Run -> *"Code Cleanup..."* action. (There is no shortcut for this command, but hitting shift + shift and typing in the first letters of the command will pull up the action. If you prefer to type instead of using the mouse, it is possible to run this without reaching for the mouse.)

Notice that the name of `changeAddress` was not converted to a PEP 8 compliant name. In this case we could use the *"Rename"* command (shift-F6) on the function declaration. Note that you can not do this refactoring on calls to that function, you have to do it on the function itself. If you come across libraries with non-PEP 8 compliant naming, unless you are the author of third-party libraries, you will just have to live with that.

4.9 Code Formatting for Teams

If you are working on a team, you will have multiple developers checking in code to the source control system. If your team has not adopted a style guide, managing these changes can be a headache. A good goal is to remove friction caused by indentation preferences, newlines, comma placement, and naming. By having a style guide, it can eliminate many of these problems.

There are many style guides in the wild. PEP 8 is a style guide, but it is pretty flexible. We would suggest something a little more rigid. Google has a style guide for Python code[14], OpenStack has another[15]. There are third-party command line tools such as yapf[16] and black[17], that you can run from the command line. There are plugins[18] for both of these tools so they can be run from PyCharm or the command line (if you have colleagues who prefer other editors).

Alternatively, if you have specific needs, you can customize PyCharm's *"Code Style"* options. Then you can export them, and share them with your team members.

If you have multiple teams with different styles, or also work on open source or other projects with different standards, PyCharm can deal with that as well. Use the "Scheme" drop-down and select "Project" there in the Code Style preferences for Python. This will allow you to specify styles on a per-project basis.

[14] https://github.com/google/styleguide/blob/gh-pages/pyguide.md

[15] https://docs.openstack.org/hacking/latest/user/hacking.html

[16] https://github.com/google/yapf

[17] https://github.com/ambv/black

[18] https://plugins.jetbrains.com/plugin/10563-black-pycharm

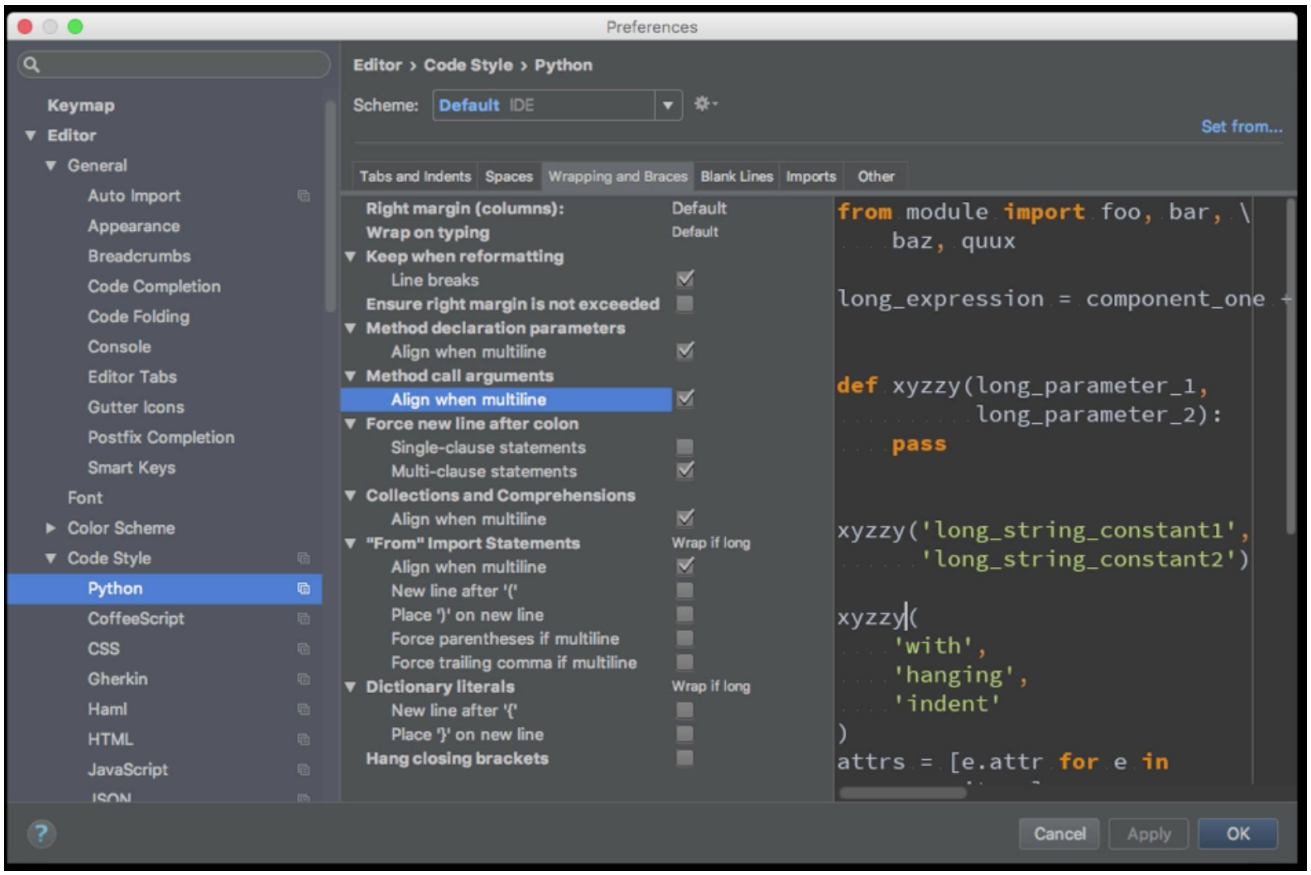


Figure 418: Figure illustrating changing code style preferences. You can also import and export these settings from this page.

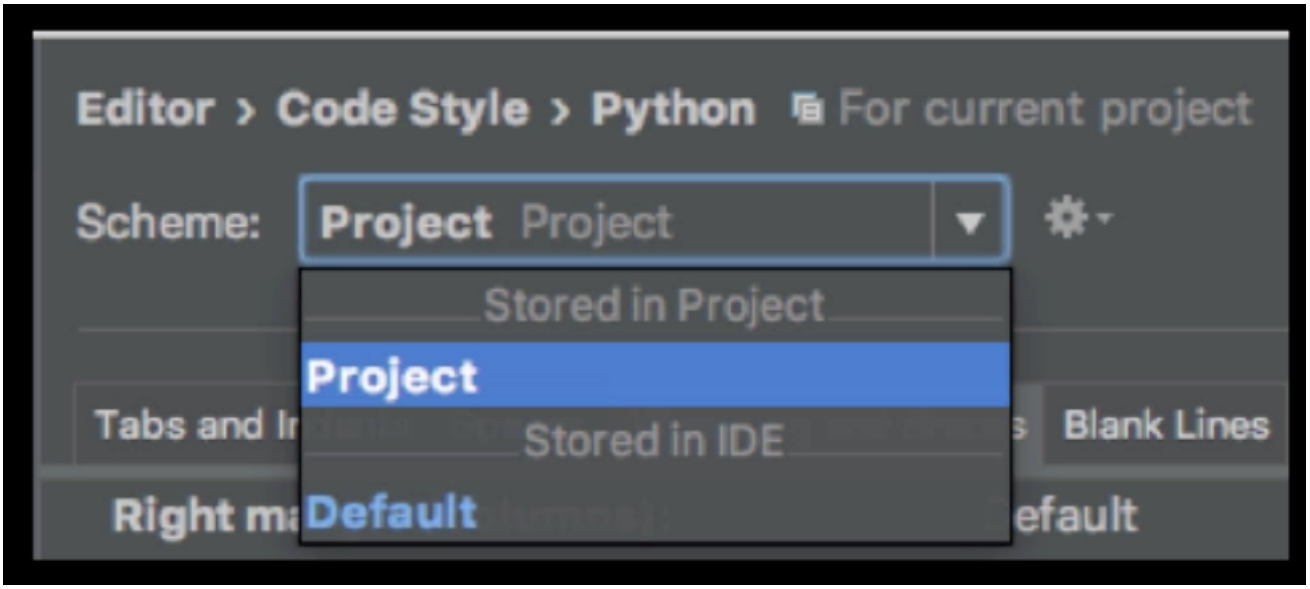


Figure 419: Figure illustrating setting the scheme to ”Project” so that the project formatting is not global.

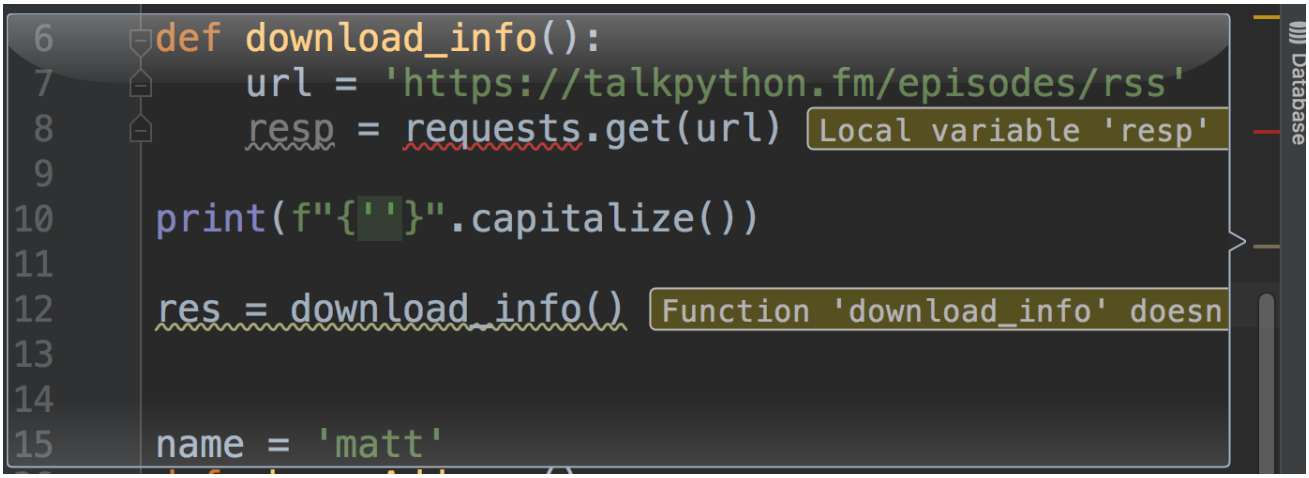


Figure 420: Figure illustrating Lens Mode. To activate lens mode hover the mouse over the stripe in the scrollbar. If they are outside of the scroll box, the lens mode popup will appear

Configuring formatting is important because it makes interacting with others much easier. The last thing a code reviewer wants to see when reviewing code is a bunch of whitespace tweaks, with that single meaningful change to code lost in the noise.

4.10 Lens Mode

As PyCharm is evaluating your code, you will notice that there are stipes embedded in the scrollbar. These stripes highlight errors or warnings. If you click on them, PyCharm, will navigate to that point in the file and bring up the bulb so you can fix the issues. You can also hover over the stripes. If you hover over a stripe for code that is currently on the screen, PyCharm shows a tooltip with the error or warning. If you hover over a stripe that is above or below the window view, PyCharm will bring up *Lens Mode*. The lens mode tooltip will show the issue and some of the code around it to give some context to it.

4.11 Object-oriented and Class-based Features

Python is a multi-paradigmatic language supporting many styles of coding. You can code in an imperative style (like C or Basic), a functional style (like Lisp or Scheme), and an Object-oriented style (like C++ or Java). If you are defining classes and nest class hierarchies, PyCharm has some features to help you understand the code better. PyCharm has *gutter icons* to help you navigate and understand the code.

Assume we are building a game (old school MUDS anyone?). We may have created a `Creature` base class that has `Wizard` and `Dragon` as subclasses. If you looked at the source for `Creature` there would be gutter icons in the left gutter. There are also arrows by the gutter icons. The up arrow indicates that there are parent methods. The down arrow tells you what subclasses override the class or method. You can hover over the gutter icons to see what classes are referenced. Or you can click on a gutter icon to navigate to the implementation.

Gutter icons are very useful for navigating type hierarchies in object-oriented programs.

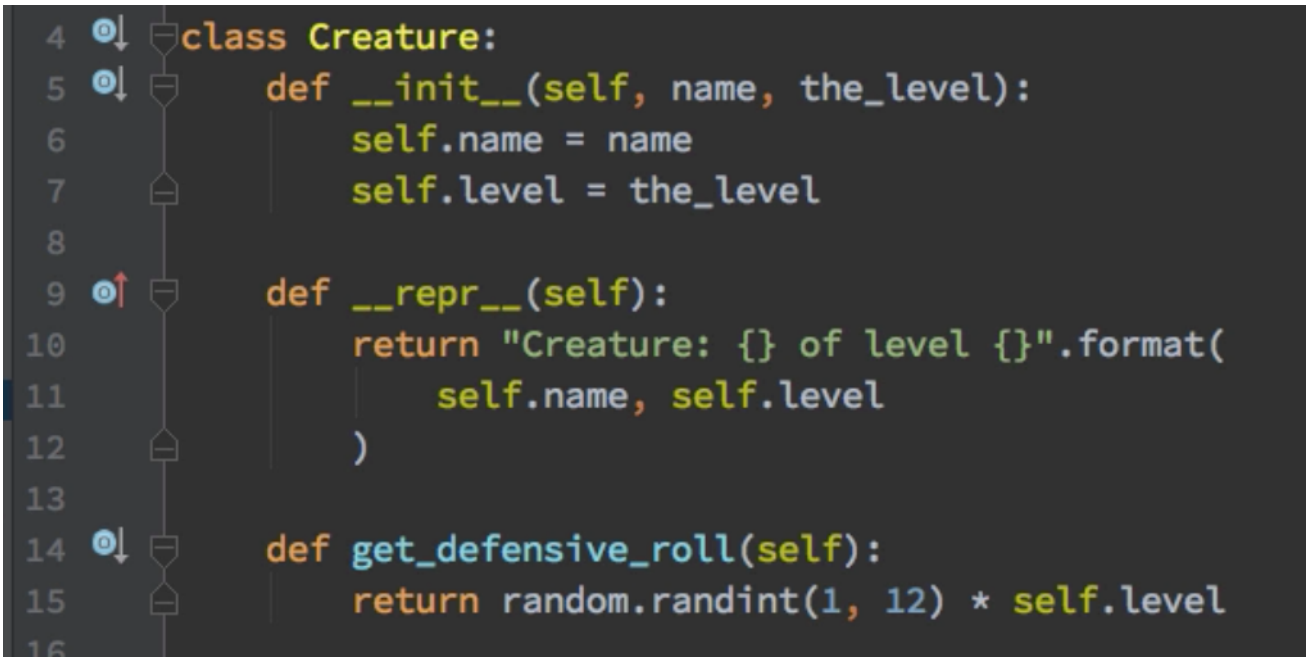


Figure 421: Figure illustrating the gutter icons. On the left of `class`, `__init__`, `__repr__`, and `get_defensive_roll` are the gutter icons. The down arrows indicate that there are subclasses (on the class) or that subclasses override the methods. The up arrows indicate that the method is overriding a base class (`object` is the implied base in Python 3)

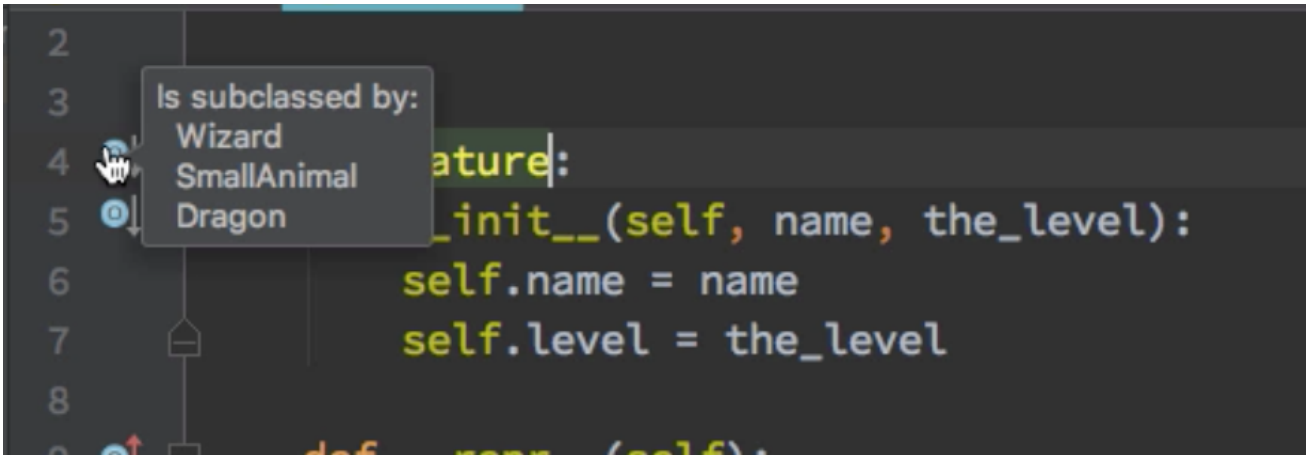


Figure 422: Figure illustrating hovering over a gutter icon. In this case, it indicates which classes subclass the `Creature` class. We can click on the icon to navigate to those classes.

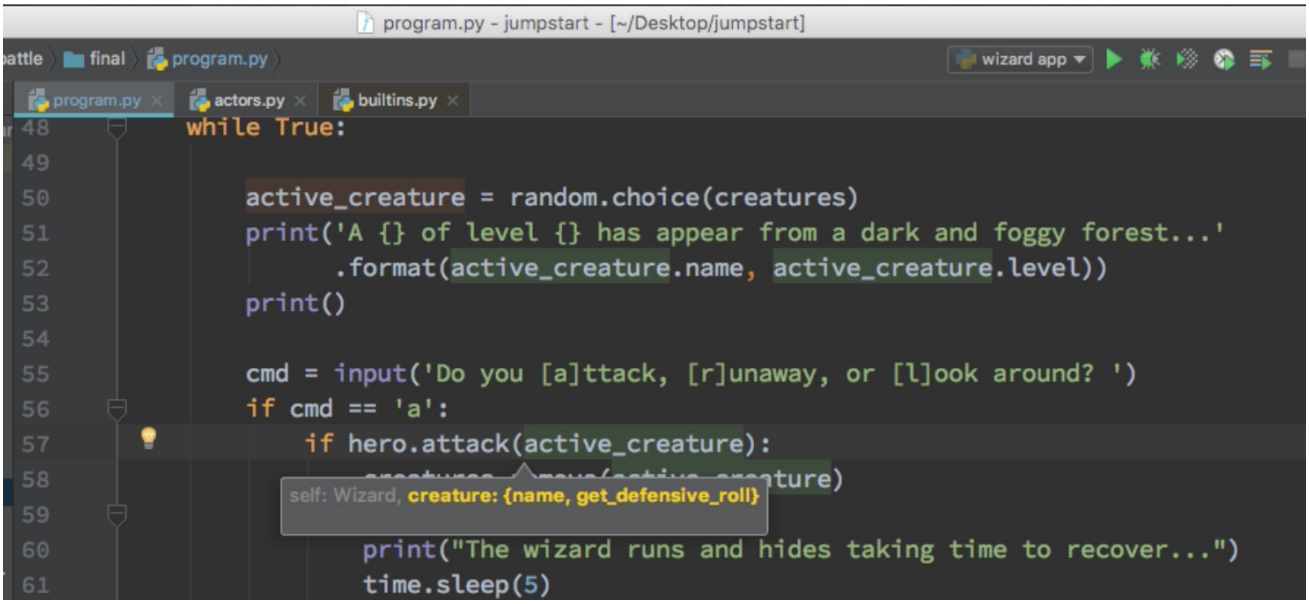


Figure 423: Figure illustrating parameters for the `.attack` method. Notice that it says a creature should have a `name` and `get_defensive_roll` attribute).

4.12 Viewing Documentation

Let's explore some of the other help that PyCharm provides. When you are calling a function or method, PyCharm will bring up the *"Parameter Info"* (command-p, or ctrl-p) if you just typed out the function name. This hint is useful to see what parameters to provide. This tooltip does not pop up if not are cursoring inside of a function call later, you will need to explicitly call it.

If you want to see the documentation for the method, use the *"Quick Documentation"* command (ctrl-j on Mac, or ctrl-q on Windows/Linux). This will pull up a popup with the docstring and any type information or inferred types.

4.13 Creating Documentation

Let's look at how to create documentation. If you on a function or method you viewed the quick documentation and found it empty, you could add the documentation yourself. Hit navigate to declaration from that function to locate it. If you type a triple-quoted string (the " character 3 times) and hit enter, PyCharm will stub out some of the documentation. It will fill in the parameter names and the return value. There is no hotkey for this, it is automatic, but if you want to change the format of the docstring, you can edit that in the Python Integrated Tools section of the preferences.

PyCharm also tracks the parameter names in a function. If you rename those, PyCharm will rename the documentation as well. Also, any types in the quick documentation are hyperlinks, so you can quickly navigate to the related documentation as well.

4.14 Find Usages

If you are considering a refactoring, you might ask yourself, is this function even being called anymore? PyCharm has a *"Find Usages"* command (alt-F7) that will enumerate the places where it is called.

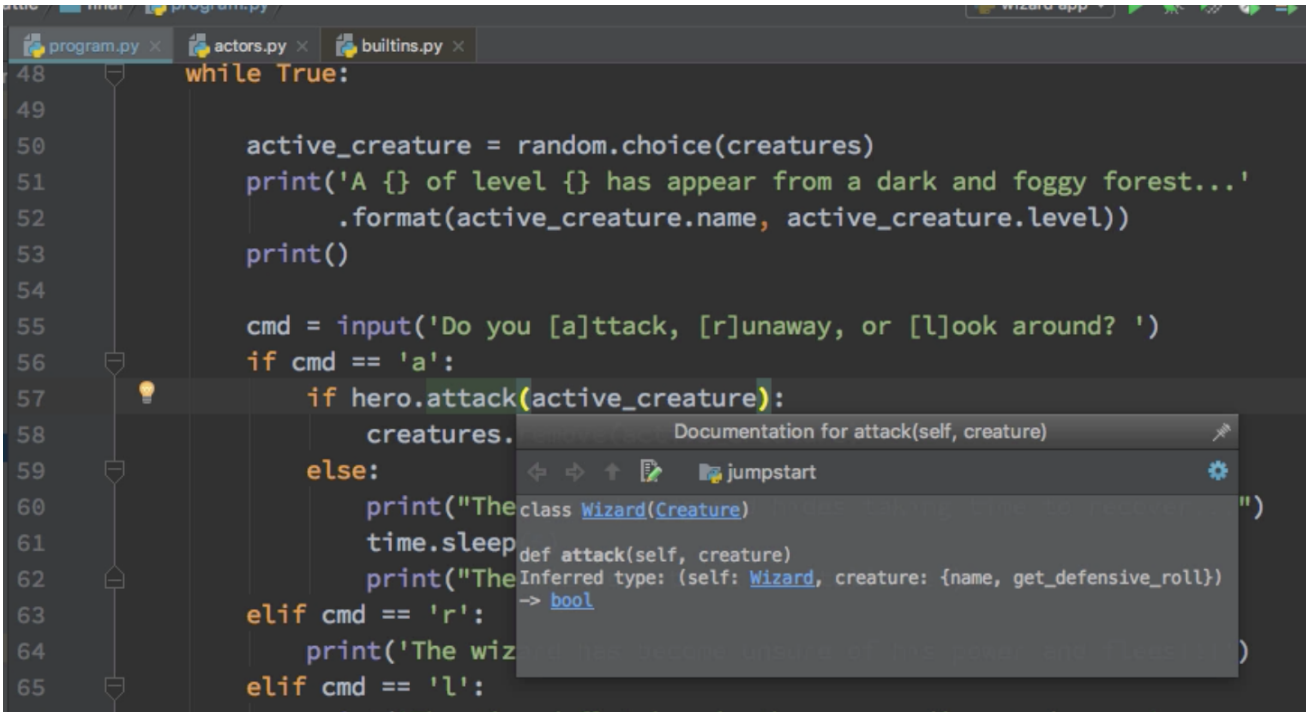


Figure 424: Figure illustrating quick documentation for the `.attack` method. If there were a docstring on this method it would be shown. Notice that the type information is present as well (in this case it is inferred).

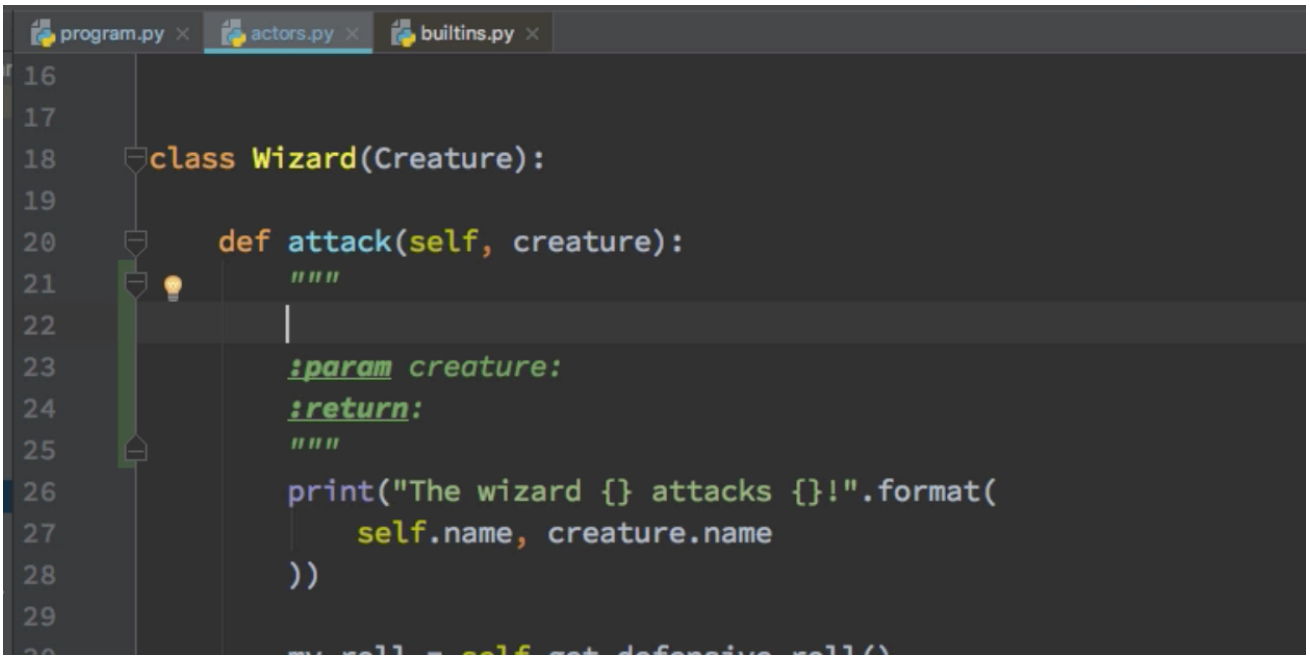


Figure 425: Figure illustrating the result of inserting triple quotes and hitting enter. PyCharm stubbed out the parameters and return value.

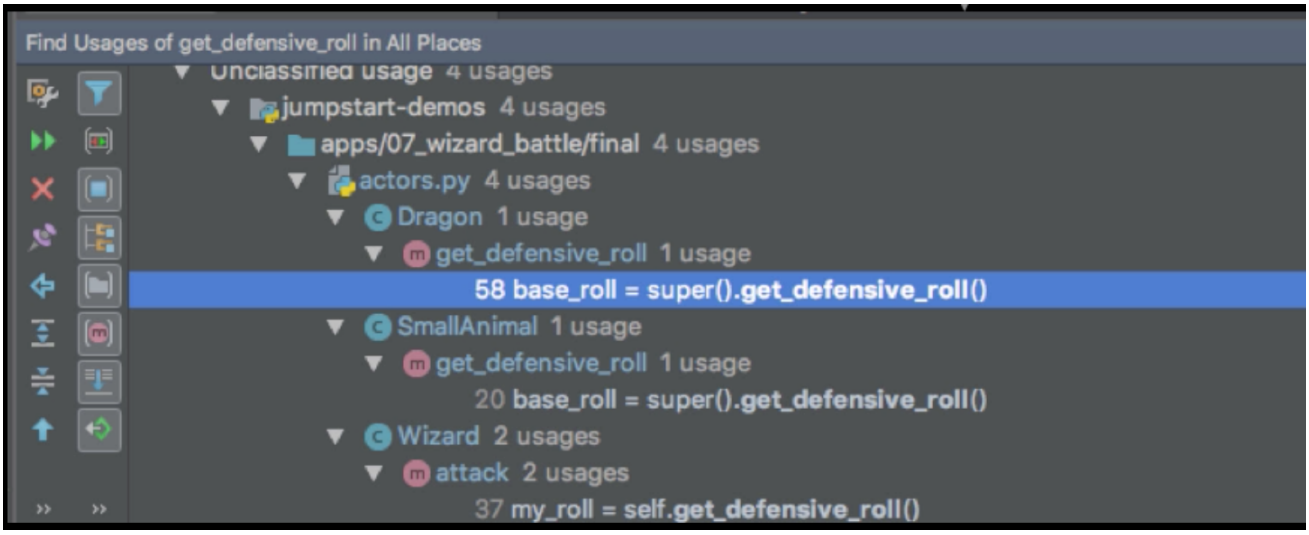


Figure 426: Figure illustrating the result of calling find usages on the `.get_defensive_roll` method.

Again, this is more than a simple grep through the files. PyCharm understands how code is referenced and will present template results as well. If there are multiple classes that happen to use the same method name, PyCharm should be able to find the correct method and not show the other method. PyCharm is also able to do intelligent renaming across the whole project.

4.15 Live Templates

Previously we saw that if we typed `main` and hit TAB (or enter), PyCharm would use a *Live Template* to create skeleton code for:

```
if __name__ == '__main__':
    #
```

It would also place the cursor where the comment is in the code above. This is a simple template, there are others that are more complex, such as typing `iter`. If you type `iter` and hit TAB you will see code like this with `CURSOR` being the actual cursor rather than the word:

```
for  in CURSOR:
```

You can type in what you want to enumerate over, say `range(10)`, and then hit TAB again. You will see the cursor move to right after the `for` keyword:

```
for CURSOR in range(10):
```

You can enter a variable name for iteration, say `i`, and hit TAB. You will see this:

```
for i in range(10):
    CURSOR
```

Now you can fill in the body of the for loop.

There are many live templates. You will want to explore the available ones. Open the preferences window and search for "Live Templates". If you expand the Python section, you will see a list of them. There are also templates for other file types as well. You can create more if you like. If you click on the `iter` template, you will see that the "Template text" for it reads:

```
for $VAR$ in $ITERABLE$:
    $END$
```

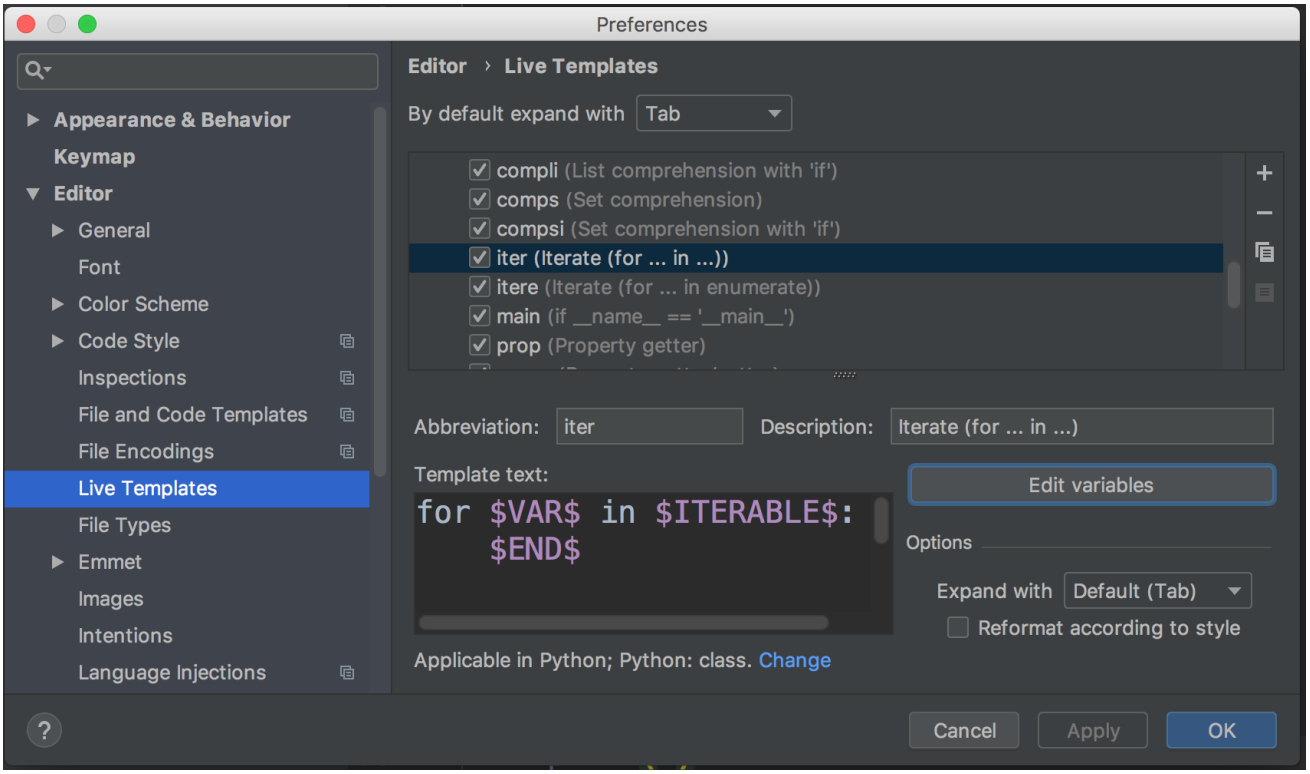


Figure 427: Figure illustrating the live template for the `iter` abbreviation. You can click the "Edit variables" button to change the navigation order of the variables.

You might be wondering, given the template, why the cursor went to `$ITERABLE%` first instead of `$VAR$`. On the right side is a button called "Edit Variables". If you click it shows you the order that the variables are entered. You can reorder them as desired.

If you find yourself repeatedly typing the same code, you can make a template for it. Click the plus button to create a new template, give it an abbreviation and description, and tell it what context to run in (ie. Python, XML, etc). This is an easy way to improve productivity, especially if you can share them with other team members.

4.16 Summary

This chapter covered many commands that will make you productive in PyCharm. If you start using these commands and the commands from the previous chapter, you will find yourself leveraging much of the power of PyCharm. We explored using PyCharm to clean up code, make suggestions, auto-complete, and explore documentation. PyCharm has a very powerful and configurable editor. Because you will be using it a lot, it pays dividends to invest some time in learning the features of it.

4.17 Commands

- *"Show Intention Actions"* - (alt-enter or clicking on the light bulb)
- *"Reformat Code"* - (alt-command-L or ctrl-alt-L),

- *"Code Cleanup..."*
- *"Live Template"* - (type start of template and TAB)
- *"Parameter Info"* - (command-p or ctrl-p)
- *"Basic Completion"* - (type start of variable and TAB)
- *"Code Completion"* - (ctrl-space)
- *"Find Action..."* - (command-shift-A or ctrl-shift-a)
- *"Refactor Method"* (command-alt-m or ctrl-alt-m)
- *"Rename"* - (shift-f6)
- *"Lens Mode"* - (hover over scrollbar)
- *"Quick Documentation"* - (ctrl-j on Mac, ctrl-q on Windows/Linux)
- *"Find Usages"* - (command-F7 or alt-F7)

4.18 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/2-the-editor` to add a feature to a class.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/2-the-editor` to clean up code for PEP8 compliance.
3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/2-the-editor` to remove unused imports.
4. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/2-the-editor` to add documentation to a method.

Chapter 5

Source Control

Once you start working with teams, you will need some sort of source control. This makes group development much easier as you can branch, merge, and commit changes. PyCharm has built-in support for source control for many of the common systems. If you have ever wanted a UI for your favorite source control system, you are probably in luck as the common ones are supported by PyCharm. There is support for Git, Subversion, Mercurial, Perforce, and CVS. Beyond these, you can find plugins for PyCharm adding additional capabilities such as Microsoft's TFS.

5.1 Accessing Source Control

Let's look at an example with Git. If you checked out one of the projects from GitHub in a previous chapter that was a Git project. Git is very common these days, so we will focus on that. When you have a Git project open, you will see that there is a "VCS" menu that has many commands for integrating with source control. Let's ignore the menu option for now and look in the bottom right corner of PyCharm. You will see a button that says "Git: master". This button is also called the Git Widget.

If you click on the button, PyCharm brings up a menu to make a new branch or tag or interact with other local and remote branches.

Along the top of PyCharm is another set of buttons. You will see "Git:" to the left of theses buttons. There is a button for *"Update Project"* (command-t, or ctrl-t), *"Commit"* (command-k, or ctrl-k), *"Compare with the Same Repository Version"*, *"Show History"*, and *"Revert"* (command-alt-z, or ctrl-alt-z). In the project file browser, the files are also colored based on their status. Blue means the file has been updated, red means the file is not in version control, and green means that the file has been added to version control but not checked in.

If you want to look at any changes you have made, click on the "Compare with the Same Repository Version" button. That will show a screen with two code panes. On the left is the pristine copy from the repository, and on the right is your version. Any differences will be highlighted.

If you had updates that you wanted to share, a common workflow is to first run "Update Project" to pull and auto-merge the latest changes. Then you could "Commit" your changes. This will open up a "Commit Changes" window. You can select what changes you want to include and provide a commit message. There are also useful pre-commit changes that you can make including reformatting code, or optimizing imports. Those can be helpful depending on your groups coding standard. The "Commit" button here is fancy and has multiple options. Hitting "Commit" in Git will update your local repository. If you want to push your changes up to a remote repository, click

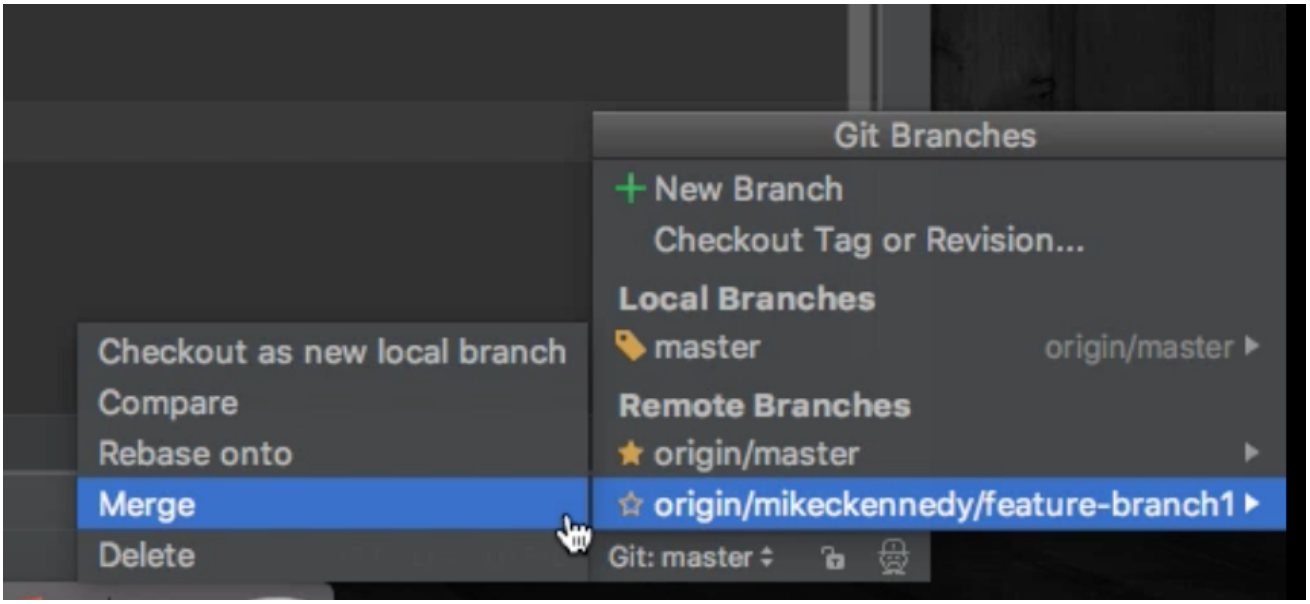


Figure 51: Figure illustrating clicking on the Git button in the lower right. Notice that the button itself indicates the current branch. When you click on it, there are menu options to create a new branch or tag as well as submenus for interacting with other branches.

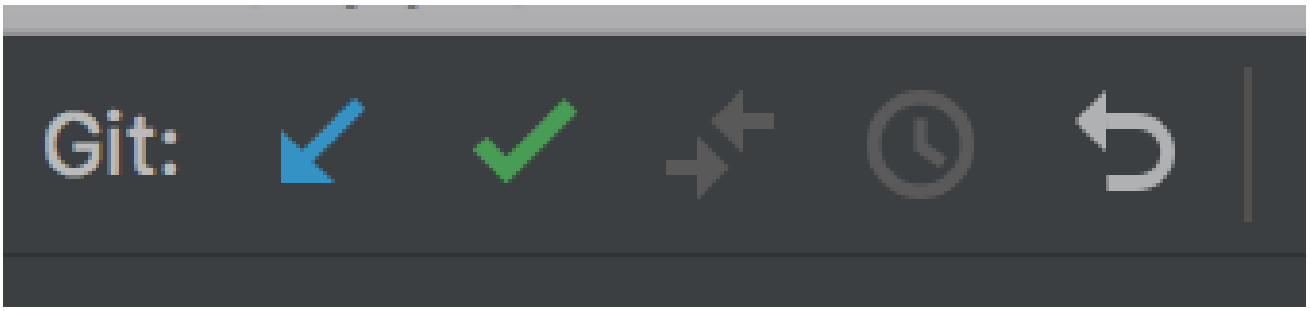


Figure 52: Figure illustrating buttons for updating the project, committing changes, comparing with the repository version, showing the history, and reverting. These are found in the upper right of the editor.

"Commit and Push...". If you decide to push, a "Push Commits" window opens with opens for including tags or forcing a push.

PyCharm has some nice options in this UI that are painful or annoying from the command line tool. If you want to remove a change from a commit, you need to unselect the checkbox next to the change. This lets you pick and choose the changes you want to commit.

Another cool feature of Git integration is the *"Annotate"* command (VCS -> Git -> Annotate). This allows you to see all the check-ins to a file. If you run this command on the left is a list of who committed the changes and when. If you click on a change, another screen showing the differences will appear.

5.2 More Source Control

Let's dive in a little deeper. In addition to the buttons that let us view differences and the file color that indicates the status of the file, PyCharm has some other features. In the gutter of any changed

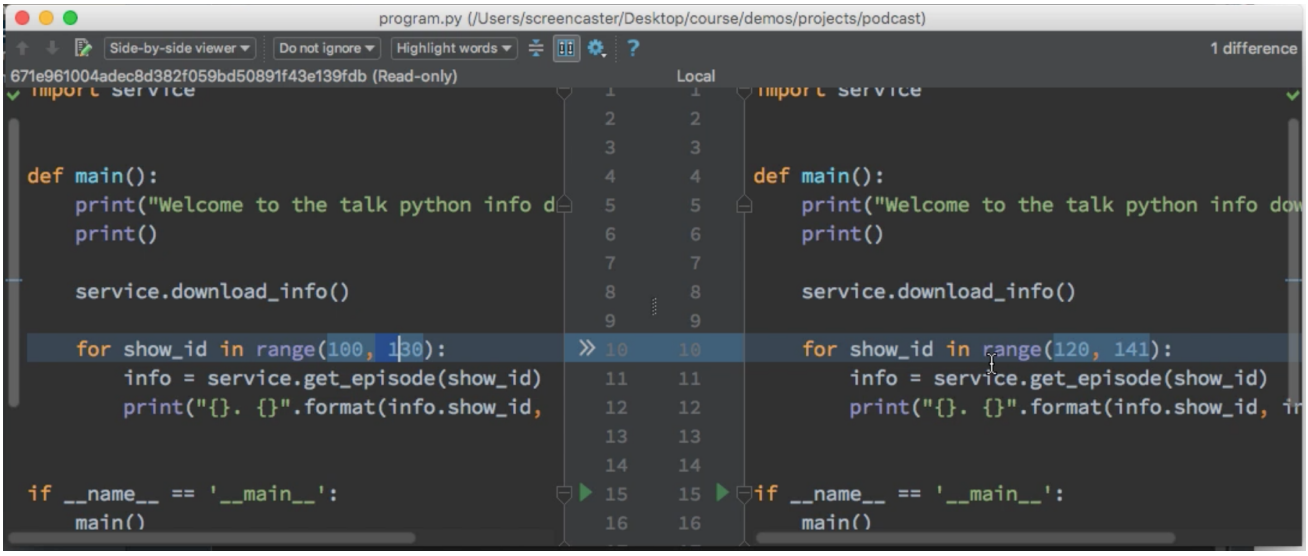


Figure 53: Figure illustrating the Compare with the Same Repository Version screen. On the left is the repository version and on the right is the local version.

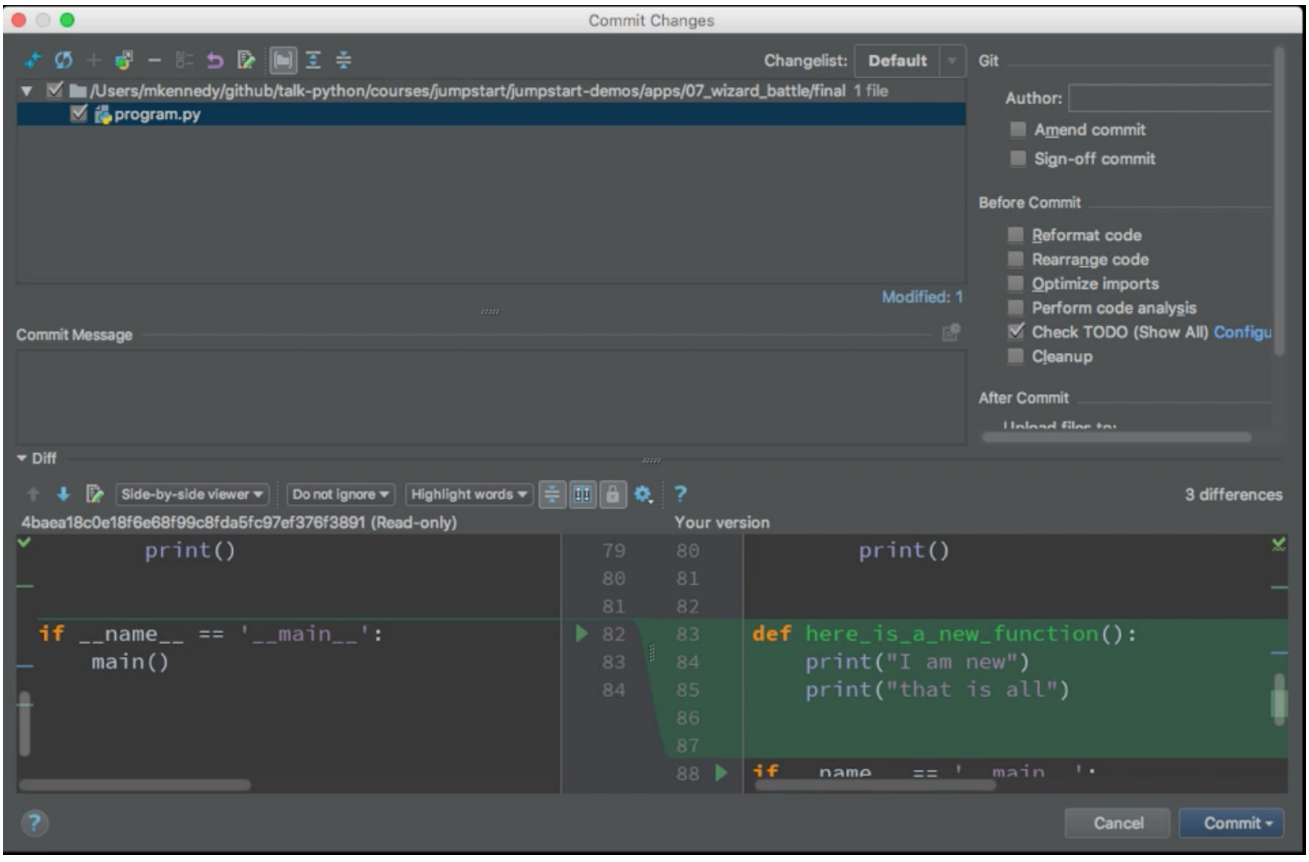


Figure 54: Figure illustrating committing work to a repository. Note that you can check what files to include and provide some context of the changes in the "Commit Message" text box.

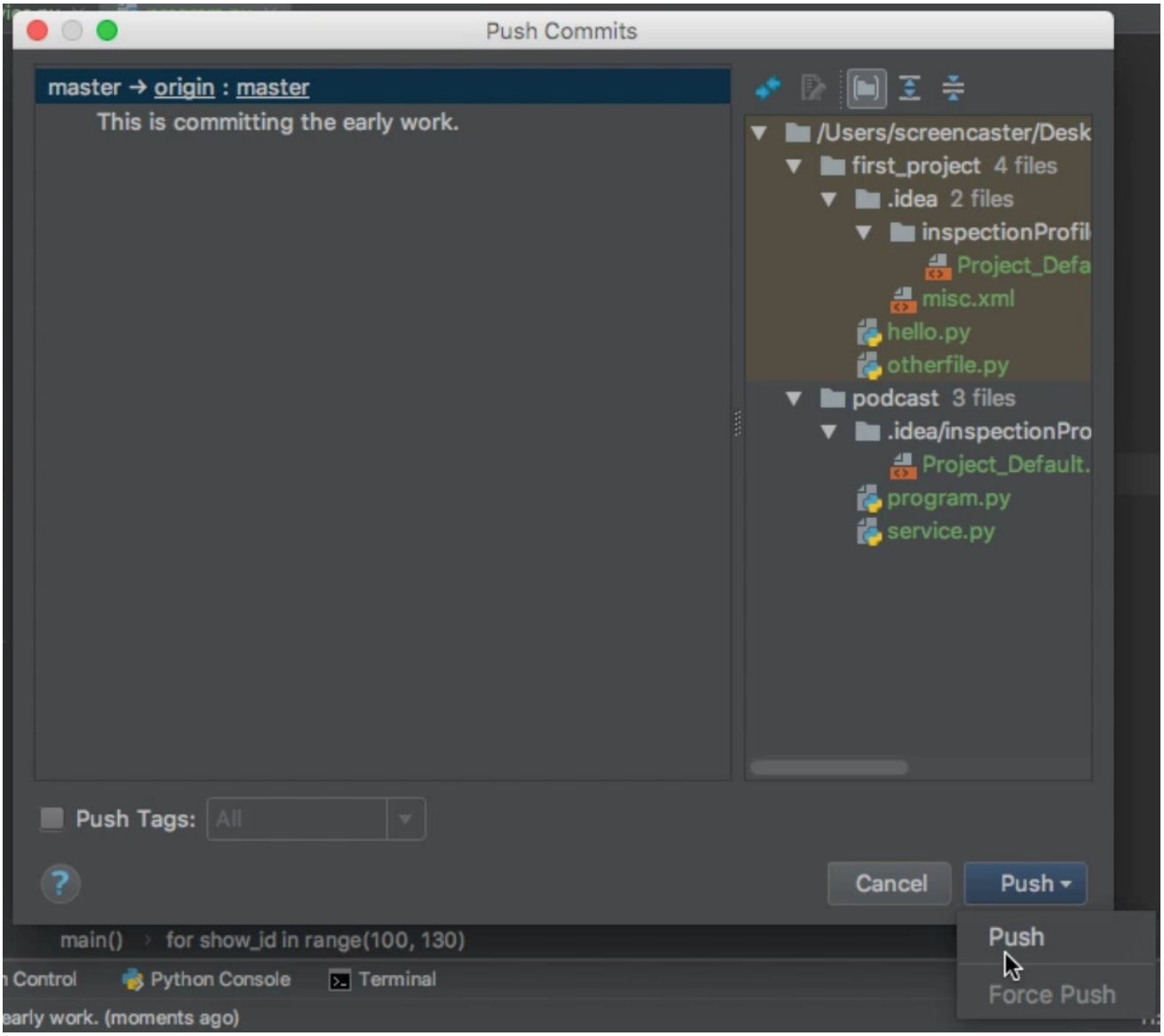


Figure 55: Figure illustrating the "Push Commits" window. From here you can include tags, or force a push.

code is a highlight. If the gutter color is green, then that is a new line that was not in a prior commit. If the color is blue, it indicates that the line was in a prior commit but has since been updated.

If you click on the highlight in the gutter some buttons will appear. You can jump to the *"Previous Change"* (ctrl-shift-alt-UP), the *"Next Change"* (ctrl-shift-alt-DOWN), *"Rollback Lines"* (command-alt-z, or ctrl-alt-z), *"Show Diff"* (command-d or ctrl-d), *"Copy"* (command-c, or ctrl-c), and *"Highlight Words"*. Clicking on a blue gutter will also show the difference inline. Note that the copy command will copy the original version, not the current version.

PyCharm will also insert an arrow pointing to the right. That indicates that a section was deleted from the repository version. These are nice unobtrusive hints for you to see what has changed in the file you are editing.

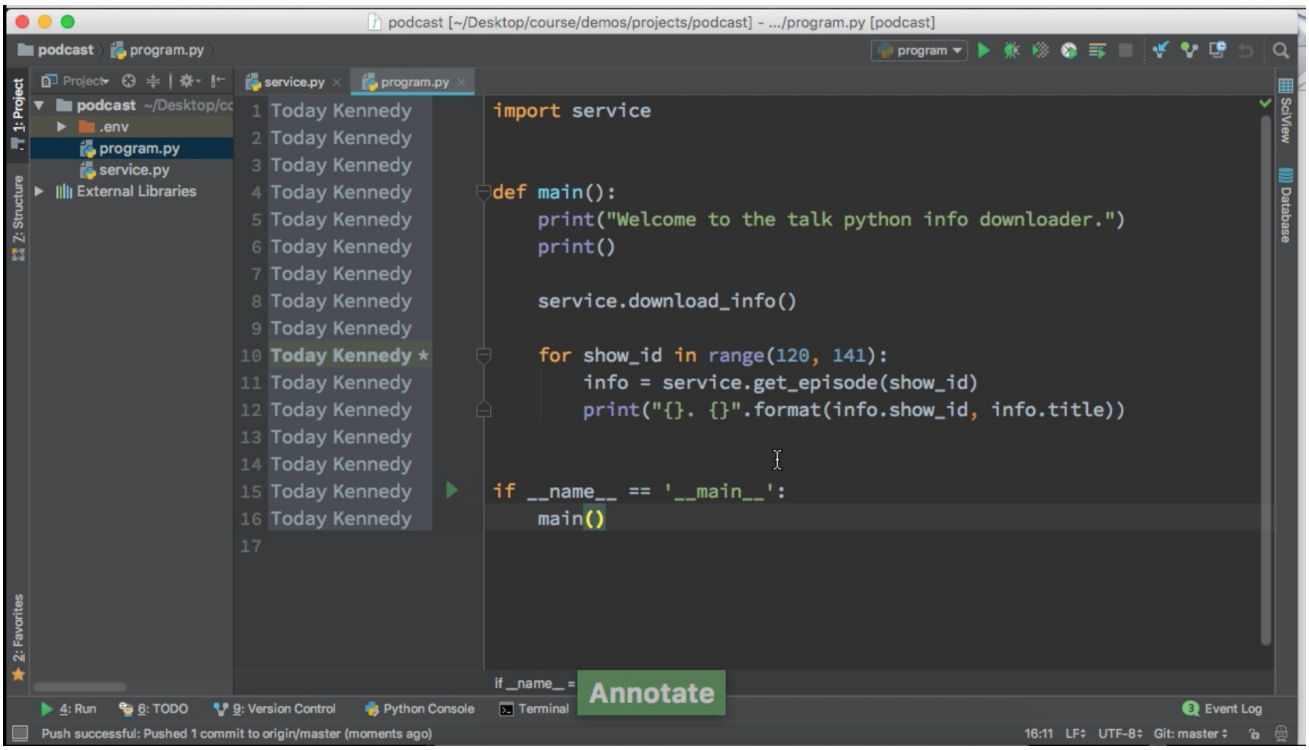


Figure 56: Figure illustrating the annotate view. On the left-hand side is a list of changes made to a file.

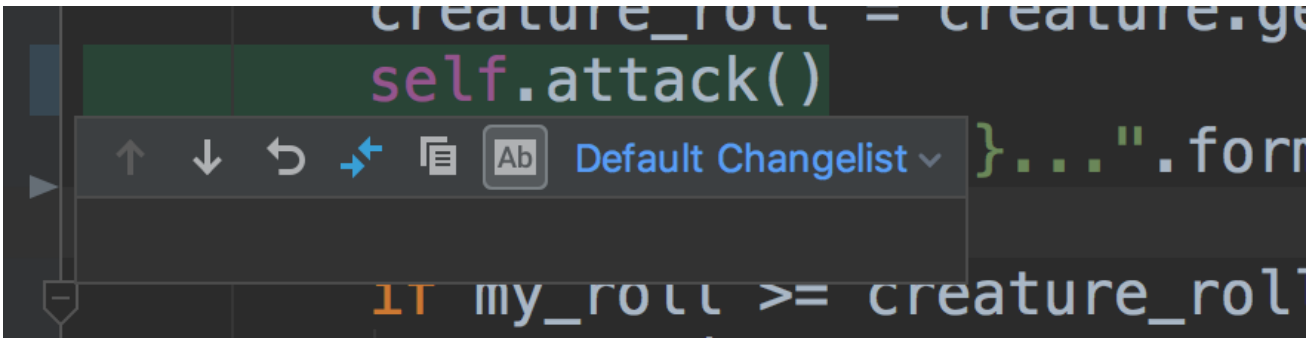


Figure 57: Figure illustrating the highlight in the gutter and the context menu that appears when you click on it. Also shows the arrow where code was deleted.

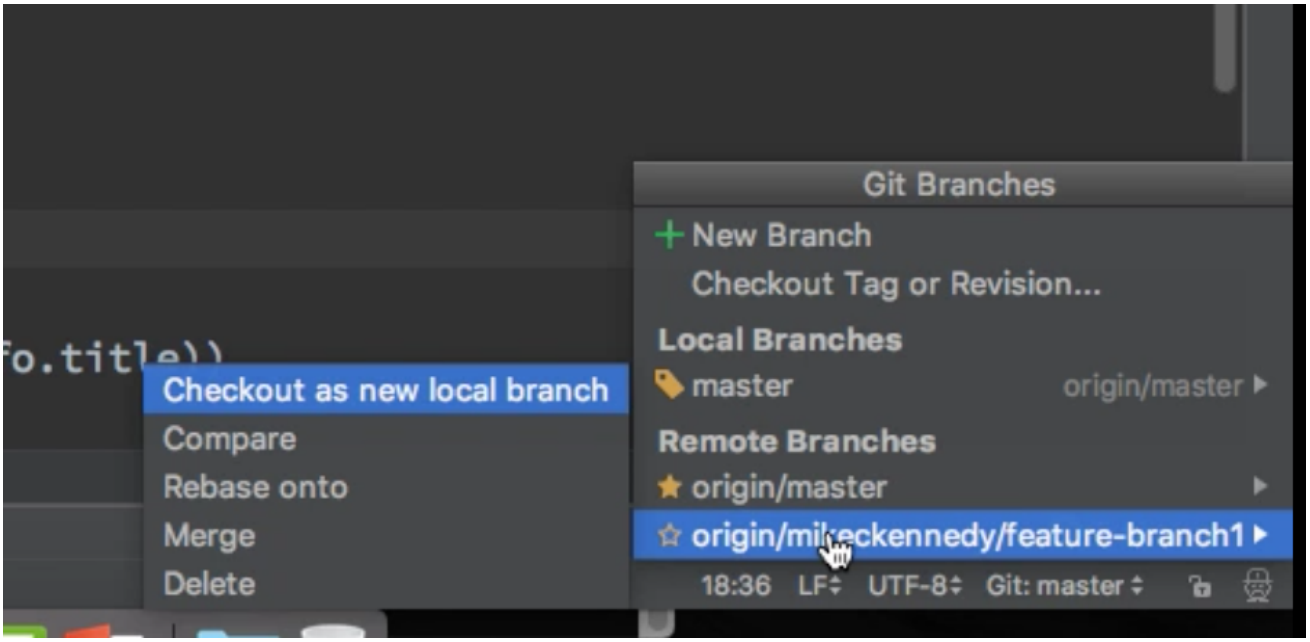


Figure 58: Figure illustrating checking out a remote branch.

5.3 Branching

In this section, we will look at creating a new branch and working with it. Recall that in the lower right-hand portion of PyCharm there is a Git widget button. If you click on it, you will have options for working with branches. If you wanted to work on a remote feature branch, you can click on the remote branch name in the popup and then click "Checkout as new local branch". At this point, your project will be updated with the code from that branch. You can follow the previously described workflow to commit into the remote branch.

Once you have created a new local branch, you can quickly toggle between a branch by clicking on the local name and then hitting "Checkout". If you have local changes, you have two options. One is to "Force Checkout", which will throw away local changes. The other is "Smart Checkout", which will stash the changes, checkout the branch, and then unstash the changes.

PyCharm will also let you merge branches or rebase them. Checkout the branch that you want to update with the changes. Click on the Git widget and select the branch you want to get the changes from. You can choose "Merge into Current", or "Checkout with Rebase". If there are conflicts, you will have the chance to merge them.

5.4 Local history

In addition to integration with source control systems, PyCharm also has a "Local History". This is independent of Git, or any other source control system managing your project's files. Local history is a detailed timeline of all changes that have occurred to the file. It is built into PyCharm. Local history is handy if you happen to want to rollback to an intermediate state of a file that wasn't managed by source control. You can access it by right-clicking on a file and selecting "Local History" -> "Show History".

If you find yourself in a bind where you have created and deleted large amounts of code without checking them in, do not fret. Local history can help you. You can even generate patches and share

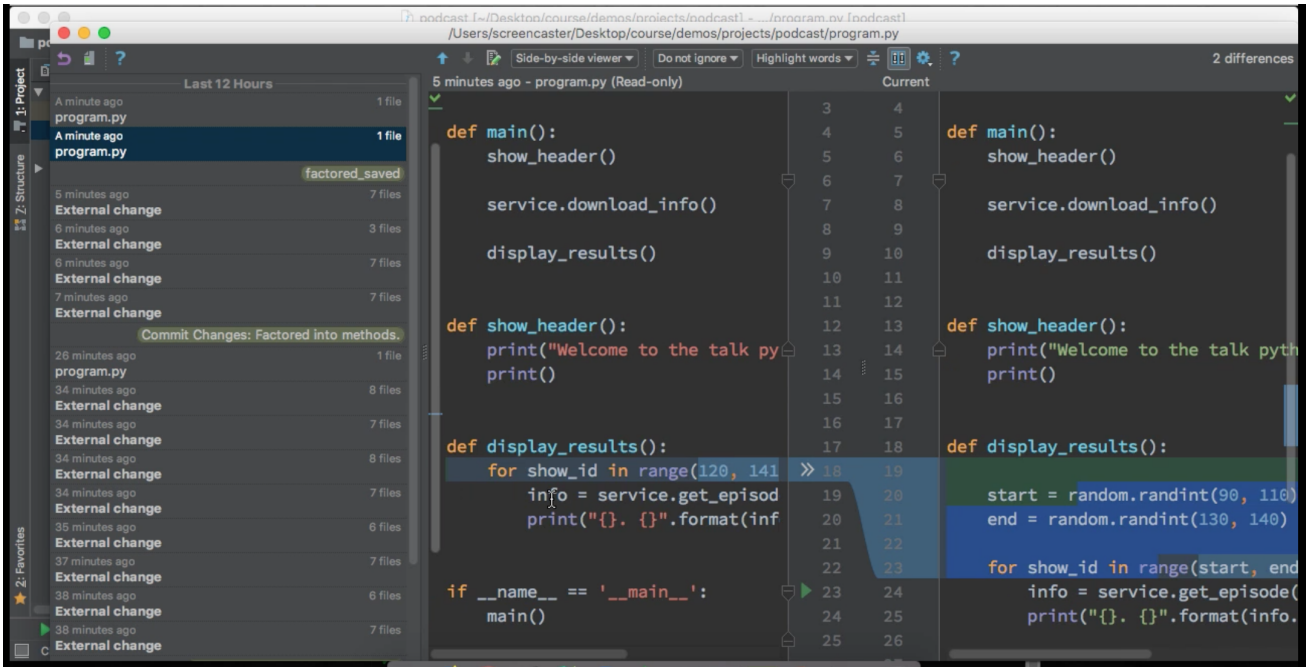


Figure 59: Figure illustrating Local History. Independent of source control, PyCharm manages its own version of the changes to the project.

them with others. Do be aware that PyCharm caches this information on your machine for the past five days by default. If you want to change this value, you will need to open the PyCharm Registry and change the `localHistory.daysToKeep` parameter and restart PyCharm.

5.5 PyCharm and Git Flow

Git is a powerful tool with many options. A simple suggested workflow for Git is called Git Flow[19]. The gist of this workflow is to have two main branches, master and develop. The master branch is only pushed to when we have a confirmed release. The develop branch has work for the next release. There are other branches that can be used for creating a fixture, testing a release, or doing a hotfix. However, these are short-lived branches and will be merged back into the develop branch (which will eventually be merged into master when a release is cut).

Let's walk through adding a new feature. First, we will create an issue in GitHub for the feature. Then make a new branch with the feature. Finally, we will merge that feature back into master. Let's assume that we have filed a bug in GitHub. Let's use PyCharm's integration with GitHub so we can view the tasks for a project. To configure it go to "Tools" -> "Tasks & Contexts" -> "Configure Servers...".

After you have configured GitHub, there will be a "Default task" button in the upper right of PyCharm. If you click on "Open Task..." it will show the issues in the project (calling them "tasks"). This will open the "Open Task" window and allow you to create a new branch for addressing this issue. After clicking "OK" you will see that PyCharm has created a new local branch for you. The Git widget will no longer display "master", but the name of the branch.

[19]https://nvie.com/posts/a-successful-git-branching-model/

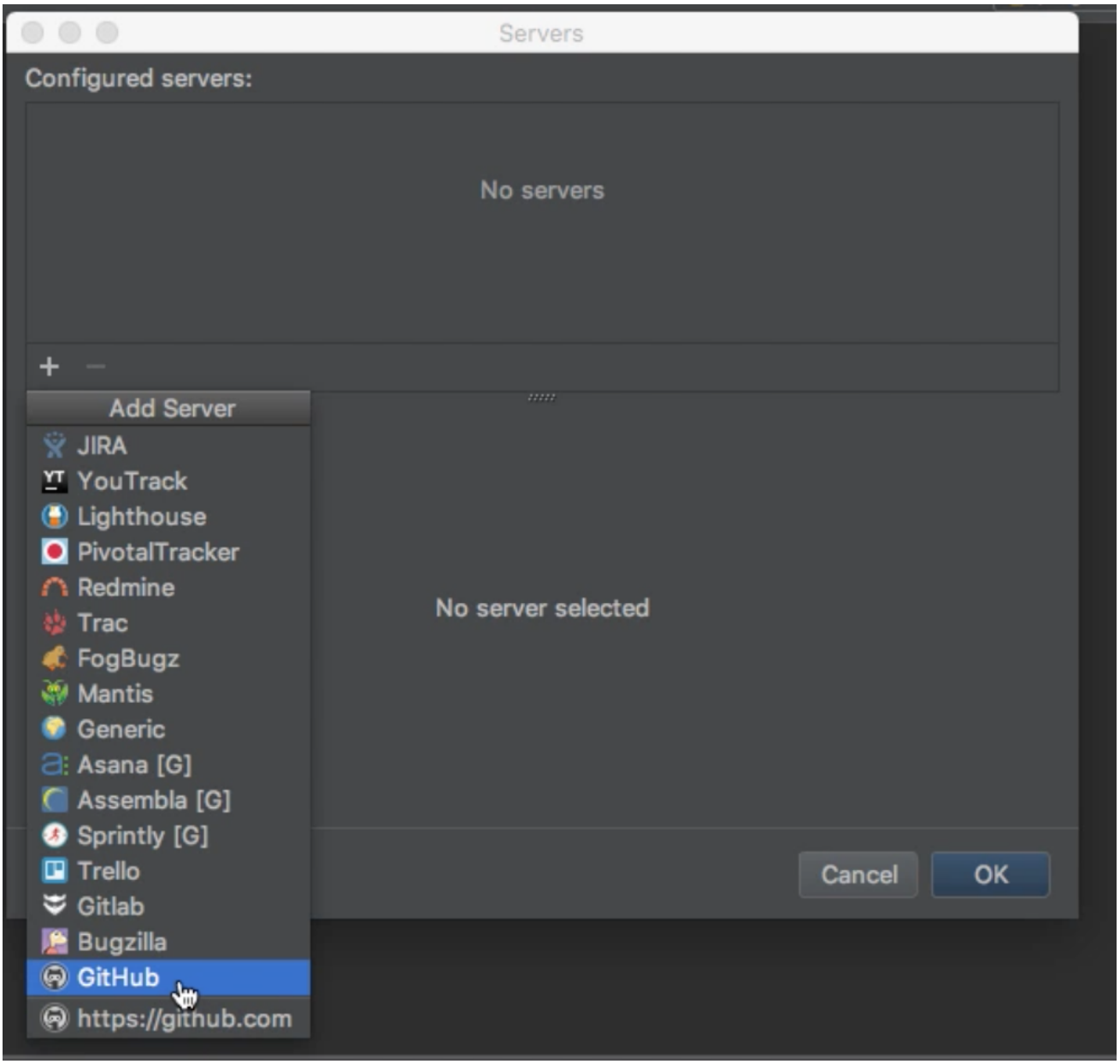


Figure 510: Figure illustrating adding a server for GitHub.

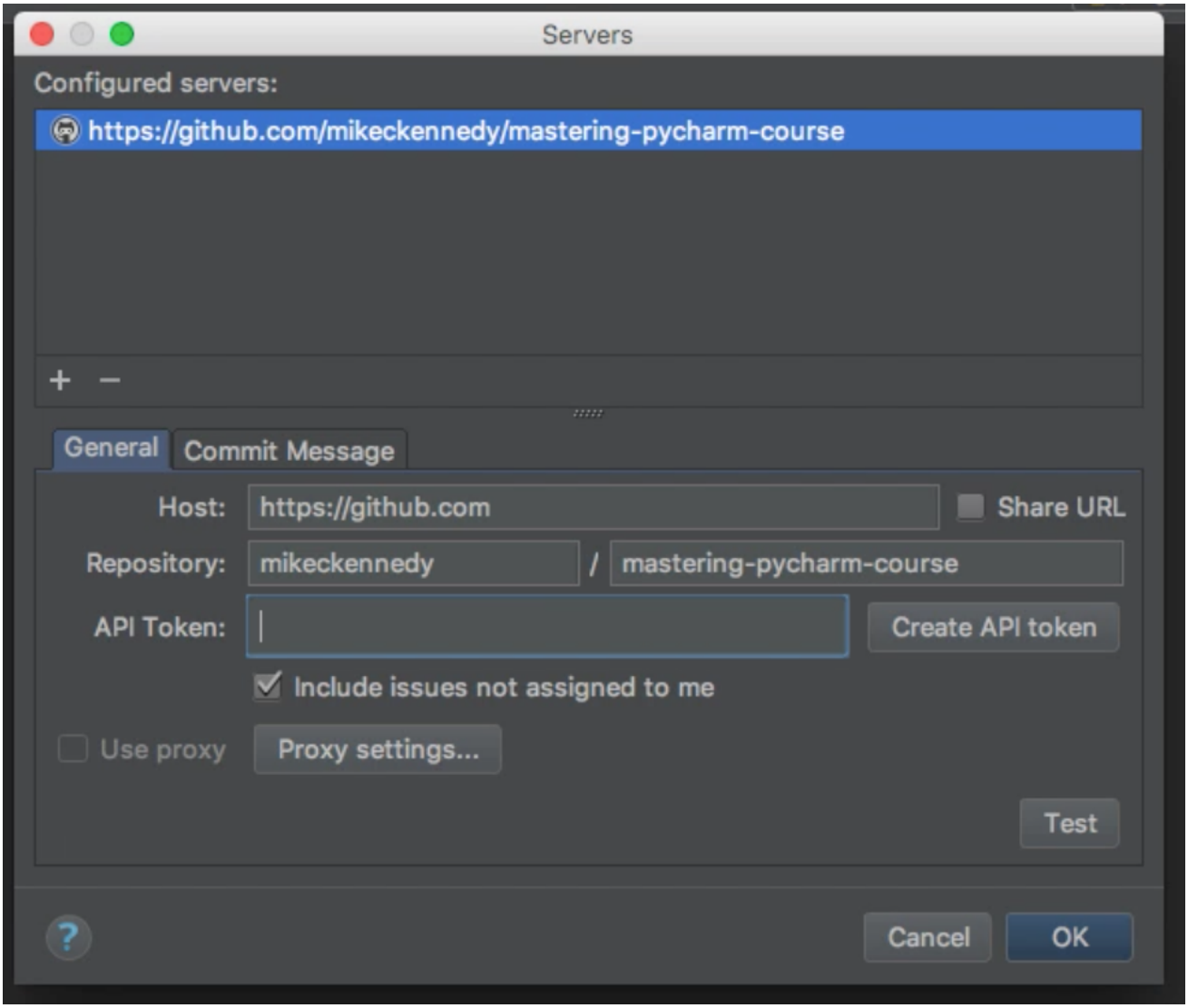


Figure 511: Figure illustrating configuring the server for GitHub.

At this point, you can code away. After you have added the code necessary to address the GitHub issue, you need to close the task. To do that run the *"Close Active Task"* (alt-shift-w). When you hit "OK", you will be presented with the opportunity to push the branch. If you push the branch, PyCharm will commit the branch on GitHub and put you back on the latest version of master. You will need to go back to GitHub to create a pull request and merge it and close the issue.

By using the Task feature in PyCharm you will have access to the issues for a project. You can also easily create new branches based on them and follow a Git Flow type workflow if desired.

5.6 Pull Requests

PyCharm supports creating GitHub pull requests as well. If you are in a GitHub project you will have the option in "VCS" -> "Git" -> *"Create Pull Request"*. You will have to login with you credentials or a "Personal access token". To create a token on GitHub go to Settings -> "Developer

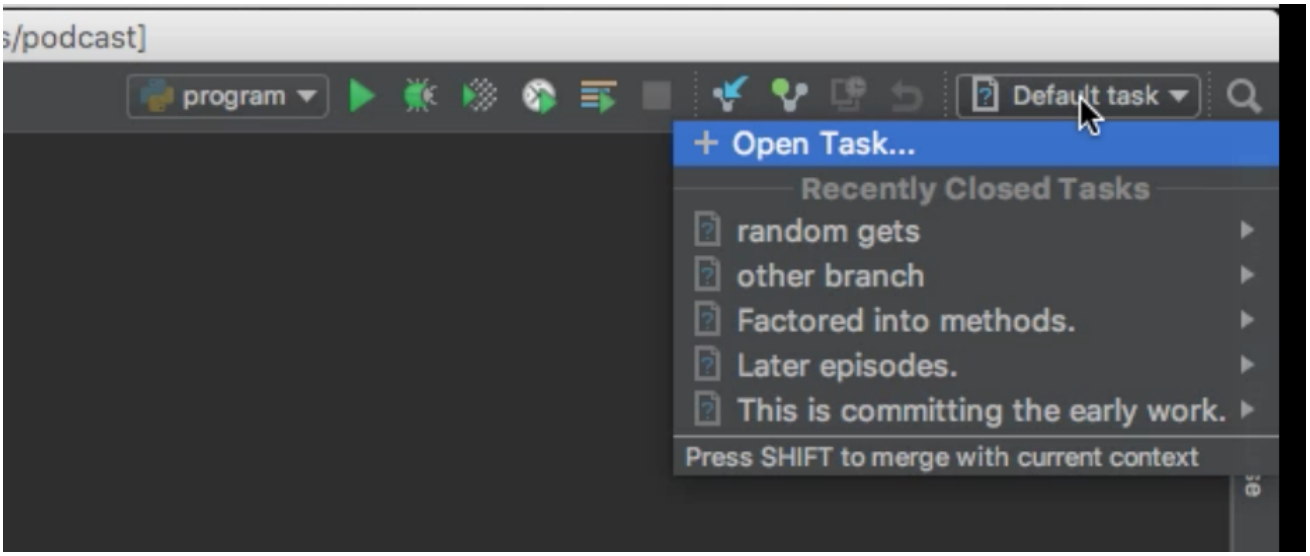


Figure 512: Figure illustrating successful configuration with GitHub and tasks button appearing in upper right.

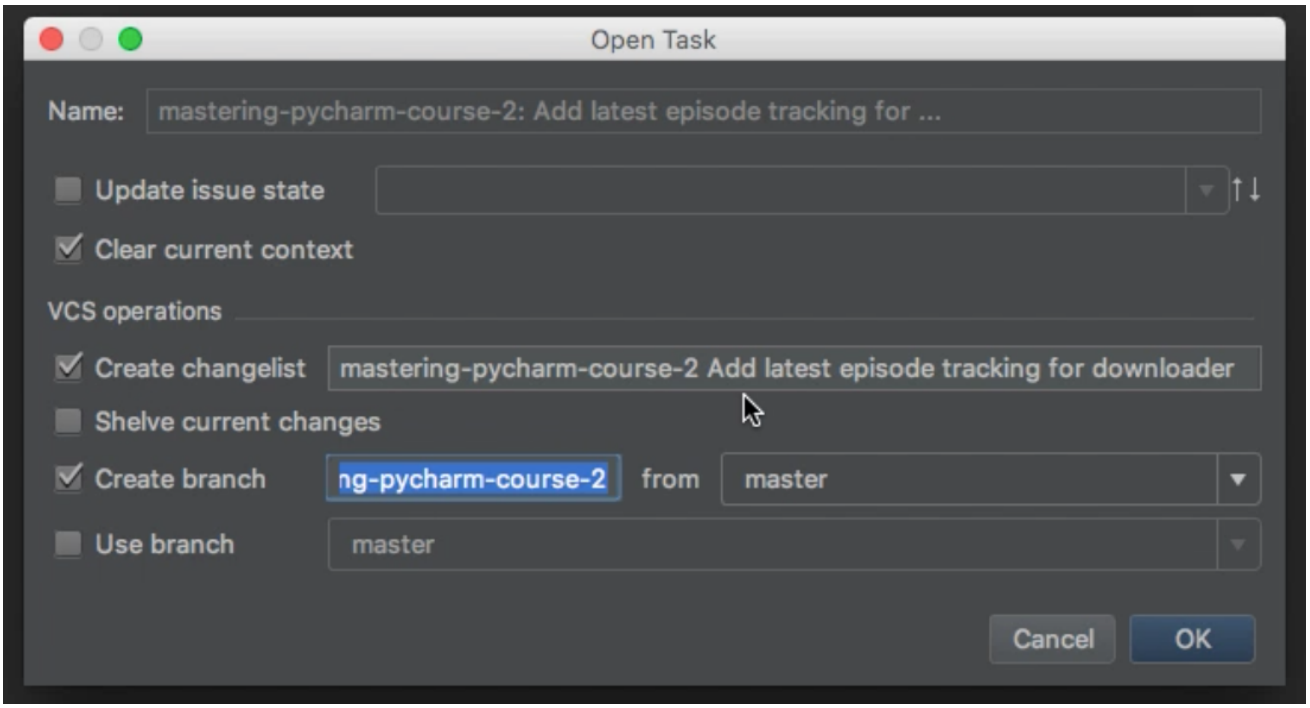


Figure 513: Figure illustrating the Open Task window for creating a new branch for a GitHub issue.

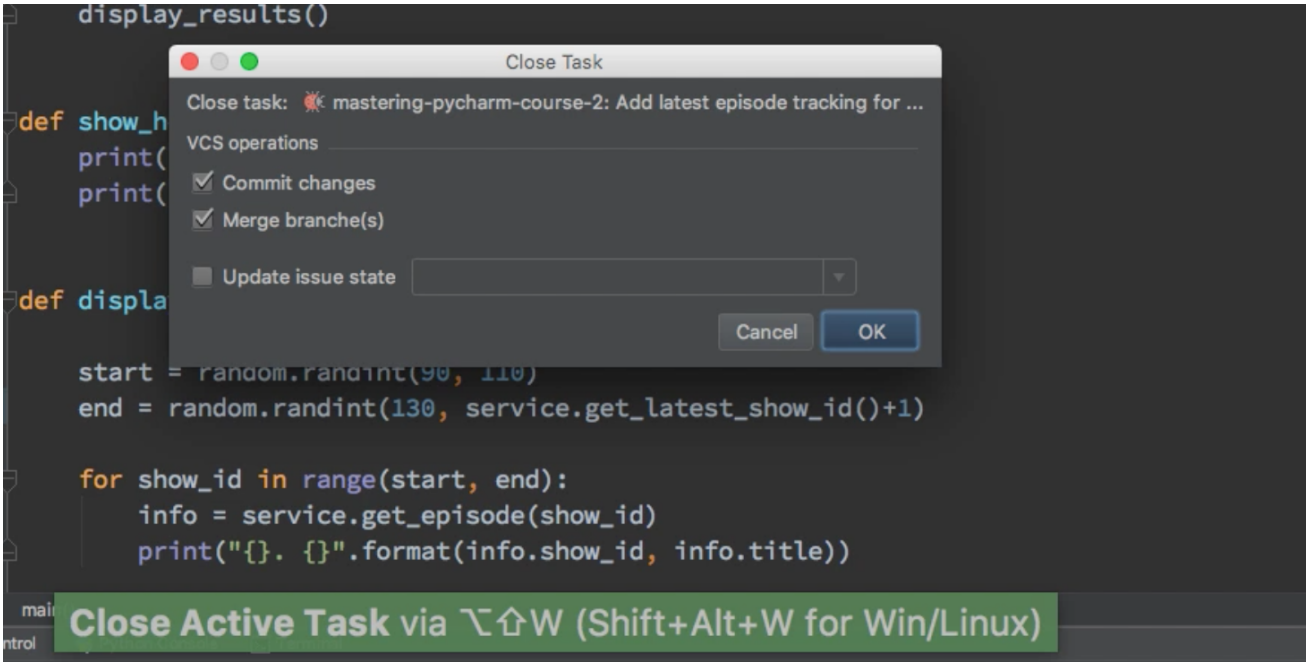


Figure 514: Figure illustrating closing the active task.

Settings" and click on "Generate new token". After logging in, you should have the ability to create a pull request. Fill in the appropriate information and click "OK".

You can also *"View Pull Requests"*. When you run this command, PyCharm will show a "GitHub Pull Requests" tab along the bottom of the screen. It will list all of the pull requests on the project. You can filter them, open them in GitHub, view the diff, or create a location branch from the pull request. To run the *"Create New Local Branch..."* action, right click on the pull request and select that option.

5.7 Quick gist

Finally, PyCharm has the ability to create a *gist*. A gist is a small snippet of code that GitHub will host for you. This can be publicly searchable or private. It is often used when sharing a temporary item that is larger than would fit in a text or chat that would not be lost in a long email thread.

You can create a gist of a whole file or of a portion of a file. You can also create a gist from the terminal output. If you want to only use a portion of text, highlight just that section and then right-click. If you want to include the whole file, right-click on the file. At the bottom of the context menu is an option *"Create Gist..."*.

The create gist command will open a window where you can set the preferences on the gist. There are options for editing the filename, changing the description, and indicating if it is secret or anonymous. After you have created a gist, you can share the GitHub link to it with others.

5.8 Conclusion

In this chapter, we explored source control with PyCharm. PyCharm also has the notion of local history in case we forget to check in code or aren't using source control. We saw some examples with the popular Git source control system.

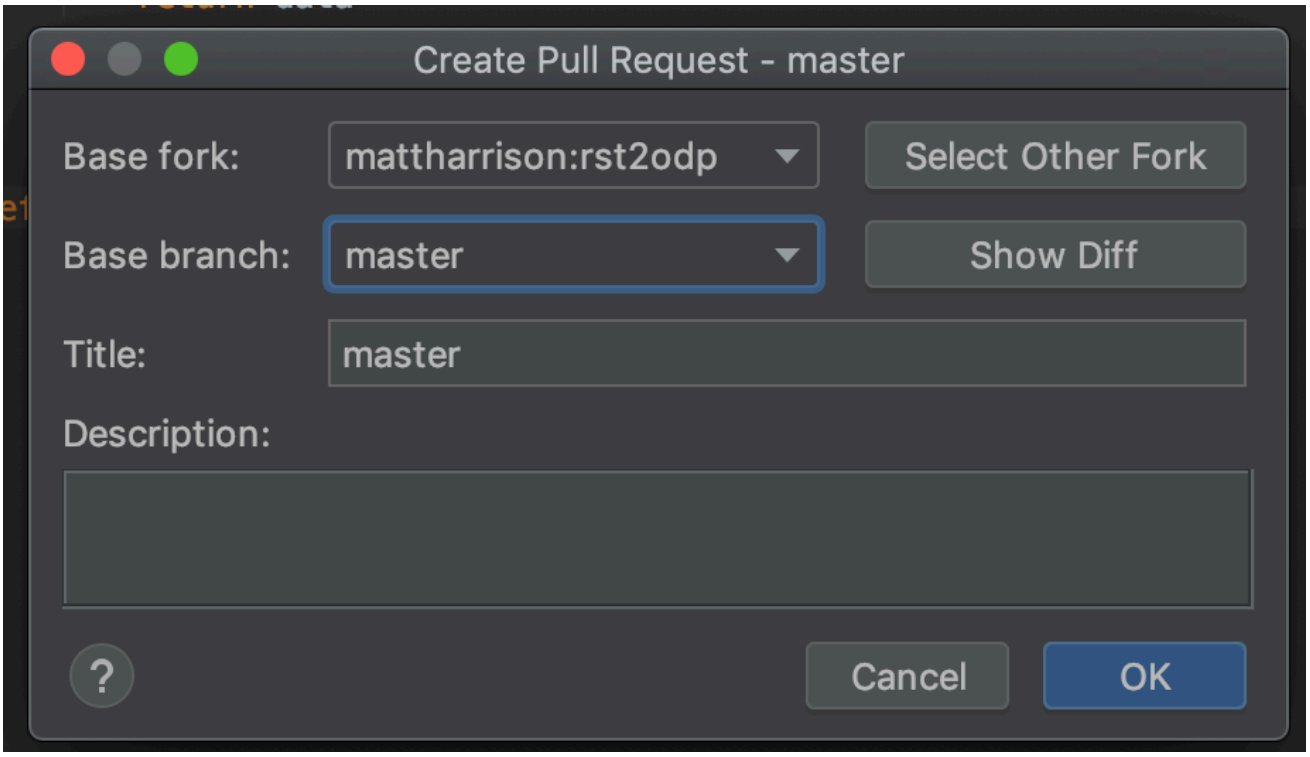


Figure 515: Figure illustrating creating a pull request.

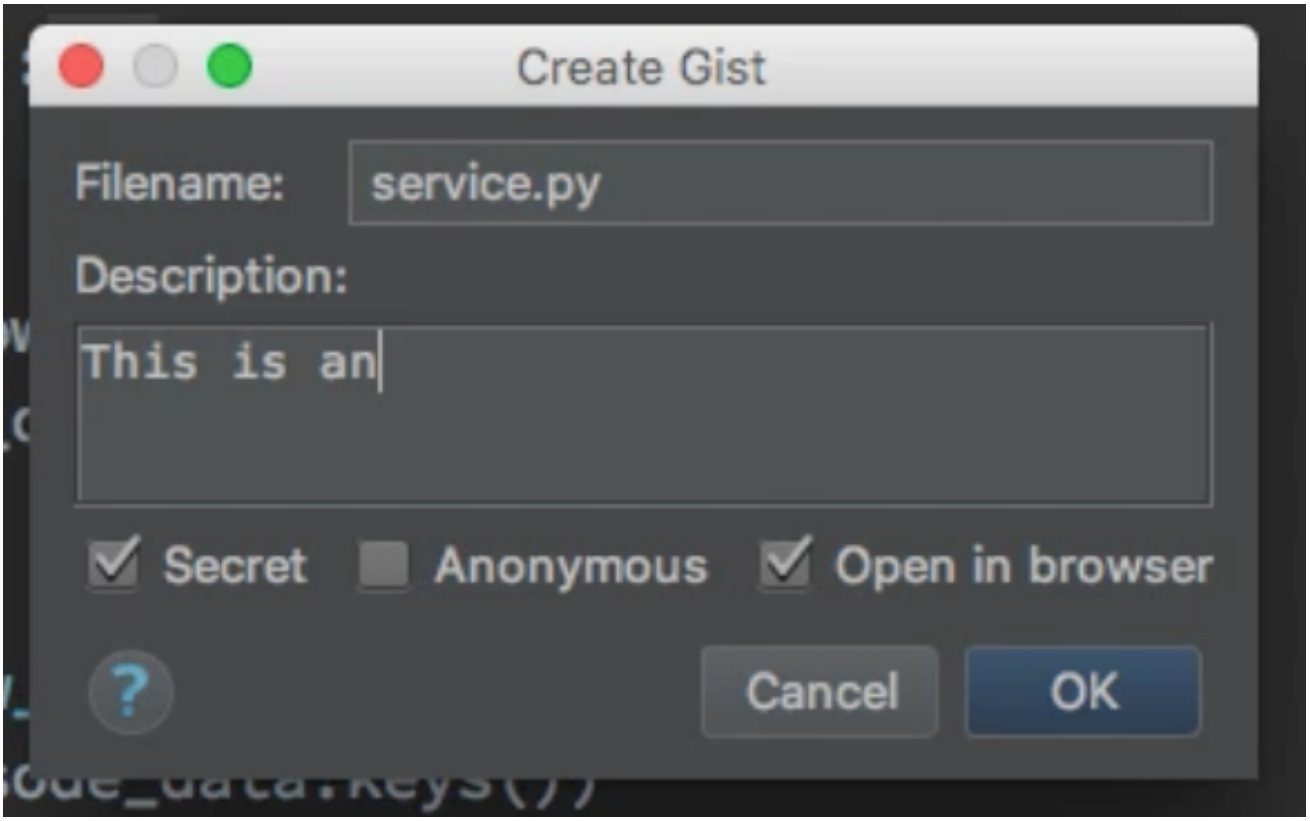


Figure 516: Figure illustrating creating a gist.

We specifically looked at the Git integration as well as GitHub integration. As soon as a project is associated with source control, there will be a Git widget in the lower right and buttons to perform common source control actions in the upper right. Using the Git widget we can switch branches and perform other common branching operations. We can use task integration with GitHub to create new branches based on issues and follow a Git Flow process.

5.9 Commands

- *"Update Project"* - (command-t or ctrl-t)
- *"Commit"* - (command-k or ctrl-k)
- *"Compare with the Same Repository Version"*
- *"Show History"*
- *"Revert"* - (command-alt-z or ctrl-alt-z)
- *"Annotate"* - (VCS -> Git -> Annotate)
- *"Previous Change"* - (ctrl-shift-alt-UP)
- *"Next Change"* - (ctrl-shift-alt-DOWN)
- *"Rollback Lines"* - (command-alt-Z or ctrl-alt-z)
- *"Show Diff"* - (command-d or ctrl-d)
- *"Copy"* - (command-c or ctrl-c)
- *"Highlight Words"*
- *"Close Active Task"* - (alt-shift-w)
- *"Create Pull Request"*
- *"View Pull Request"*
- *"Create New Local Branch..."*
- *"Create Gist..."*

5.10 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/3-source-control` to try out Git in PyCharm.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/3-source-control` to try out local histories in PyCharm.

Chapter 6

Refactoring

6.1 What is Refactoring?

In this chapter, we are going to discuss refactoring support in PyCharm. Refactoring is much more than doing a search and replace across some files. PyCharm parses your files into an abstract syntax tree. This allows it to fully understands the code in context so it can enact powerful changes on your entire project. Want an example? Let's say you have two classes with a method `to_json`. You want to rename one of them. PyCharm knows the difference between the two symbols and only renames the one you are refactoring. Symbols are not strings.

In Wikipedia we read the following definition of refactoring:

> Code refactoring is the process of restructuring existing computer code—changing the factoring—without changing its external behavior. Refactoring improves nonfunctional attributes of the software. Advantages include improved code readability and reduced complexity; these can improve source-code maintainability and create a more expressive internal architecture or object model to improve extensibility.

We will see that PyCharm has excellent refactoring capabilities.

How do we know when to refactor? You may have run across the idea of *code smells*. These can be a powerful guide to when and how to perform a refactoring. A code smell is a term coined by Kent Beck to describe that there is something off-putting (or an unpleasant scent) about the code, often evidence of a deeper problem in the system.

Here are some common code smells. Duplicated code that exists in more than one location. Large classes that have many methods and responsibilities. Lazy classes that do very little. A function or method with too many parameters. Wikipedia lists some other code smells[20].

If you have duplicate code, that could be a sign that you need to create a function or method. This function could encapsulate the code and limit duplicating bugs or facilitate easier changes in the future.

A large class typically does too much. You can break it up into smaller classes that do a single thing and then stitch the classes together using composition.

The opposite of a large class is the freeloader or lazy class. If it isn't doing anything, it should be removed or perhaps replaced with a function or inlined into where is it used.

[20] https://en.wikipedia.org/wiki/Code_smell

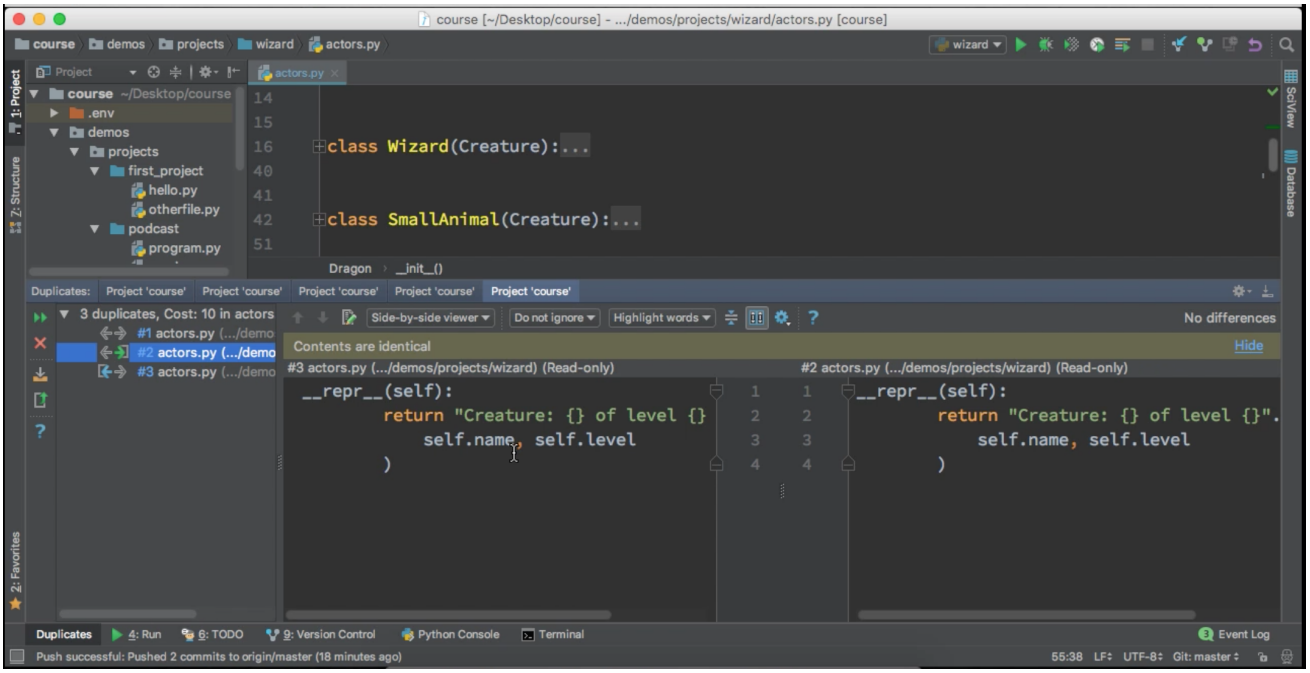


Figure 61: Figure illustrating the Duplicate Tool Window on the bottom of PyCharm.

Having too many parameters can be confusing. It might be hard to remember what each parameter is doing. It also might indicate that the function is doing too much, or the parameters should be grouped together into their own configuration class. In modern Python, this could be accomplished using Python 3.7's Data Class feature. For older versions, namedtuples are an alternative.

When you are thinking about refactoring keep your nose out for these code smells. They can give you hints as to when you should change the nonfunctional aspects of your code. Refactoring is an investment that can pay huge dividends in the future.

6.2 Finding Duplicate Code

Duplicate code is code that repeats itself in different places. Refactoring can be a remedy to this. However, you need to find the duplicated code before you can refactor. Finding duplicated code can be tricky. Sometimes code is very similar, but has slight differences. PyCharm can find these and let you determine how to deal with the code.

PyCharm has a *"Locate Duplicates"* command (Code -> "Locate Duplicates...") which will find duplicated code. If you run that command it will pull a sequence of windows. The first will ask you if you want to apply it to your project, a single file, or a custom scope. The next window will ask you which language you prefer to look for duplicates in. We will select Python, but remember that PyCharm understands other languages as well and if you are working on a web application, these come in useful. We will select Python as the language. There are some other options for choosing how PyCharm detects duplicates, but you can normally use the default settings. If you hit "OK" you will see the "Duplicate Tool Window" appear in the bottom of PyCharm.

In the Duplicate Tool Window, you can see the duplicates that PyCharm discovered. On the left-hand side is a tree with all of the duplicates. You can expand those and in cases where there

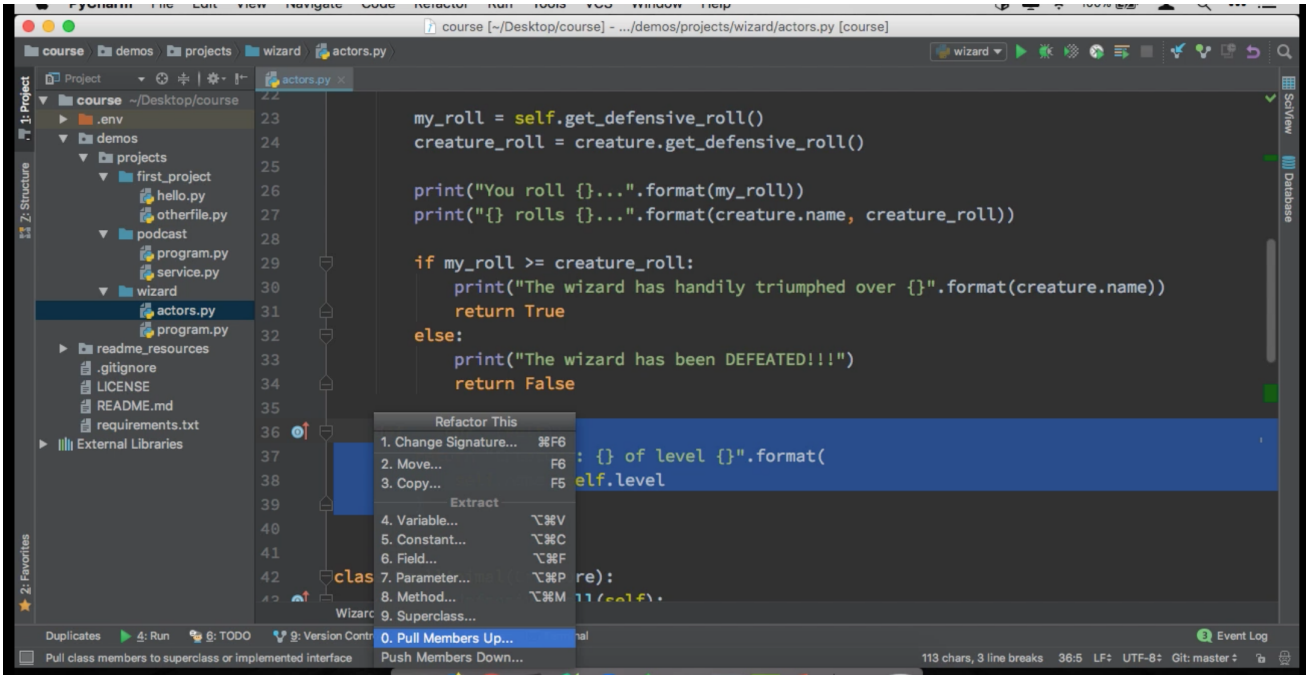


Figure 62: Figure illustrating the context menu that appears when "Refactor This" command is run.

are more than two duplicates, you can click on them to insert them into the difference view on the right.

Below is a class hierarchy, where `Wizard` and `Dragon` are subclasses of `Creature`. Both of the subclasses have the same method.

```
class Creature:
    def __init__(self, name, level):
        self.name = name
        self.level = level


class Wizard(Creature):
    def __repr__(self):
        return f"Creature: {self.name} level: {self.level}"


class Dragon(Creature):
    def __repr__(self):
        return f"Creature: {self.name} level: {self.level}"
```

If you had a case similar to the following code, you might want to refactor the `__repr__` method into the base class of `Creature`. To do so, select the method you want to refactor. This refactoring is called "pulling up", as we are moving a method from a subclass to a parent class. We would select the `__repr__` method in `Wizard` and the *"Refactor This"* command (command-alt-shift-t or shift-ctrl-alt-t) to bring up the refactoring popup. Select *"Pull Members Up..."* to start the refactoring.

A window will appear and you can select the class you want to move the method to. In this case select `Creature`. Hit "Refactor" and your code should look like this:

```
class Creature:
    def __init__(self, name, level):
        self.name = name
        self.level = level
```

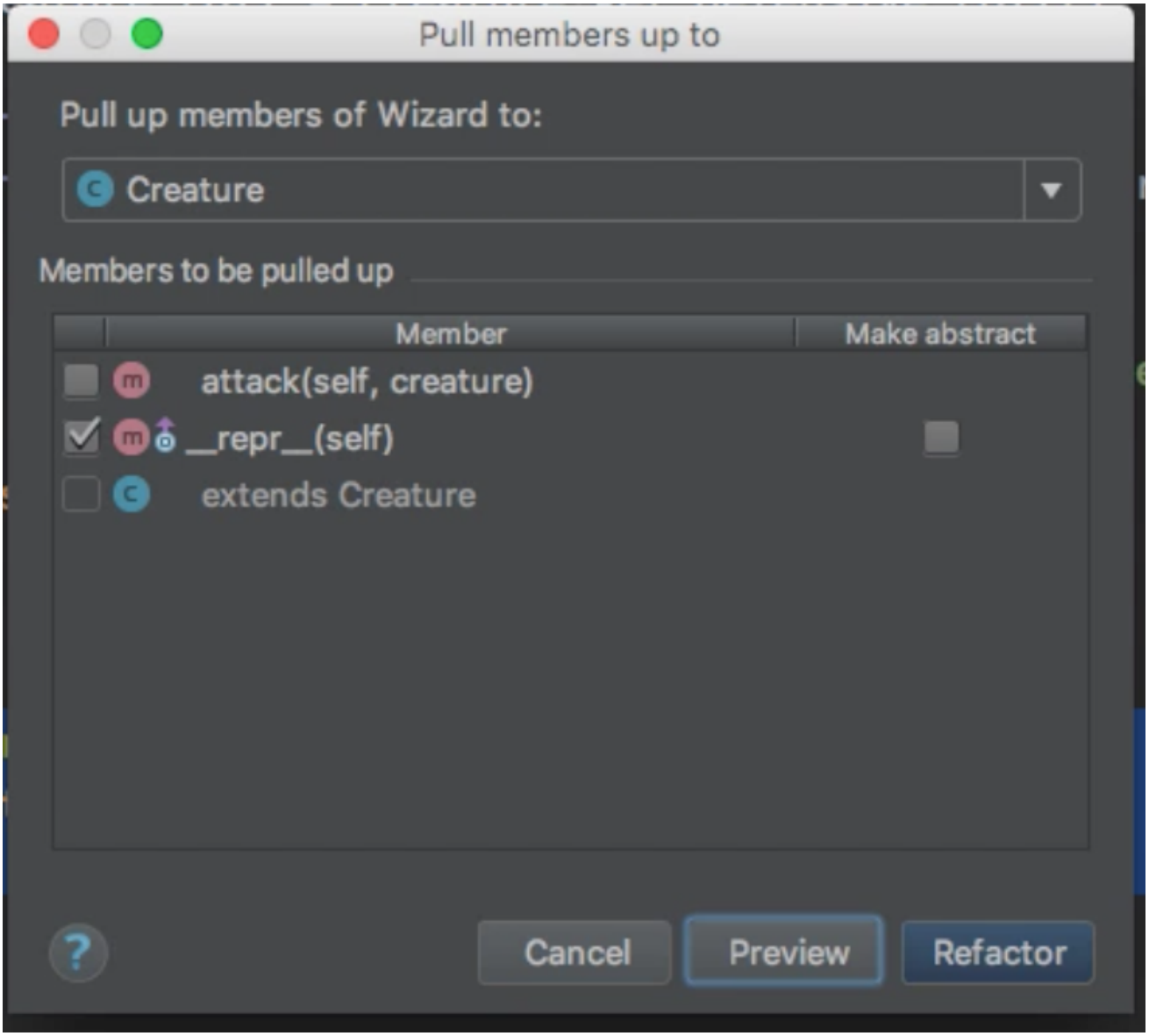


Figure 63: Figure illustrating "Pull members up to" window. You can select the methods you want to move up and the class you want them to land in.

```
    def __repr__(self):
        return f"Creature: {self.name} level: {self.level}"

class Wizard(Creature):
    pass


class Dragon(Creature):
    def __repr__(self):
        return f"Creature: {self.name} level: {self.level}"
```

Note that `Dragon` still has the `__repr__` method. The easiest thing to do is delete it. PyCharm is not quite confident enough to know that it should delete that as part of the refactoring. If you

re-run the find duplicates command after you've deleted `Dragon`'s `__repr__` method, you should see that this duplicate is now gone.

6.3 Renaming items

Another common refactoring is renaming symbols (variables, methods, types). One primary use of this refactoring is to make the code increasingly self-documenting. Consider a game with 3D elements that use the bounding box algorithm for hit detection. If you saw this code, would you be sure what it did without the comment?

```
# compute the bounding box of this shape
shape.compute()
```

I wouldn't be sure. However, if we rename `compute` to what it does more explicitly, you would know right away and you could drop the comment:

```
shape.compute_bounding_box()
```

These renamings can be scary if you don't catch all the uses. But that is not a problem with automated refactoring.

Back to our D&D game: If you have a variable called `player` in an adventure game and you wanted to change the name to `hero`, you could do this by doing a search and replace. That might work, but if you had another variable named `player_2` or a method called `attack_player`, those could be erroneously renamed to `hero_2` and `attack_hero`. Find and replace also might miss a rename in a different file. Even if you matched exactly on player, perhaps it is a different symbol, with the same name but from a different module.

The *"Rename..."* command (shift-F6) will perform the rename in a safe manner. You can select the variable you want to rename and run this command. If you don't want to memorize all of the hotkeys for refactoring, you can also access the refactoring commands from the "Refactor This" command.

When you are refactoring a variable, you type the new name and PyCharm will insert it in the correct places.

If you want to rename a method, say replace the `attack` method with `fight`, highlight an instance of the method (it can be an invocation or the method itself), and run the "Rename..." command on it. You can tell PyCharm to include renaming in comments and strings.

A "Refactoring Preview" window will appear in the bottom. You can see where references were found, and tell PyCharm to perform or cancel the refactor

You can also apply "Rename..." to files. PyCharm will ensure sure that any file importing the renamed file will be updated. Also, the VCS integration will allow you to easier track the file rename. Of course if you realize you made a drastic mistake, you can always *"Undo"* (ctrl-z or command-z) and PyCharm will revert the changes as on unit (including filename changes).

The next time you want to rename something before you reach for find and replace, consider using the safer, more intelligent "Rename..." command.

6.4 Introducing variables

Let's look at another common issue in code, *magic numbers*. Magic numbers are inlined literal numbers that had meaning to the author of the code, but might not be clear to readers of the code (or even the author) later. Below is an example of a method with magic numbers in it:

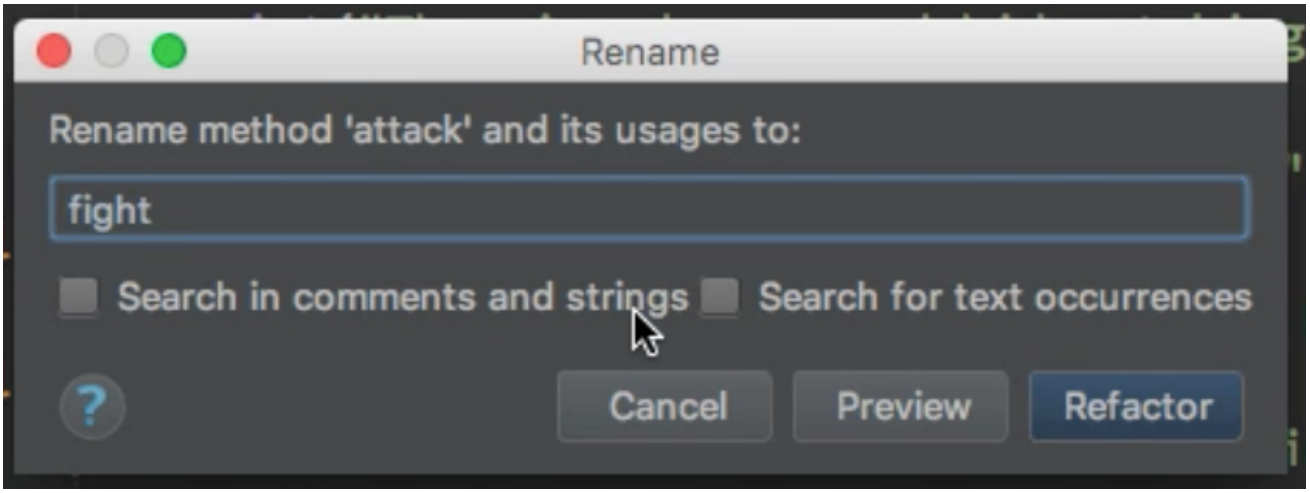


Figure 64: Figure illustrating options for renaming a method.

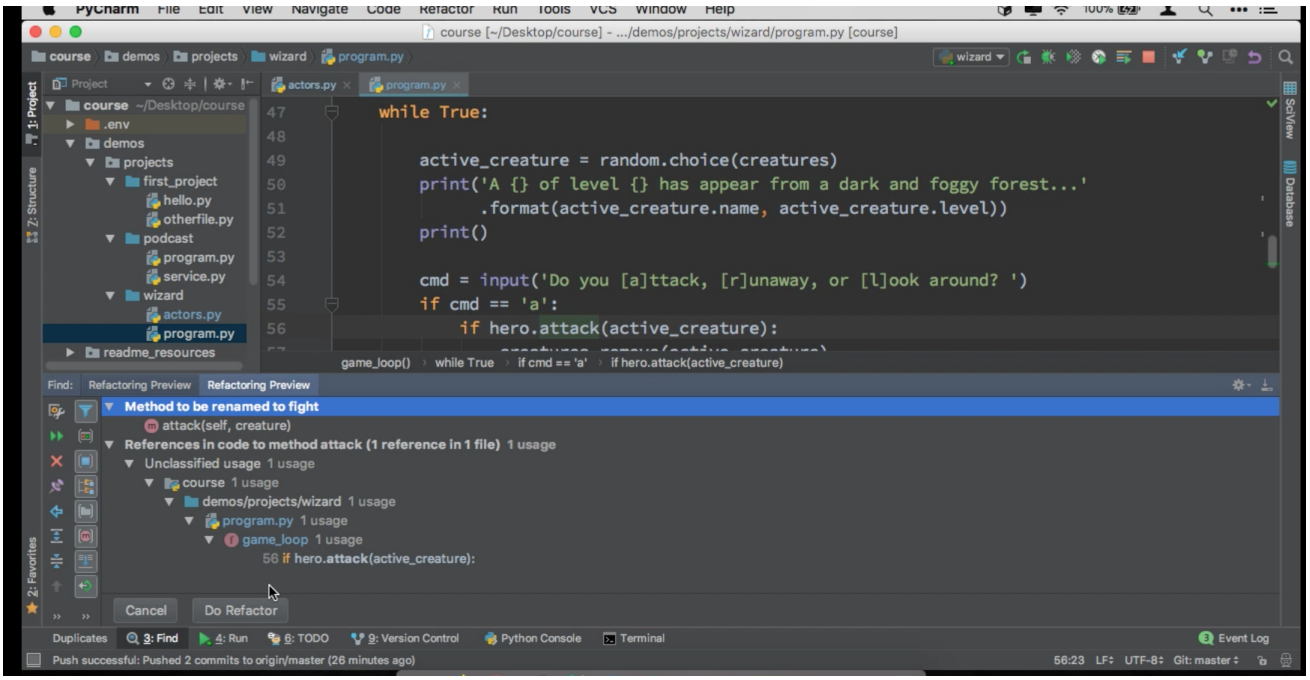


Figure 65: Figure illustrating refactoring preview. Here you can see where the references were found before approving the refactor.

```
def get_defensive_roll(self):
    return 3 * random.randint(1, 12) * self.level
```

The inlined literals are `3`, `1`, and `12`. What does `3` mean. Presumably, the other numbers are for randomly choosing a number between `1` and `12` inclusive (note that `randint` doesn't follow the half-open interval as `range` does, and includes the `12` as an option). These can be confusing to colleagues who inspect the code later.

Another issue with debugging this code is that we can't set a breakpoint and inspect the random number itself. We can only set a breakpoint on the `return` statement and reverse engineer what the random number was.

A refactoring to pull out the random number in its own variable is called *"Extract Variable..."* (command-alt-v or ctrl-alt-v). If you highlight the expression with the call to the `randint` function, run the "Extract Variable..." command, and name it as `roll`, your code will look like this:

```
def get_defensive_roll(self):
    roll = random.randint(1, 12)
    return 3 * roll * self.level
```

Following that refactoring, you could set a breakpoint in the method and inspect the value of `roll`.

> **Note**
>
> There is also a *"Inline"* (command-alt-n, or ctrl-alt-n) refactoring, which will take a variable and inline its value. It is the opposite of Extract Variable.

The `3` number is a boosting modifier the defensive roll here. If we wanted to make this value a configurable value we can use the command *"Extract Parameter..."* (command-alt-p, or ctrl-alt-p). You could select it and run the command with a parameter name of `modifier` to get:

```
def get_defensive_roll(self, modifier=3):
    roll = random.randint(1, 12)
    return modifier * roll * self.level
```

We have refactored our code to make it more clear about the intent. We have also added the ability to change the `modifier`, but the default value is the same, so the behavior of the method works with existing code. The method we looked at in this section is somewhat simple. However, you can apply these same ideas to larger, more complex code as well.

6.5 Creating Constants

PyCharm can also *"Extract Constant..."* (command-alt-c or ctrl-alt-c). Python the language has no notion of constants. But we can use a convention and create a global variable with an all uppercase variable.

Assume you were printing out an ASCII art banner to a terminal from my app. You could use code where you embedded a string directly in the `print` function:

```
          _   _
print(''' _ __  _   _| |_| |__   ___  _ __
| '_ \| | | | __| '_ \ / _ \| '_ \
| |_) | |_| | |_| | | | (_) | | | |
| .__/ \__, |\__|_| |_|\___/|_| |_| is cool!
|_|    |___/''')
```

If you wanted to reuse that banner in other places, you could copy and paste the code. Alternatively, you can select the string and run the "Extract Constant. If you rename it `PYTHON_TEXT` you will get:

```
PYTHON_TEXT = ''' _ __  _   _| |_| |__   ___  _ __
| '_ \| | | | __| '_ \ / _ \| '_ \
| |_) | |_| | |_| | | | (_) | | | |
| .__/ \__, |\__|_| |_|\___/|_| |_| is cool!
|_|    |___/'''

print(PYTHON_TEXT)
```

To use that text in another area, reference the `PYTHON_TEXT` variable.

6.6 Moving Code

You can also *"Move"* (F6) code to another module. To apply this refactoring, select the code you want to move and run the Move command.

Assume that you had a single module with all of your code in there. If you had a main function that processed command line arguments and ran a command line script, you could refactor that method to a `main.py` file. For brevity sake, we will assume the code structure looks like this:

```
# process.py
import argparse


class Processor:

     def process_data(self):
         # compute stats


def main():
    p = argparse.ArgumentParser()
    p.add_argument('-f', '--file)
    opts = p.parse_args()
    if opts.file:
        proc = Processor()
        proc.process_data()
```

After selecting the `main` function and running the Move refactoring with a file named `main.py`, your original file would look like this:

```
# process.py
class Processor:

     def process_data(self):
         # compute stats
```

And `main.py` looks like:

```
import argparse

from process import Processor


def main():
    p = argparse.ArgumentParser()
```

```
p.add_argument('-f', '--file)
opts = p.parse_args()
if opts.file:
    proc = Processor()
proc.process_data()
```

This allows you to create dedicated modules that are specific to certain tasks. You can always use the ability to run the code or execute tests (as we will see in a later chapter), to verify that the code still works.

PyCharm also supports moving modules to packages and vice versa. A module is a file that ends in `.py`. A package is a directory that has a file named `__init__.py` in it. If you had a module that was growing large and you wanted to keep the namespace, you could refactor it into a package. The implementation of the module would be placed in the `__init__.py` file, but you could add other modules to it later.

6.7 Summary

We discussed many refactorings in this chapter. It was not a comprehensive discussion (as there are whole books devoted to refactoring), but it should whet your appetite to improve your code. Take advantage of the functionality in PyCharm to find duplicates in your code and clean them up with refactoring. Another next step would be to research common code smells and the remedies for them. PyCharm should make cleaning up your code much easier and also do it in a robust manner.

6.8 Commands

- *"Locate Duplicates"* - (Code -> "Locate Duplicates...")
- *"Refactor This"* - (command-alt-shift-t or shift-ctrl-alt-t)
- *"Pull Members Up..."*
- *"Rename..."* - (shift-F6)
- *"Extract Variable..."* - (command-alt-v or ctrl-alt-v)
- *"Inline"* - (command-alt-n or ctrl-alt-n)
- *"Extract Constant..."* - (command-alt-c or ctrl-alt-c)
- *"Move"* - (F6)

6.9 Exercises

1. Research Code Smells on Wikipedia. See if you can find any of them in your own code.
2. Research Refactorings. There are many books and articles that dive deep into this subject. See if you can find places in your own code to apply them. PyCharm should make that easier.

3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/4-refactoring` to try out finding some code smells and fixing them with PyCharm.

Chapter 7

Databases

7.1 Introduction

Many applications that create or read data use databases to make their lives easier. PyCharm has great database support and can interact with most any SQL database.

Many companies even have dedicated roles to supporting and maintaining databases. This can make developers feel that a database is someone else's domain. Also, tools for using databases can be hard, unfamiliar, or interrupt your flow. Also, most databases have different tools, so learning each tool is a chore. Supporting and creating migrations as well as dealing with object relational mismatches are pains.

PyCharm alleviates some of these issues. It has integrated database support. PyCharm also uses the same UI across databases platforms. You only need to invest in learning a single tool.

Most of this chapter will focus on DataGrip, which comes with PyCharm Professional, but is also a separate product.

7.2 Simple Database

Let's use SQLite to make a simple database. Most databases require a separate server running and lots of configuration. SQLite is great to get started with because it only uses a file as the database and runs in our Python process. Moreover, SQLite is included with Python so there is nothing to install. It will act as a simple store to keep a running log. The principles and tools shown in this chapter apply to any of the data sources that DataGrip supports.

Make a file, `dbexample.py` that has the following content:

```
import sqlite3


conn = sqlite3.connect('runs.db')
with conn:
    c = conn.cursor()
    c.execute('CREATE TABLE Person(id INTEGER PRIMARY KEY, name TEXT)')
    c.execute('CREATE TABLE Location(id INTEGER PRIMARY KEY, place TEXT)')
    c.execute('''CREATE TABLE Run(id INTEGER PRIMARY KEY,
                 pid INTEGER,
                 lid INTEGER,
                 distance NUMBER,
                 FOREIGN KEY(pid) REFERENCES Person(id),
                 FOREIGN KEY(lid) REFERENCES Location(id)
```

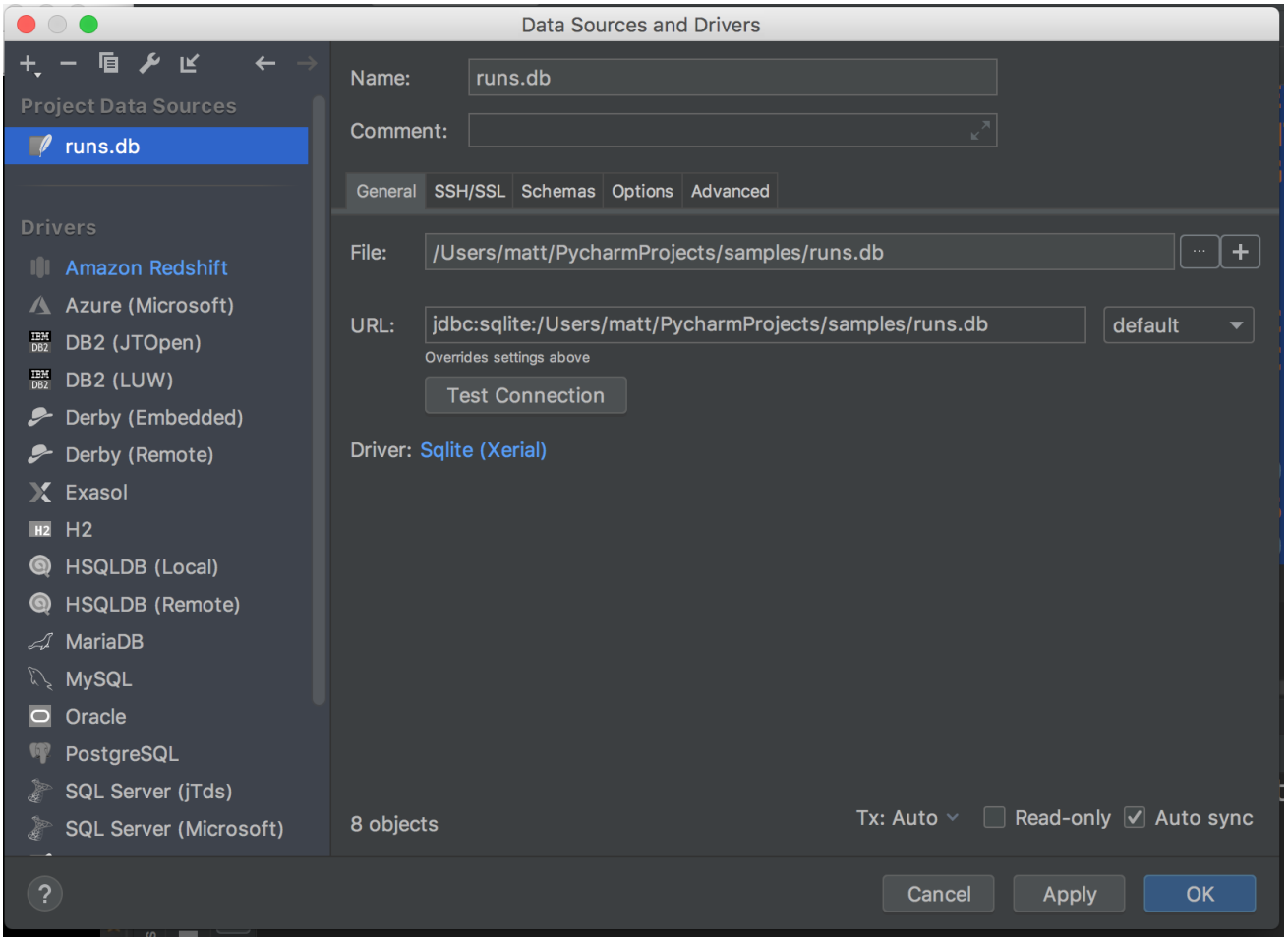


Figure 71: Figure illustrating configuring a new Data Source for SQLite.

```
            )
        ''')
c.execute("INSERT INTO Person VALUES(0, 'Matt')")
c.execute("INSERT INTO Location VALUES(0, 'Squaw Peak')")
c.execute("INSERT INTO Run VALUES(0, 0, 0, 52)")
c.execute("INSERT INTO Run VALUES(1, 0, 0, 50)")
```

If you run this from PyCharm, it should execute without issue. This code creates a file, `runs.db`, on the filesystem in your working directory (often next to the `dbexample.py` file). To get your project to recognize it as a datasource, right-click on the project in the Project view and select "New" -> "Data Source". It will ask for the "Path" to the data source, which you can point to the location of `runs.db`. At that point, the "Driver" option should be filled in with "Sqlite (Xerial)". If you hit "OK", a new window pops up to configure the data source.

The settings on the configure window should be fine. However, if you see a red font by the driver file that will be an indication that you need to download the driver. Click on the blue "Download" link, and PyCharm will handle the rest. At this point hit "OK" and you will notice on the far right of the editor is a "Database" tab.

The Database view has a tree view with the name of the datasource, `runs.db` in our case. If you expand schemas and main, you will see all of the tables we have created: Location, Person, and Run. If you expand one of those tables you will see the columns with their corresponding types

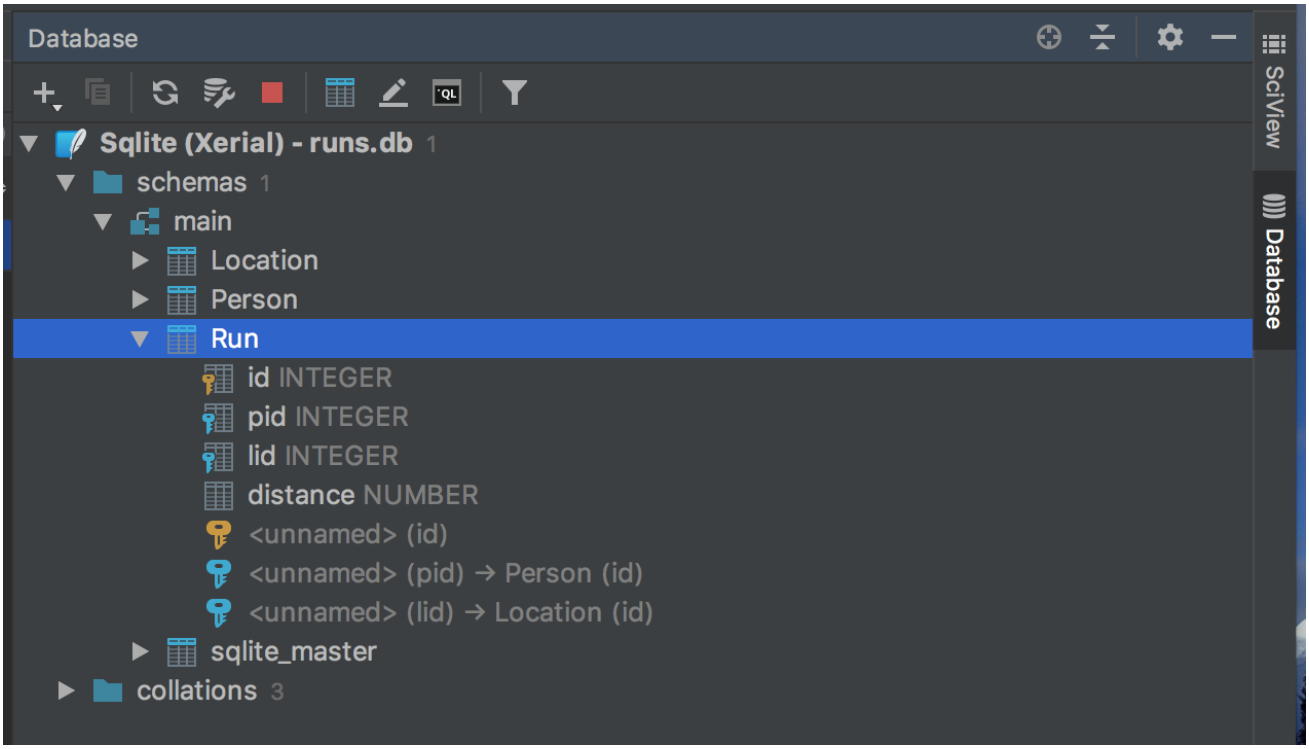


Figure 72: Figure illustrating datasource view. Note that you can expand it to view column types.

and any keys or indexes that were defined. There are icons on the column names to indicate if they are a key (if we had an index, there is also an icon for that).

The nice thing about this view is that it will work with any data source. Once you have the muscle memory, you can use a different database and it should feel the same.

Tip

You can also drag and drop a sqlite file onto the database tab and PyCharm will configure it for you!

7.3 Database Diagrams

Our schema consists of three tables and a few columns. You will probably work with tables that are much larger. Reading the DDL (data definition language) of the tables is one way to understand the relationships between those tables. Often it is easier visualize the relationships. DataGrip to the rescue.

The "Show Visualization" command (right click on the data source "Diagrams" -> "Show Visualization", command-shift-alt-u, or ctrl-alt-shift-u) will show the tables, columns, and their relationships. Along the top of this view are options to hide columns or keys, zoom in or out, and save the visualization.

If you want a floating window, there is a sibling option to "Show Visualization", "Show Visualization Popup", that will create a popup window with the visualization in it.

A final option that might not be intuitive at first is viewing only specific tables. The visualization is interactive, and you can move the tables around relative to one another. You can also delete a table from the view by highlighting it hitting "Delete". Alternatively, you can select only the tables

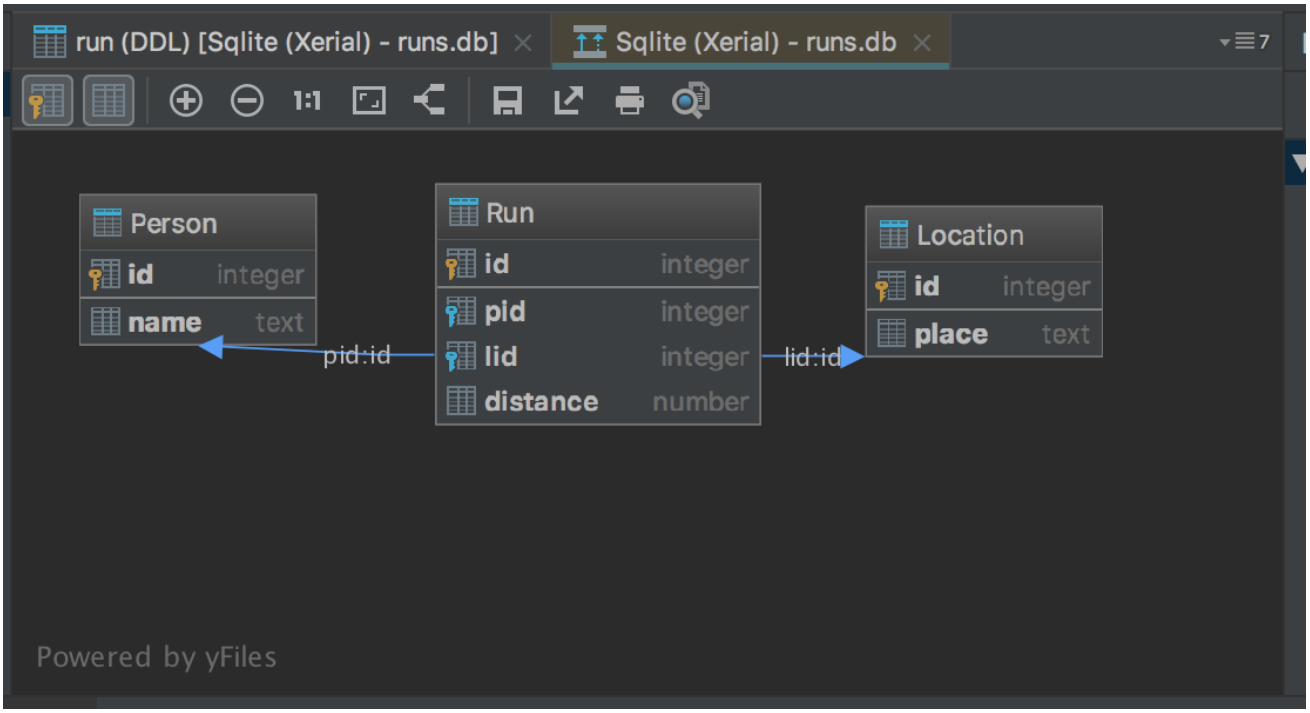


Figure 73: Figure illustrating the result of the "Show Visualization" command.

you want to include in the data source view, and then show the visualization, which will be limited to the tables selected.

7.4 Querying Data

We have a data source that has been configured. We also have a visualization to see the relationships in our tables. Let's look at querying the data. The easiest way to view data is to double-click on a table. If you do that, DataGrip will show you the first 501 rows from that table.

The console is an editor where you can type in SQL and execute it. Because we have configured a data source, it knows about tables and columns, and we have code completion.

If you are versed in SQL and want to use structured query language to pull data out, you can run the *"Open Console"* command (right click on data source or F4). Let's type in this query:

```
SELECT r.id, l.place, p.name, r.distance
FROM Run AS r
      LEFT JOIN Person AS p on p.id = r.pid
      LEFT JOIN Location AS l on l.id = r.lid;
```

Although DataGrip has code completion, if you started writing `SELECT l.` and were waiting for assistance, you would be waiting for a long time. In that case, DataGrip doesn't know what the alias `l` refers to. Even if you typed out `SELECT Run.`, DataGrip would not pull up the options selecting the columns of the `Run` table. A hack to trick DataGrip is to first say `SELECT * FROM Run as r`. At this point, DataGrip will know what `r` is aliased to, and will give you column completion on that name.

If you *"Execute"* (command-enter, or ctrl-enter) the query a data grid will appear with the results. The data grid that displays the results is live. So you can sort the table by clicking on the column headers. If you wanted to sort by distance, you could click on that column.

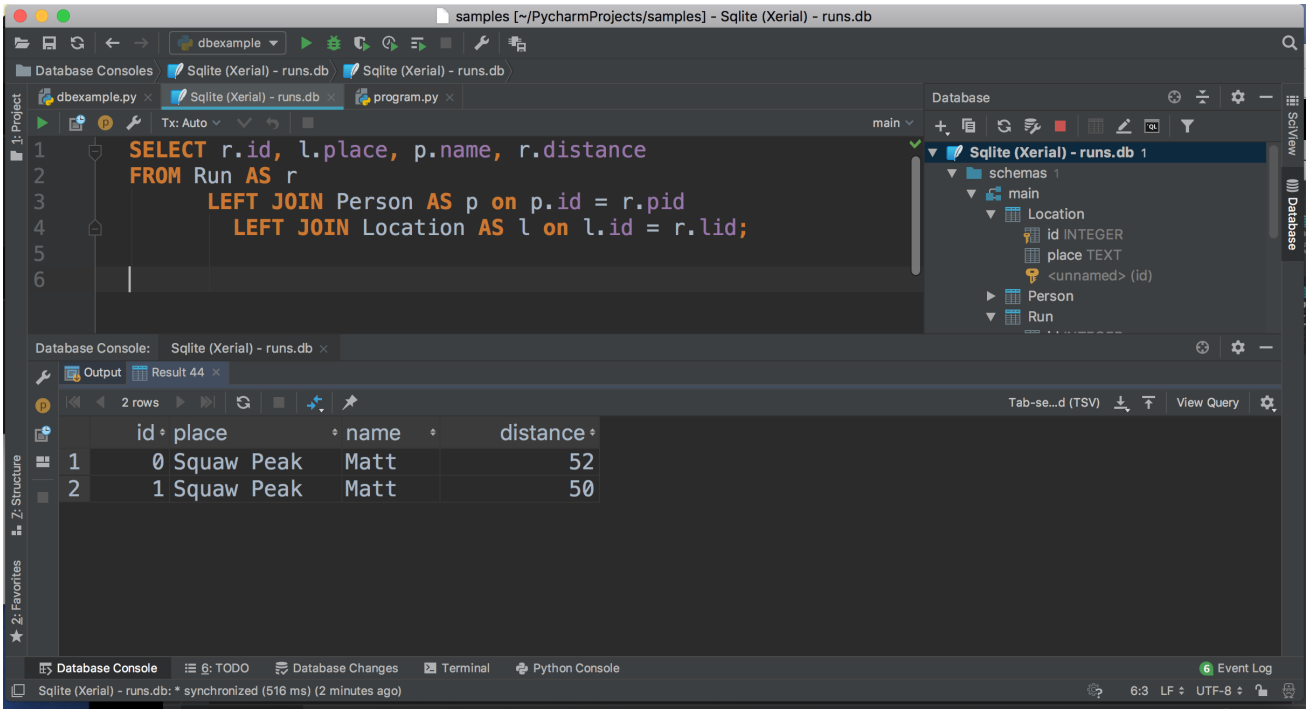


Figure 74: Figure illustrating executing a query from the console

If you view a table directly (by double-clicking it on the data source view, or doing a `SELECT * FROM Runs`) you can also edit the data. You can double click on a field to change its value. After you change the value, it isn't automatically pushed back to the database. You need to run the *"Submit"* command (hitting the button with a green arrow pointing to the DB, or command-enter, or ctrl-enter).

You result view also has the ability to export the results as a TSV, CSV, JSON, or the clipboard. In addition, you can view the DDL for a table. This can be useful if you didn't create the table or don't have the code that created it. You can use the DDL and a dump of the data to make a local copy for your own purposes.

7.5 Modifying Schemas

From the data source view, we can change the structure of the database. There are simple refactorings we can do such as renaming a database column. For more powerful changes run the *"Modify Column..."* (command-F6, or ctrl-F6) command. This brings up the *"Modify Table"* view. From here you can change the name, type, default, and uniqueness attributes of a column. You can add columns, keys, and indexes. DataGrip can run these commands for you or give you the appropriate SQL in a console for you to run.

Remember to run the *"Synchronize"* command (refresh button or command-alt-y, or ctrl-alt-y) after you have changed the database. This will update the data source view, so it reflects the new changes.

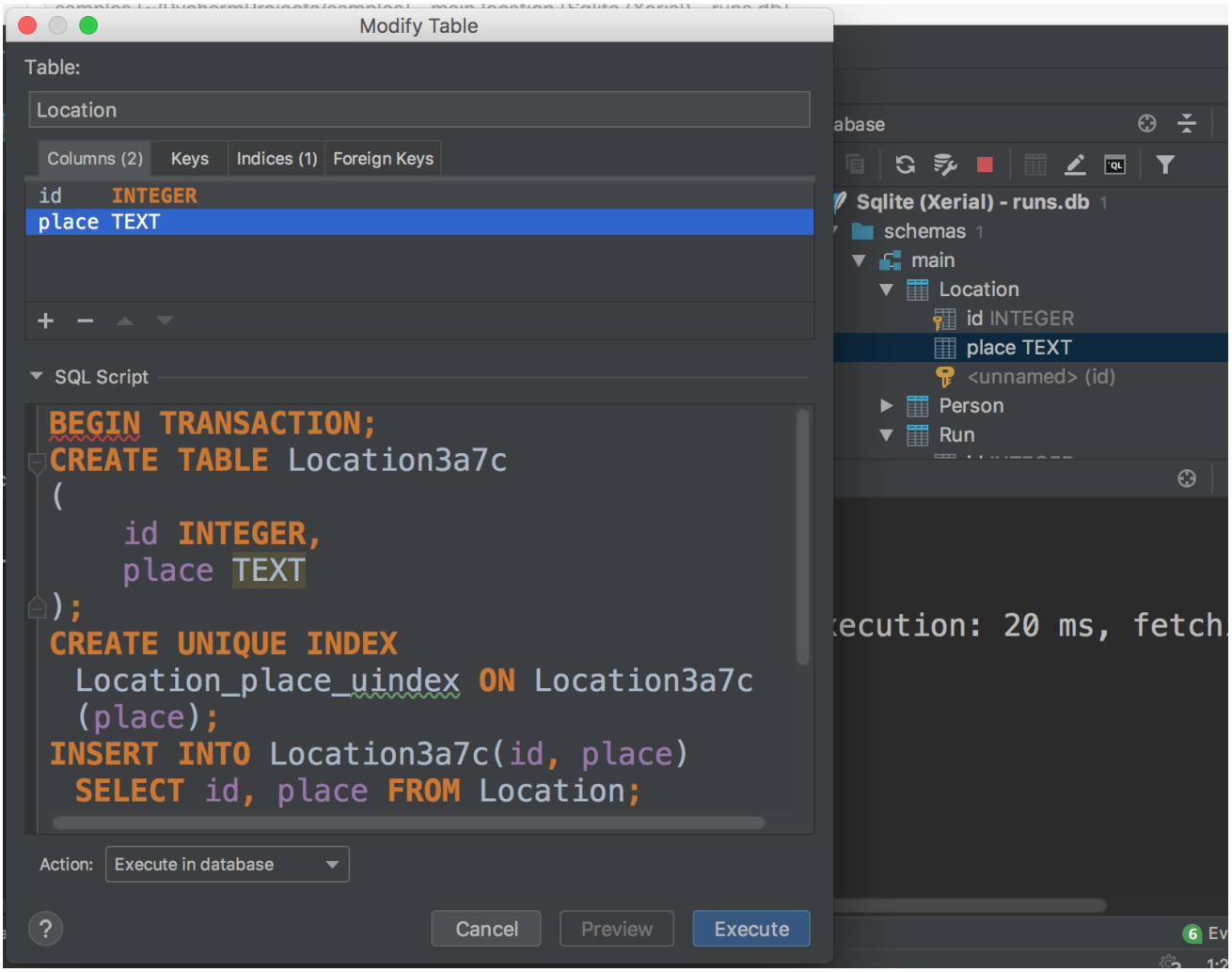


Figure 75: Figure illustrating different options for modifying parts of a table. You can add and remove columns. You can also change attributes of columns.

7.6 A coding delight

The DataGrip tooling is comprehensive and powerful. It supports most SQL databases, and gives you powerful creation, mutation, querying, and visualization tools. Let's look at a feature back in the PyCharm editor called *Language Injection*.

Language injection is the ability of PyCharm to treat string literals as if they contained another language. When you click on a string, the bulb appears, and you can select "Inject language or reference" to tell PyCharm what language is embedded. PyCharm is also smart enough to detect when we have typed enough SQL that a string is SQL and it will automatically convert it for us.

When you use language injection you have code completion on the schema of the connected data source. If you type:

```
q = 'select * from'
```

PyCharm will treat it as a normal string. You can click on the bulb and inject a language or type a little more SQL. Then PyCharm converts the string to SQL automatically. Once I have typed:

Figure 76: Figure illustrating typing SQL in a string. When you start typing it is treated as a normal string. If you want to hint to PyCharm that it contains SQL, you can click on the bulb and do a language injection.

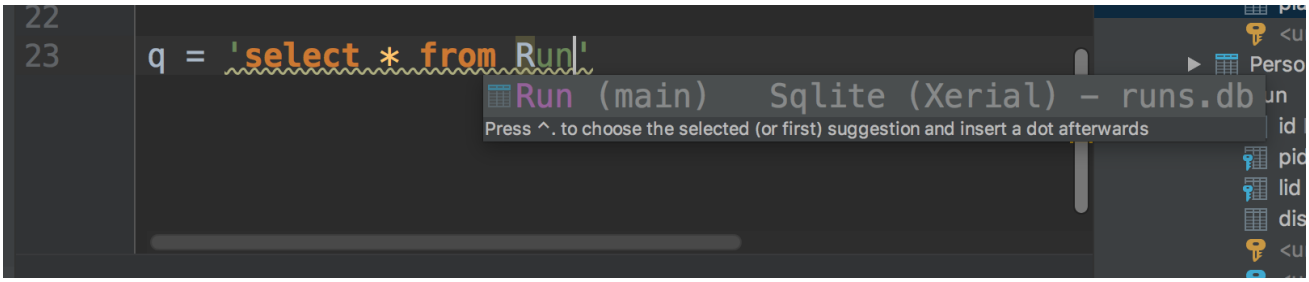


Figure 77: Figure illustrating that PyCharm will automatically inject a SQL language and give us code completion if we have a data source configured and type enough SQL.

```
q = 'select * from Run.'
```

PyCharm pulls up code completion options for you. It will complete table names, column names that are unique to your schema. When you modify the schema, PyCharm modifies the symbols in the language injected SQL strings in Python. As we can see, PyCharm's understanding of symbols gives it great power. You can even by navigate symbols as well. There are many other languages you can inject, but SQL completion is a really nice feature of PyCharm.

7.7 Summary

In this chapter we looked at the DataGrip tooling integrated into PyCharm. This is a powerful end to end suite of tools for interacting with most SQL databases. One of the advantages of DataGrip is that once you learn the tool you can use it with any other supported database. We also discussed language injection. This feature gives us code completion inside of native Python strings. The database tools in the professional version of PyCharm are a compelling reason to choose that platform.

7.8 Commands

- "Show Visualization" - (right click on the data source "Diagrams" -> "Show Visualization", command-shift-alt-u, or ctrl-alt-shift-u)
- *"Open Console"* - (right click on data source or F4)
- *"Execute"* - (command-enter or ctrl-enter)

- *"Submit"* - (hitting the button with a green arrow pointing at the DB, or command-enter, or ctrl-enter).
- *"Modify Column..."* - (command-F6 or ctrl-F6)
- *"Synchronize"* - (refresh button or command-alt-y or ctrl-alt-y)

7.9 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/5-databases` to try out opening an existing database with DataGrip.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/5-databases` to try out running a query from DataGrip.
3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/5-databases` to try out viewing a diagram of database tables.
4. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/5-databases` to try out language injection.

Chapter 8

Server-side Python Web Applications

8.1 Introduction to Server-side Web Applications

Python is a very popular language right now. Where other languages seemed to have flatlined or are decreasing in popularity, Python continues to rise. One of the reasons for this is because Python is an excellent choice for creating server-side web applications. Sites like Instagram, Pinterest, Dropbox, YouTube, and Yelp are implemented with Python.

PyCharm makes the development of server-side projects much easier because it has the understanding of a whole project (Python code, JavaScript, CSS, and more). In addition, there is support the popular Python web frameworks such as Flask, Django, Pyramid, and Google App Engine. PyCharm can start servers, run external tools, and more.

Another aspect of web development is front-end development. This chapter will focus on back-end support, and the following chapters will explore some of the features the PyCharm has for front-end development.

PyCharm has many features back-end development. You can create new projects or load existing projects. PyCharm understands the common templating languages in Python so that you can have code completion from within your dynamic HTML templates such as Jinja2 and Chameleon templates. PyCharm has integration with framework management tools, so it has integration for `manage.py` on Django and `setup.py` on Pyramid. PyCharm supports resource directories where static files like CSS, JavaScript, and images live. You will get code completion to paths in resource directories and support for CSS and JavaScript completion. You can navigate to symbols in templates. You can even use the debugger from some templating languages. Finally, you can configure external tooling as well. If you need to run Gulp or Webpack, PyCharm has integration for those.

Many web developers choose Python for an implementation language. PyCharm has excellent support for developers creating and maintaining web projects. To put it succinctly, PyCharm is a beast for developing Python-backed web applications.

8.2 Creating Server-side Projects

Let's start by creating a new project. We will create a Pyramid project for the examples in this chapter. Some of the features might be a little different like which template language is used or routing options, but the general ideas apply to all the frameworks that PyCharm supports.

To follow along, run the *"New Project..."* command. In the "New Project" window, choose a "Pyramid" project. You can take advantage of PyCharm's virtual environment support to create a

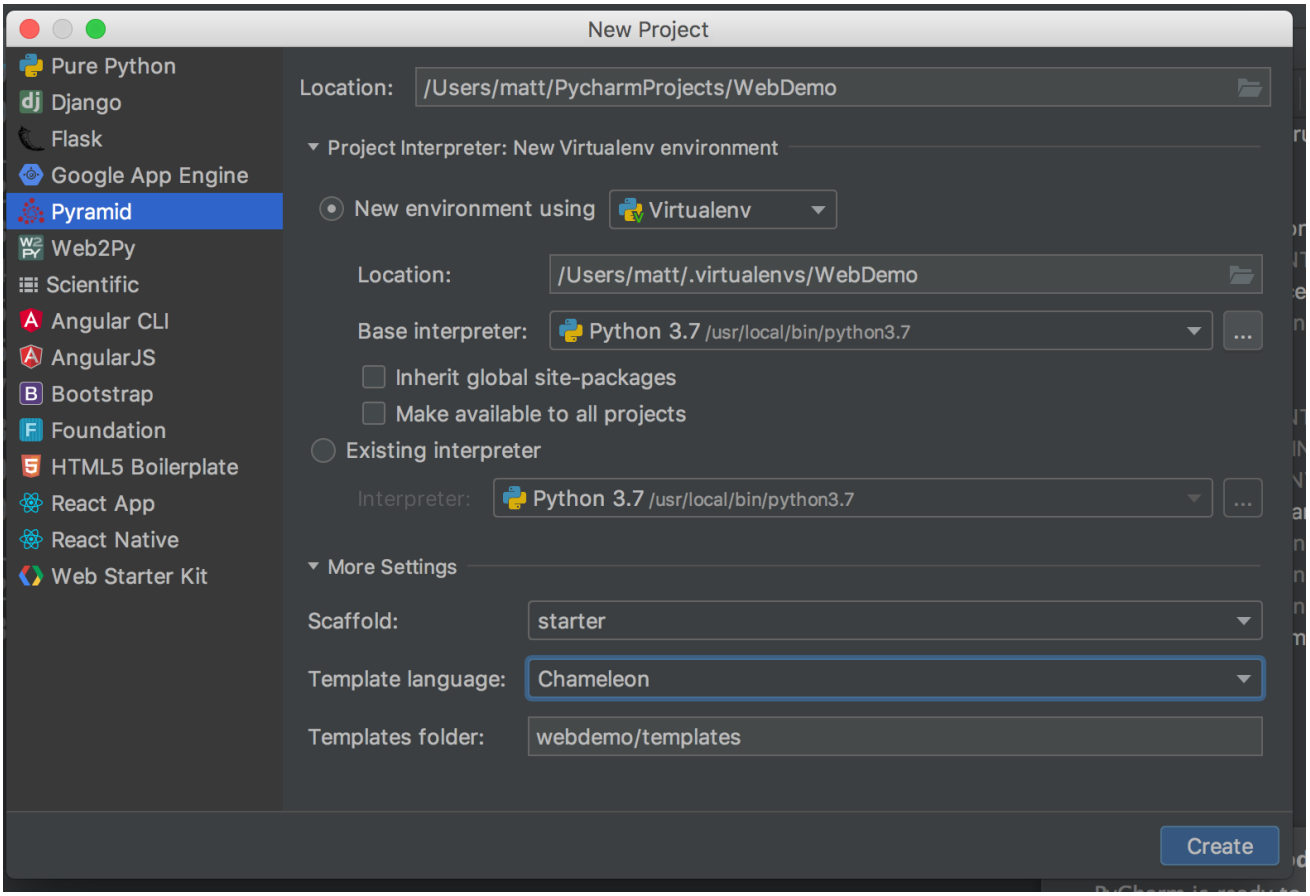


Figure 81: Figure illustrating creating a new Pyramid project.

new virtual environment. In the "More Settings" area at the bottom of the window, you will see options for "Scaffold:", "Template language:", and "Templates folder:". The scaffold is the internal project layout of the Pyramid project. The default value of "starter" is fine. In this chapter we will use the "Chameleon" template language instead of the default value of "Jinja2". Hit the "Create" button and PyCharm will generate a project for you.

After creating the project, PyCharm may generate some warnings about missing packages and the project not be installed for development. You can view what PyCharm installed for you by clicking on Terminal in the bottom and running `pip list`. We will want to install these dependencies, so click on "Install requirements".

You will also need to run `python setup.py develop`. This command installs our project as an editable package. The project source code will be found during imports from our virtual environment, but if we change any code, we won't need to reinstall it. We can do that from the terminal, click on "Run setup.py develop", or select "Tools" -> "Run setup.py Task..." and choose "develop".

At this point you click on *"Run (WebDemo)"* (shift-F10). A "Run" console will appear in the bottom of PyCharm with a link to the development server. Click on the link to view your web project. If you see an error page, you will need to open `webdemo/templates/layout.pt` and replace `WebDemo:` with `webdemo:`.

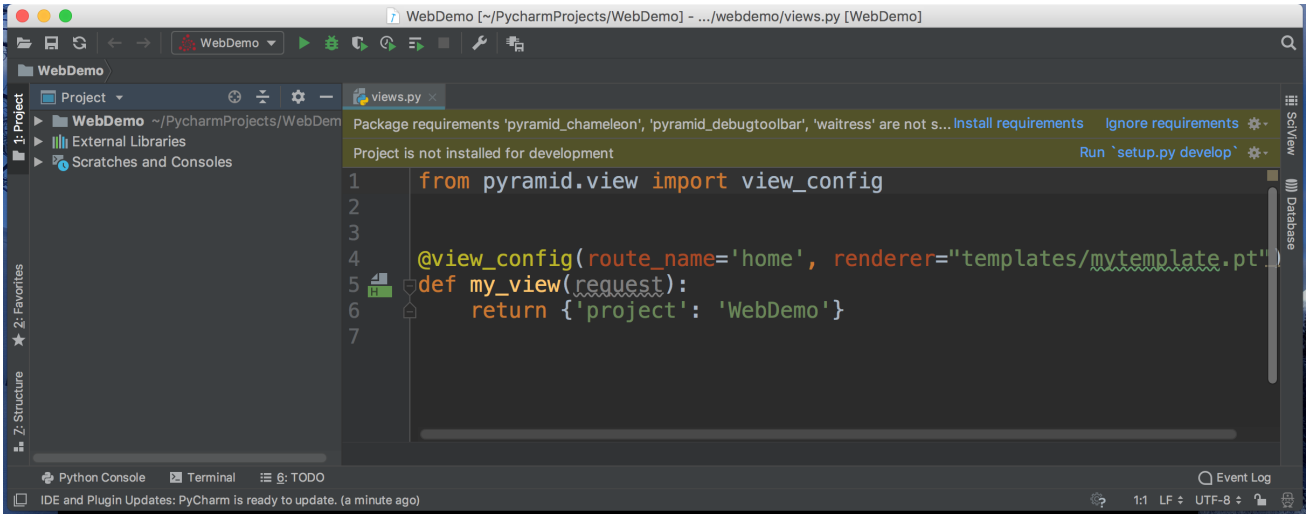


Figure 82: Figure illustrating the generated project. Note that PyCharm has generated some warnings.

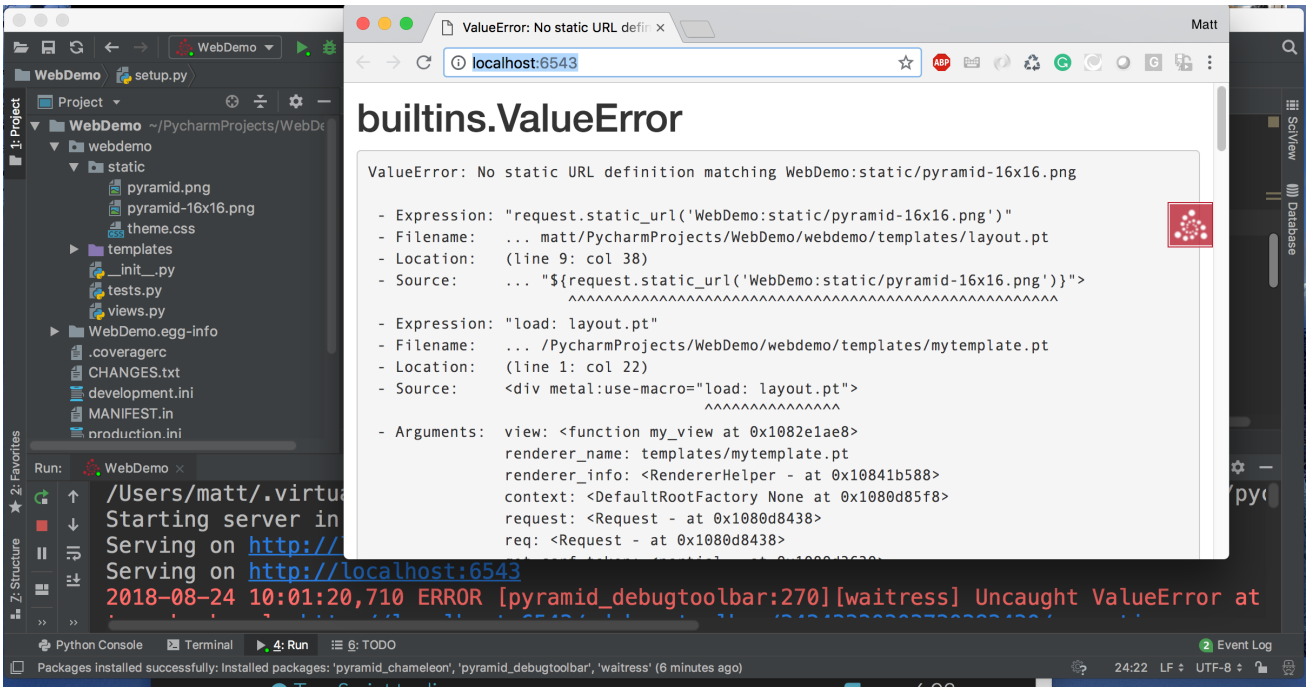


Figure 83: Figure illustrating error if static calls are pointing to `WebDemo:` instead of `webdemo:`.

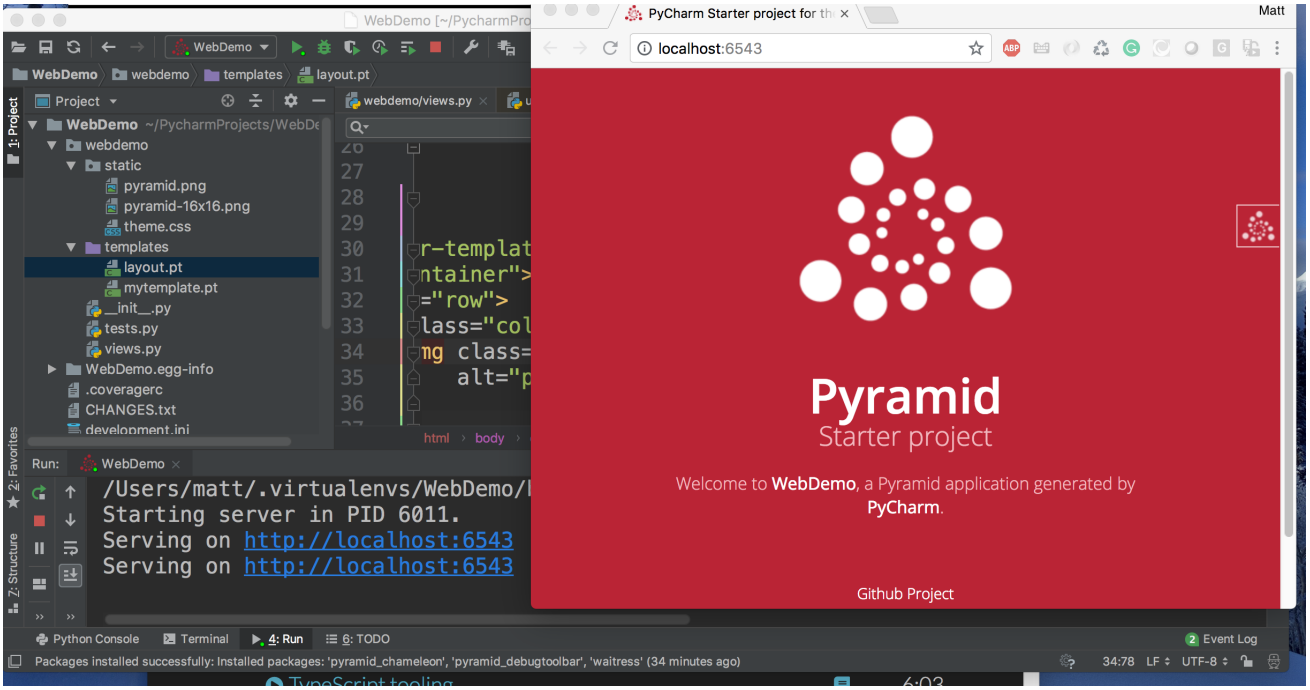


Figure 84: Figure illustrating successfully launching web page.

Tip

Once you have the runner working for your webpage, you should make one more change. If you try to run it again you will get an error complaining that the "Address already in use". This is because the port that the web application is using is not available. A small change can address that.

Tweak the run configuration by running *"Edit Configurations..."*. Then select the "Single instance only" checkbox. This insures that if you hit run again, PyCharm will kill the prior version and you won't see this error.

Pyramid routes views to renderers. In `webdemo/views.py` you will see this code:

```
@view_config(route_name='home',
             renderer="templates/mytemplate.pt")
def my_view(request):
    return {'project': 'WebDemo'}
```

This code says this view has the name of `home`, uses the `templates/mytemplate.pt` template, and returns a dictionary of data.

In `webdemo/__init__.py` there is some code that maps the `home` name to the route of `/`. The line looks like this:

```
config.add_route('home', '/')
```

This routing abstraction might seem confusing, but it can be powerful. To validate that the web application is dynamic, update the return value of `my_view` to add a name:

```
@view_config(route_name='home',
             renderer="templates/mytemplate.pt")
def my_view(request):
    return {'project': 'WebDemo',
            'name': 'Matt'}
```

Figure 85: Figure illustrating the view. Note the icon with the H in the left gutter. If you click it, you will be taken to the template.

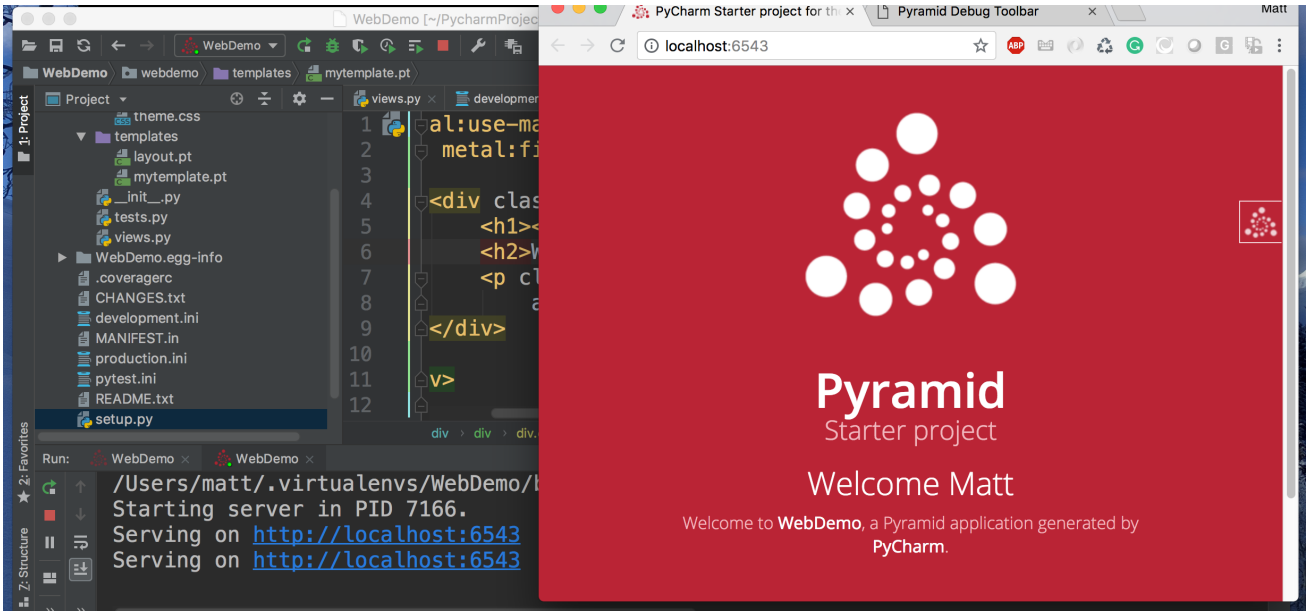


Figure 86: Figure illustrating template rendering data from a view.

You will see on the left of the view in the gutter is an icon with an H under it. If you click it, it will take you to the template for that view. Now change the template (`templates/mytemplate.pt`). There is a line that looks like this:

```
<p class="lead">Welcome to <span class="font-normal">WebDemo</span>
```

Add a line before it that says:

```
<h2>Welcome ${name}</h2>
```

Reloading the webpage will show you that the template is using the data provided from the view.

Note

If you don't want to restart Pyramid all the time, you can add `--reload` as an "Additional options:" flag in the "Run Configurations". This is specific to Pyramid, but other frameworks have similar functionality. You might need to start Watchman in a terminal (`watchman -f`) or install Watchdog (`pip install watchdog`).

This tweak makes development easy. You change some code, and the server notices it, and restarts.

8.3 Template Tooling

In this section we will explore some of the code completion support in templates. Go back to the view add update it to provide some data and a boolean flag:

```
@view_config(route_name='home',
             renderer="templates/mytemplate.pt")
def my_view(request):
    return {'project': 'WebDemo',
            'name': 'Matt',
            'data': [1, 1, 2, 3, 5, 8, 13],
            'show_odd': True}
```

Go back the `mytemplate.pt` and delete the content inside of the `<div class="content">` tag. Add the following content to show flag indicating whether we are showing odd numbers. Because we are using Chameleon, we will be using TAL (template attribute language) to do that. Other templating language have different syntaxes but the same ideas apply:

```
<h1>Template Demo</h1>
<h2 tal:condition="show_odd">Showing odd numbers as well</h2>
<ul>
  <li tal:repeat="n data">Num ${n}</li>
</ul>
```

A few things to note. PyCharm will automatically close HTML tags for you. PyCharm also takes care of code completion with TAL attributes. Sadly, PyCharm does not complete the context variables (`data`), but it does know about `n`.

If you change the value for the `'show_odd'` key in `my_view` to `False`, the webpage will not filter out the odd values. To hide those, you need to change the template code a little. TAL has a `tal:condition` attribute that can turn tags on or off, but you can't apply it to the `<li>` tag because we are looping with it. To remedy that, you can change the `<li>` tag to a `<div>` tag (highlight `li` and type `div`), and then add a new inner `<li>` tag:

```
<h1>Template Demo</h1>
<h2 tal:condition="show_odd">Showing odd</h2>
<ul>
    <div tal:repeat="n data" tal:omit-tag="True">
        <li tal:condition="n%2 == 0 or show_odd">
            Num ${n}
        </li>
    </div>
</ul>
```

If you take advantage of PyCharm's smarts around editing HTML, this change is rather painless.

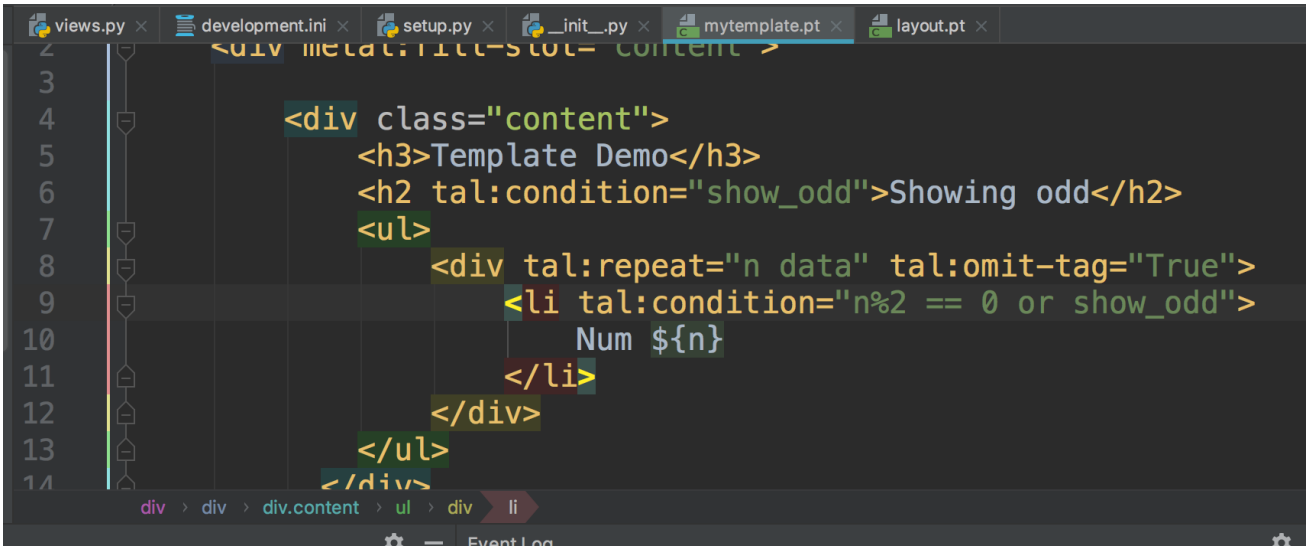


Figure 87: Figure illustrating HTML tag hierarchy at bottom of editor. Click the colored elements in the bottom bar.

Another thing to note is that PyCharm tracks the hierarchy of tags where the cursor is at the bottom of the editor. These tags are clickable if you want to use them to navigate. In the figure below, you'll see `div > div > div.content > ul > div > li`. Each of these can be clicked to quickly navigate through the DOM hierarchy without losing your place.

8.4 Emmet

PyCharm ships with Emmet integration. Previously known as *Zen Coding*, Emmet allows you to create HTML from CSS. At the most basic level, you can create single elements. To create the `<div></div>` tag, type `div` in the template editor and hit TAB.

If you wanted to make a `<div class="blue" id="header"></div>` tag, type:

```
div.blue#header
```

and then hit TAB.

Emmet can do much more. It supports children (`>`), siblings (`+`), climb-up (`^`), multiplication (`*`), grouping (`()`), ids (`#`), classes (`.`), custom attributes (`[key="value"]`), text (`{}`), and more.

The following code:

```
div.content>h1{Welcome}>^(div{Lorem}+div{Ipsum}+ul>li*3)
```

Expands to this by hitting TAB following it:

```
<div class="content">
    <h1>Welcome</h1>
    <div>Lorem</div>
    <div>Ipsum</div>
    <ul>
        <li></li>
        <li></li>
        <li></li>
    </ul>
</div>
```

Figure 88: Figure illustrating Emmet preview pop-up.

There is an option to have a pop-up window with a preview of the abbreviate before expanding it. Go to preferences and search for "emmet" and click on "Enable abbreviation preview".

8.5 Static files

We have mentioned that PyCharm allows us to mark directories as special types (source roots, resource roots, ignored, etc.). We can mark template directories as "Template Folders" (PyCharm should handle this if you create a new project, but not if you load an existing project). We can also mark directories as "Resource Root". Resource root are for static files like images, CSS, and JavaScript.

If we want `static` to be the first part of the url, we need to mark to root (the directory containing the static folder, `webdemo` in our case) as a resource root. In the `layout.pt` template we are using jQuery and Bootstrap which are hosted on a CDN (content delivery network). This is great for production, but can make development and debugging harder.

Let's install a local copy of Bootstrap to show how resource roots work. Create a directory `webdemo/static/css`. Use the "Terminal" window to `cd` into that directory and run:

```
$ npm install bootstrap
```

This command requires node to be installed npm in your system path. Now in `layout.pt`, go to the line with:

```
<link href="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/css/
bootstrap.min.css" rel="stylesheet">
```

Change the `href` value to `/static/` and you will see that you have directory completion because the root directory was marked as a resource root. Finish filling it out with the new path to bootstrap:

```
/static/css/node_modules/bootstrap/dist/css/bootstrap.css
```

At this point, PyCharm will recognize and parse the CSS and it will be available for code completion. Note that if our project references content hosted on CDN's (that aren't in a resource root), PyCharm won't use them for code completion. Also, any symbols PyCharm finds in static files (JavaScript functions, CSS class names) are "symbols" in the PyCharm sense. So we can leverage that for navigation, code completion, refactoring, etc!

Figure 89: Figure illustrating code completion after we have added a local copy of bootstrap in a resource root.

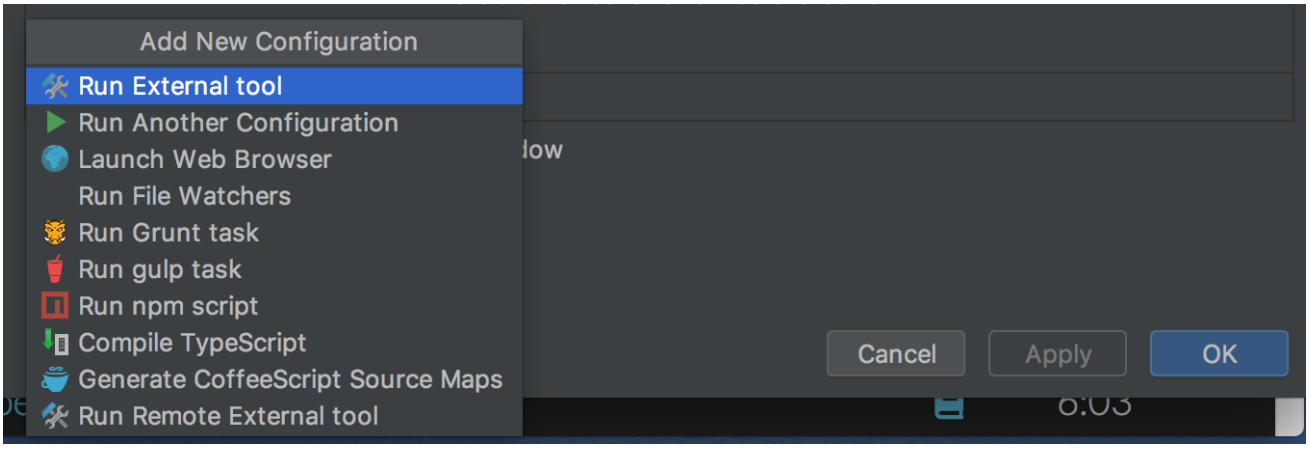


Figure 810: Figure illustrating various options for running tools before launching a web application.

8.6 Other Tools

A final configuration setting that can come in useful is integration with external utilities. Many web applications require certain tools or scripts to be run before launching the app. If this applies to your application, PyCharm has support for this. In the "Run Configurations" window for a web application there is a "Before launch: Activate tool window".

From this area, you can add various tools. There is built-in support for running Grunt, Gulp, NPM, and more. Select the option that fits your needs. You can run multiple tools as well if needed.

8.7 Summary

In this chapter, we explored PyCharm's support for server-side development. PyCharm has many frameworks supported out of the box. Sometimes we need to provide some extra hints to PyCharm so it knows how to handle certain files. Once we do, we can reap the benefits of code completion. PyCharm also enables us to quickly create HTML code with the Emmet integration.

8.8 Commands

- *"Run (WebDemo)"* - (shift-F10)
- *"Edit Configurations..."*

8.9 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/6-web-applications` to try out creating a basic Pyramid application.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/6-web-applications` to try out implementing a view method.
3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/6-web-applications` to try out rending data from a view.
4. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/6-web-applications` to try out changing the templating for a web application.

Chapter 9

Client-side Web Applications

9.1 Introduction

One of the key benefits of PyCharm is that it understands a whole project. In addition to Python, there strong support for JavaScript and web development. If you are working on a web application with JavaScript on the frontend, PyCharm has you covered. It understands frameworks like Angular, React, and Vue. It also integrates with TypeScript, LESS, and SASS. Finally, PyCharm can help with using ElectronJS to create desktop applications. In this chapter, we will be exploring these features of PyCharm.

9.2 Basic HTML and JavaScript

Let's start by using some of the basic features of PyCharm for standard JavaScript. First, create a new project. To keep things simple, create a new Bootstrap project. Ours is called *JSDemo*. Inside of the root directory, add an HTML file, `index.html`. Add the following content to the file. Make sure it has the `script` tag:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <script src="js/site.js"></script>
</head>
<body>
    <h1>Welcome to JS Demo</h1>
</body>
</html>
```

In the `js` folder create a `site.js` JavaScript file. Add the following contents to it:

```
function runAtLoad() {
  person = "Paul"

  console.log("Nice to meet you" + person)
}

runAtLoad()
```

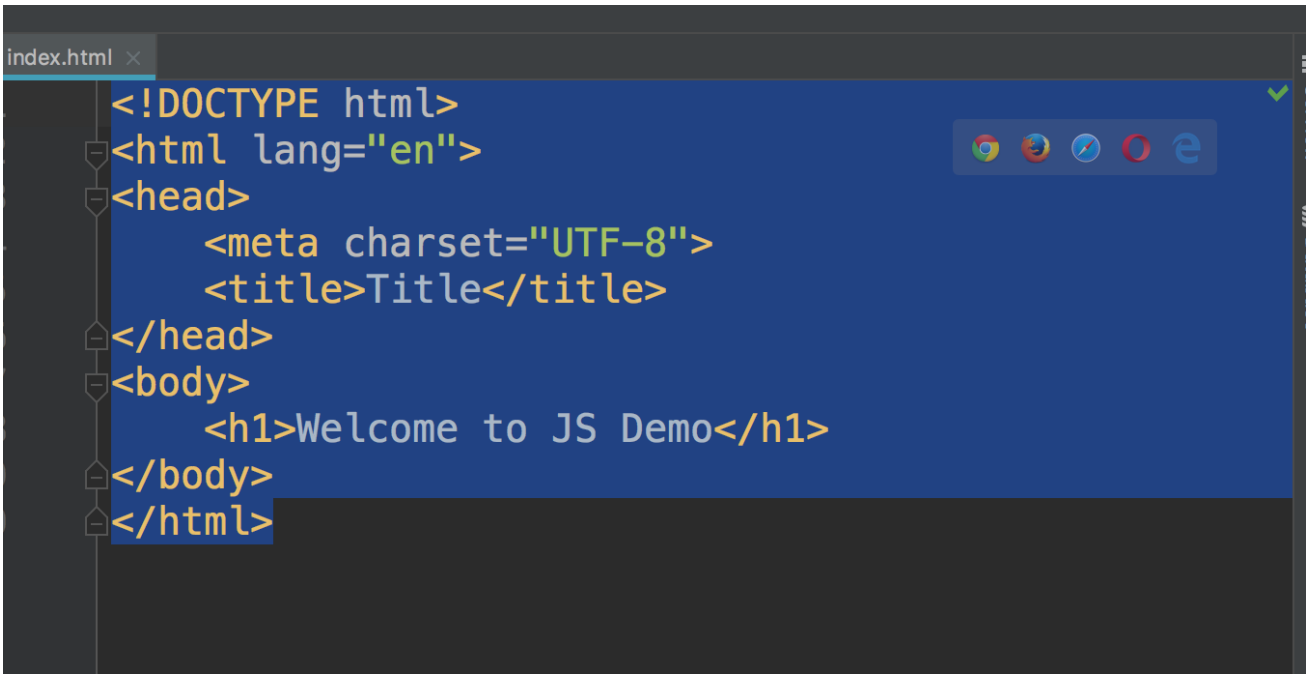


Figure 91: Figure illustrating the browser icons in the upper right of an HTML page.

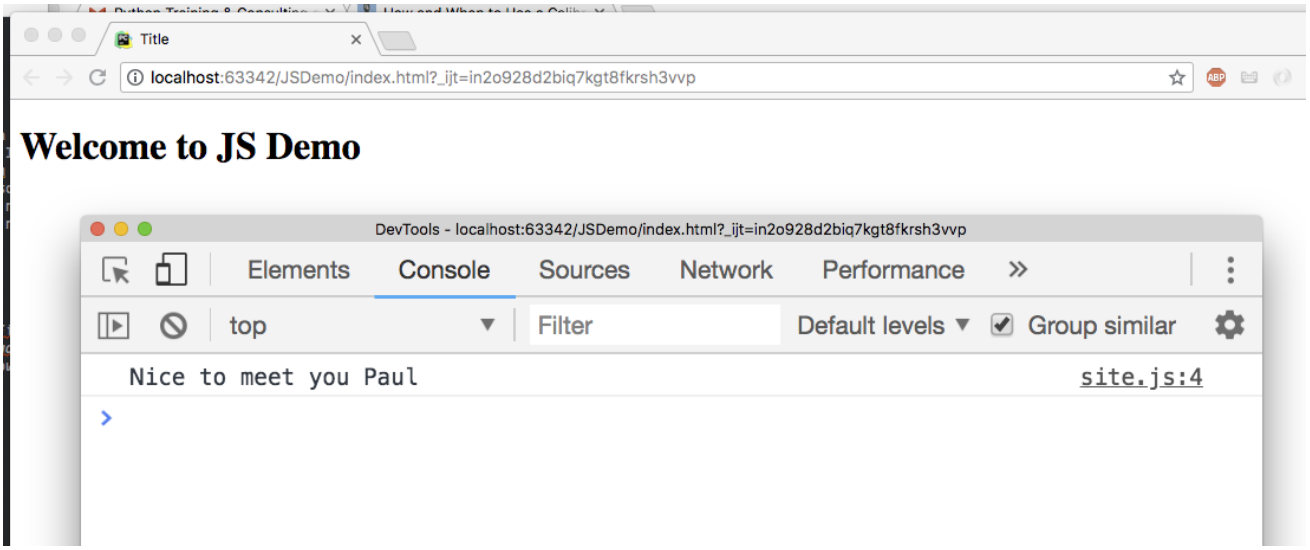


Figure 92: Figure illustrating opening up the Chrome JavaScript Console.

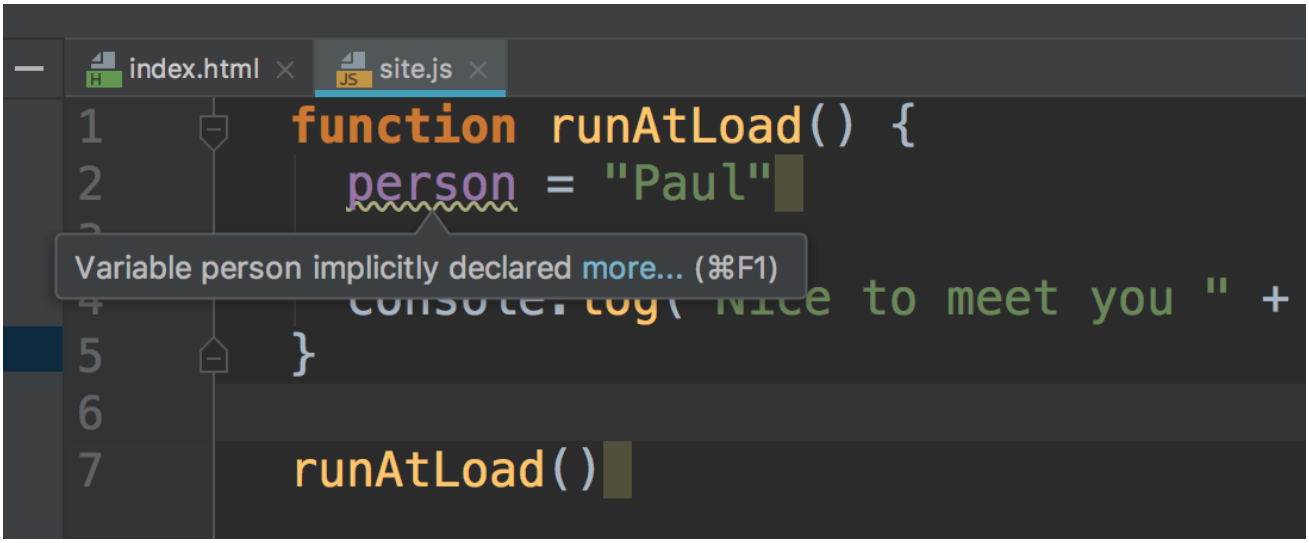


Figure 93: Figure illustrating that PyCharm will offer an intention to fix a variable lacking a scope declaration.

If you click on the Chrome icon on the HTML page, you should see a web page that says "Welcome to JS Demo". If you open the JavaScript Console (View -> Developer -> "JavaScript Console"), you should see the text `Nice to meet you Paul`.

The code appears to have run, but there is an issue. The `person` variable doesn't have a scope declaration. Unlike Python where variables are local to the scope they are defined in, JavaScript would prefer that you are more explicit when defining the scope.

The default scope in JavaScript for a variable is global scope. That's what's happening with our person string variable. Global variables tend to cause problems as they remove encapsulation and can be changed anywhere in the code. It is best to avoid them unless you explicitly need them. You will see that PyCharm underlines the `person` variable and gives you a code intention to address this.

If you click on the intention, it will change the line to:

```
let person = "Paul"
```

With the `let` scope declaration included, the `person` variable is no longer global. From the JavaScript console, you can try and access the `person` variable and it is no longer possible.

PyCharm has refactoring support for JavaScript. Highlight the code `"Nice to meet you " + person` and run the *"Extract Method"* command (command-alt-m, or ctrl-alt-m). You will be presented with the option to refactor to a global scope or inside of `runAtLoad`. Select refactor to global scope. Your code will look like this:

```
function getMessage(person) {
    return "Nice to meet you " + person;
}

function runAtLoad() {
  let person = "Paul"

  console.log(getMessage(person))
}

runAtLoad()
```

Figure 94: Figure illustrating creating a TypeScript file. Note the dialog to compile this file to JavaScript along the top.

We can also refactor to another file. Select the `getMessage` function and run the *"Move"* command (F6). Tell PyCharm to move it to a file, `js/home.js`. PyCharm will create a new file for you with the refactored content.

Note

Some of the refactorings, such as moving code to another file require ECMAScript 6 features. Browser support for such features is continuously improving.

9.3 TypeScript tooling

TypeScript is another tool that is popular among frontend frameworks. TypeScript is a superset of JavaScript that adds type annotations in order to catch certain errors earlier. Angular version 2 and above and frameworks like Ionic are built with TypeScript to take advantage of these benefits. PyCharm also has support for TypeScript.

TypeScript is not an entirely new language. It a typed superset of JavaScript that compiles to plain JavaScript. It's similar to the relationship between C++ and C. Your browser doesn't need to understand TypeScript because it never sees TypeScript, your browser just sees the JavaScript output of it.

Let's create a new TypeScript file in our project. Call it `app.ts` (note the `ts` extension). Add the following contents to it:

```
class Person {
    constructor(public name: string,
                public age: number) {
    }
}
```

The file editor should have a dialog appear along the top asking if you want to compile this file to JavaScript. If you click yes in the project explorer, the file will become expandable. When you click on the file you will see the compiled JavaScript file appear. In this case it will be called `app.js`.

Your JavaScript files can take advantage of TypeScript code. In the `site.js` file, you can update it to use the new person code:

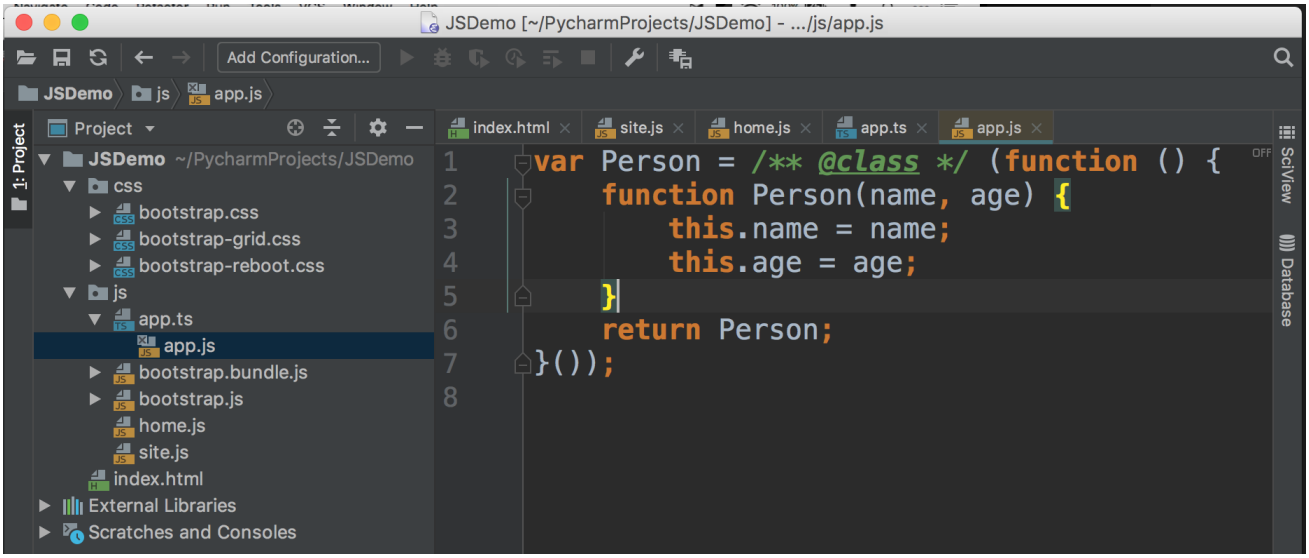


Figure 95: Figure illustrating the compiled JavaScript that is derived from the TypeScript.

```
function runAtLoad() {
  let person = new Person('Paul', 64)
  console.log(getMessage(person.name))
}
```

9.4 Using Angular

There are whole books written on Angular. Let's quickly examine how to get started running and working with an Angular app.

Create a new Angular project. PyCharm will create a directory structure and ancillary files. If you open the `app/index.html` file, you will see that this is the entrypoint for the Angular application. In the upper right side are the icons to open the page in a browser. Because this is an Angular application, we want to do a little bit more. We want to add an NPM run configuration.

There is a template for adding NPM run configurations. We can use that to fill in the appropriate information. But here is a shortcut. Open up the `package.json` file. In the gutter next to the scripts names you will see green triangles. If you click one of those buttons, it will run the NPM script (and create the run configuration).

To start a webserver to allow you to interact with your Angular application, click the button next to `"start"`. It will download any required dependencies and open a terminal at the bottom of the editor. After all of the prerequisites are finished, there will be a link pointing to `http://localhost:8000`. If you click *"Run 'start'"*, you will be brought to a web page with the application running.

PyCharm also gives you code completion inside of HTML files in Angular projects. If you type `ng` as an attribute name, the list of available options will appear.

This section has just shown a little about the support in PyCharm for Angular. In addition, there is support for linting, debugging, refactoring, and navigation. This is similar to the support for React and Vue. If you are working with single page JavaScript technologies, make sure you take advantage of PyCharm's support.

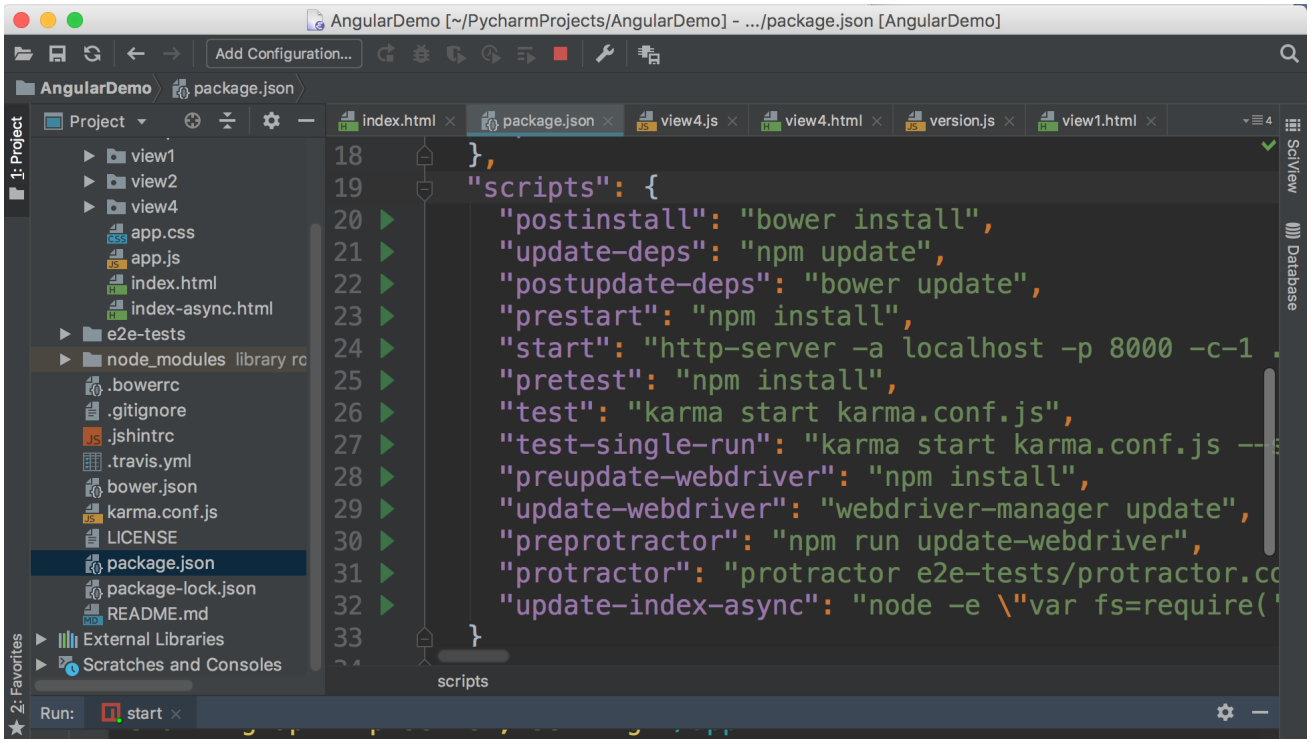


Figure 96: Figure illustrating the run buttons in the gutter of the `package.json` file.

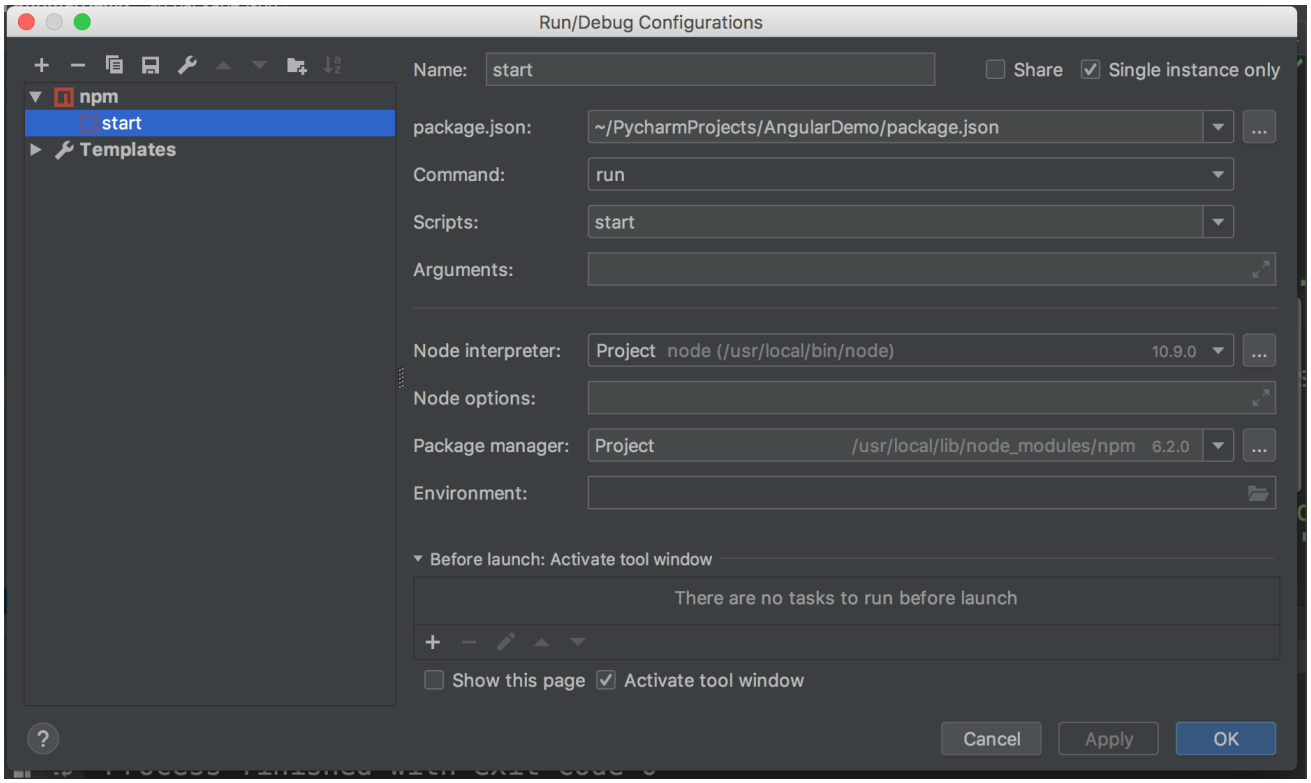


Figure 97: Figure illustrating the "Run Configuration" generated by clicking on "start" in the `package.json` file.

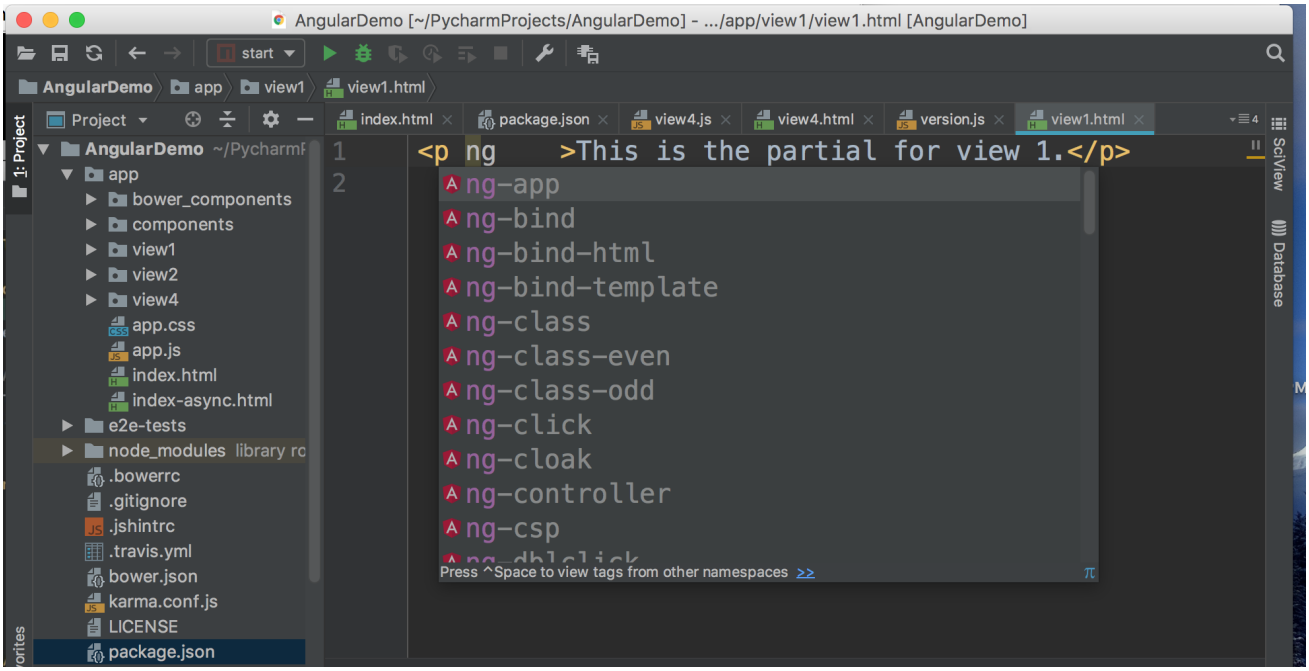


Figure 98: Figure illustrating Angular code completion.

9.5 Less

When it comes to CSS, sometimes less is more. In the case of Less[21], a language that allows us to create variables, mixins, and functions with CSS, less is definitely more.

Let's assume you have a webproject and you want to take advantage of Less. Using PyCharm, add a new file to the `css` directory that ends with `.less`. From the *New* menu there are options for various types of files, but there is no option for a Less file. That's ok, create a new file, `app.less`, and change the extension to `.less`.

When you open the file, PyCharm will present a dialog along the top that asks if you want to enable a file watcher. The file watcher will monitor the Less file and any time it changes, it will regenerate the CSS file that it creates. Click "Yes" to enable the file watcher. There will be various options, but generally the defaults will work fine.

Let's add a variable `@color1` that stores a green color. Insert the following RGB hexadecimal values into the Less file:

```
@green: #00ff00
```

In the gutter next to the variable a green button will appear. If you click it, it will open a color chooser window. From there you can customize the color if you prefer.

Note

If the filewatcher is not generating CSS, it could be because the executable is not found on your machine. If `npm` is installed on your machine you can run:

[21] https://lesscss.org/

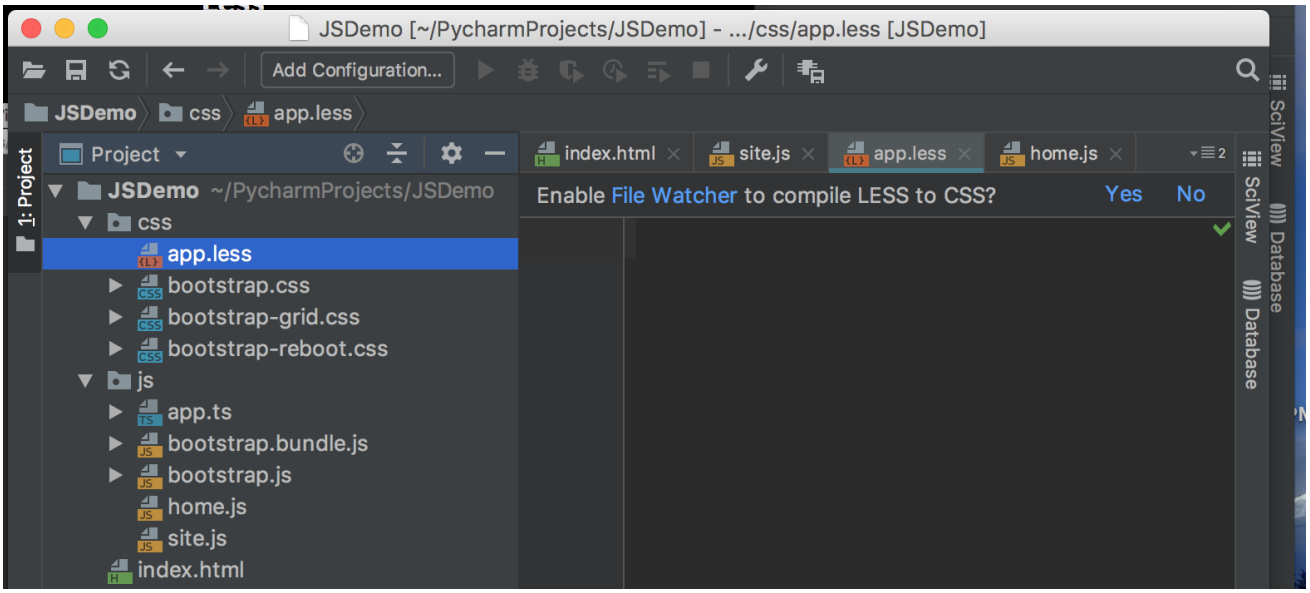


Figure 99: Figure illustrating the option to tell PyCharm to compile Less to CSS.

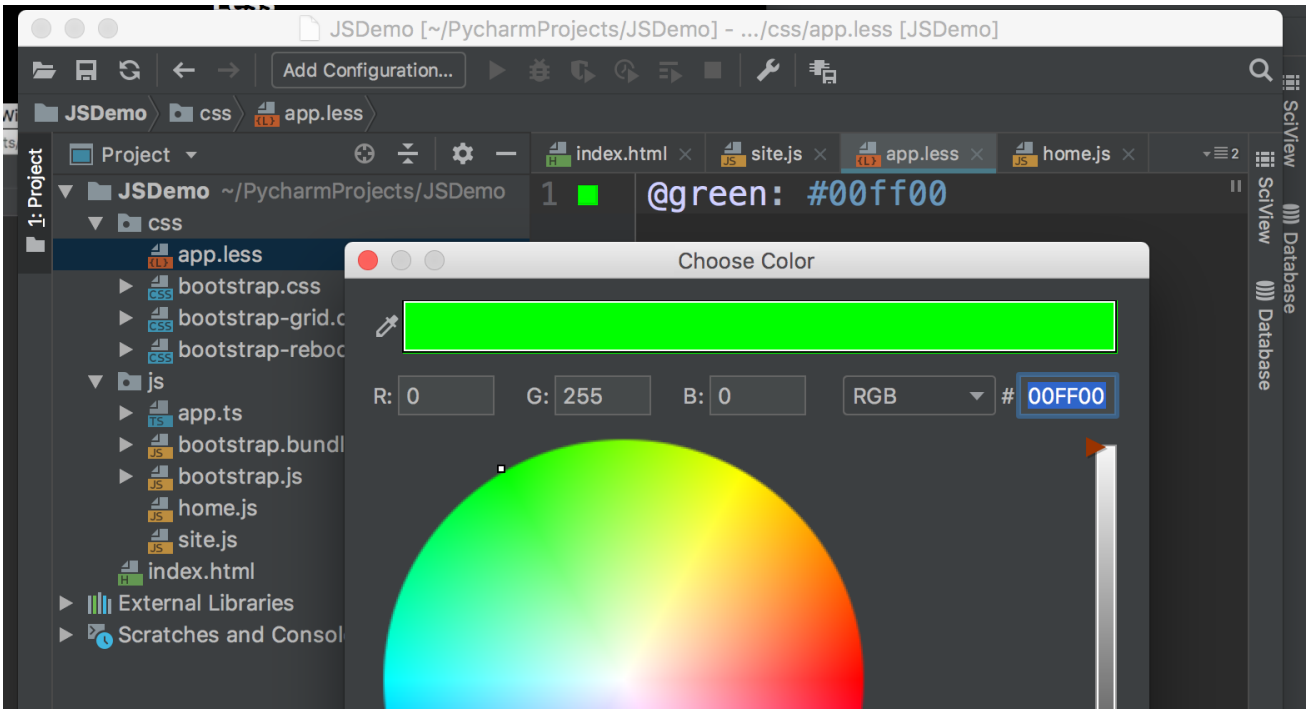


Figure 910: Figure illustrating the color picker dialog in PyCharm. You can click in the gutter next to the less variable to open it.

```
$ npm install -g less
```

On a Mac machine the executable will be found in `/usr/local/bin/lessc`. If you're on Windows, you'll find it at `C:\Users\[username]\AppData\Roaming\npm\lessc`. Use that path to update the "Program:" option on the watcher configuration page. Also, be aware that many systems have a program called `less` in their path. This is a pager utility and not what you want. As noted, you want `lessc`, the less compiler.

Update your Less file to read:

```
@green: #68ff4c;
@lite-green: #d0ffcb;
@page-color: #888888;

h1 {
  color: @green;
  background-color: @page-color;

  &:hover {
    color: @lite-green;
  }
}

h2 {
  color: @green;
}
```

This will generate the following CSS file:

```
h1 {
  color: #68ff4c;
  background-color: #888888;
}
h1:hover {
  color: #d0ffcb;
}
h2 {
  color: #68ff4c;
}
```

In this case, the CSS does contain fewer lines of code so the advantage of Less is not immediately obvious. But, as we add other styling widgets, we probably will want to reuse colors, like we are reusing `@green`. By using Less variables, we can change the color in one place and not have to worry about doing a search and replace.

Go to the `index.html` page and update it to refer to the CSS file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet" href="css/app.css">
</head>
<body>
    <h1>Welcome to JS (& CSS) Demo</h1>
</body>
</html>
```

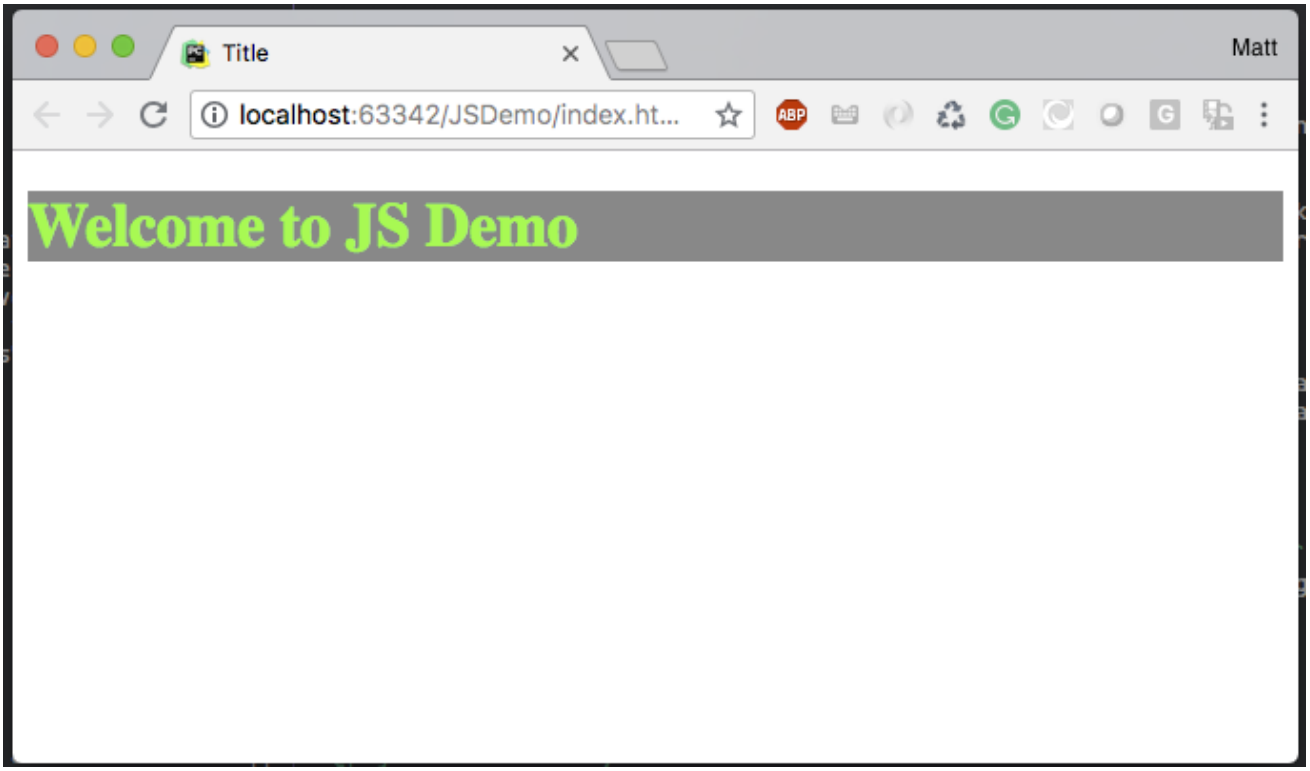


Figure 911: Figure illustrating HTML code rendering CSS generated from Less.

If you click on the browser icon, you will see a basic demo. If you hover over the `h1` tag, you will notice that the color changes.

Less is a powerful tool for managing and factoring CSS. If you are working on a project, it is a great way to control styling. PyCharm will let you create and manage CSS and Less from the editor, including code completion. Take advantage of this tooling.

9.6 Electron JS

To wrap up this chapter about JavaScript let's introduce Electron JS. This framework allows you to create GUI applications using JavaScript. It bundles the Chrome browser and Node into an executable. Notable examples of applications built upon ElectronJS include Slack, the Atom editor, VS Code, and Postman. Electron runs on multiple platforms and allows you to create desktop applications using the technologies you use for the web. This section will explore the PyCharm support for working with Electron JS.

Installing Electron JS is straightforward if you have Git and NPM installed. Let's try a simple app. Run the following:

```
$ git clone https://github.com/electron/electron-quick-start
$ cd electron-quick-start
$ npm install & npm start
```

These commands will launch a simple application. You should note that this is not a web browser, it is a stand-alone application. Let's go and explore PyCharm's support for managing Electron JS code.

Use PyCharm to create a new project from the `electron-quick-start` directory. Sadly, Electron JS uses JavaScript (Node) instead of Python. You will want to make sure that you have

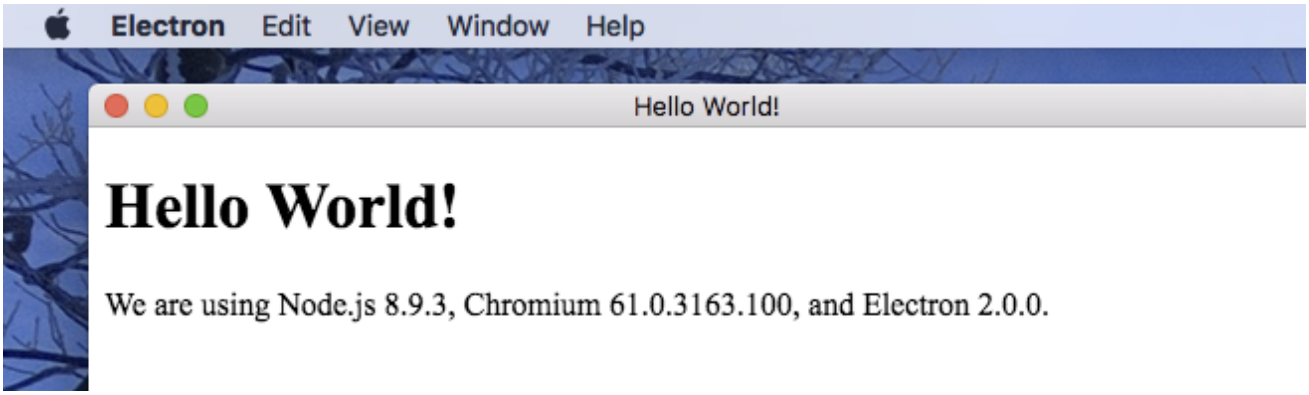


Figure 912: Figure illustrating a simple Electron JS application. Note this is not a web browser as you can see it has its own menu.

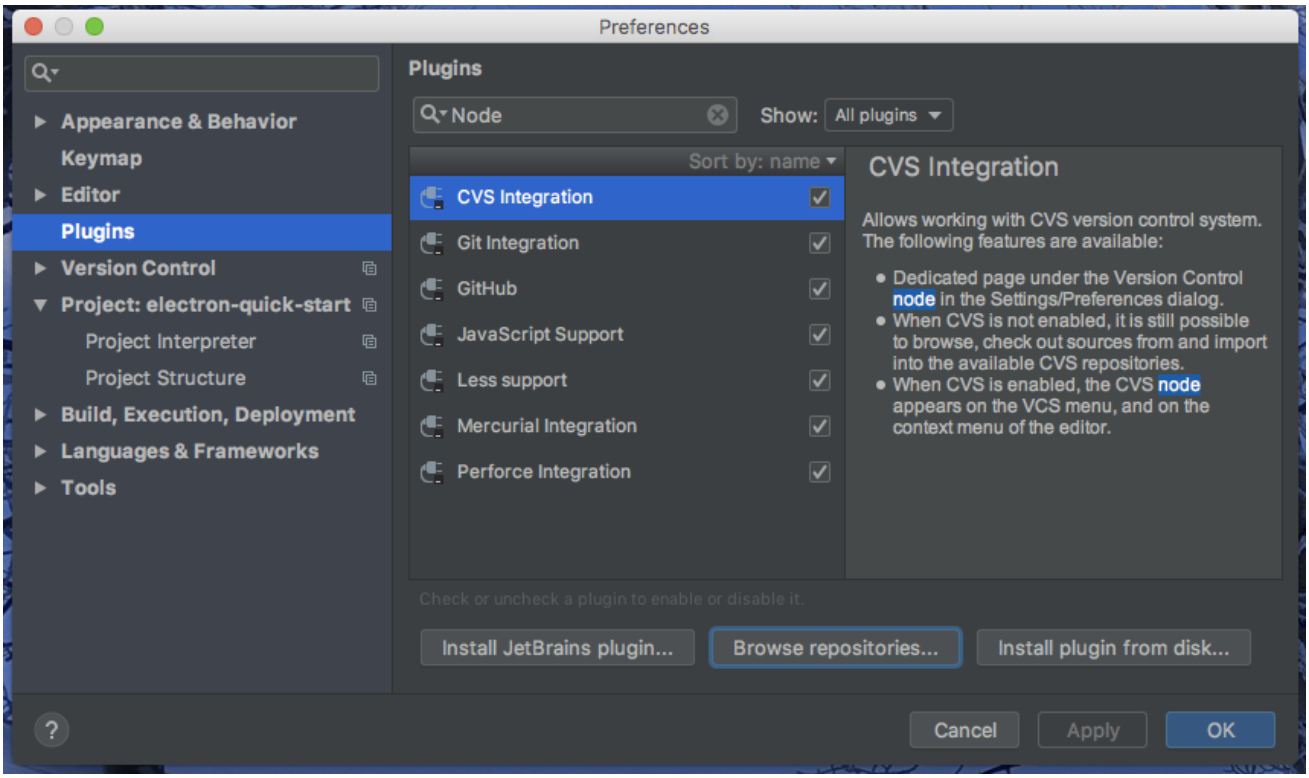


Figure 913: Figure illustrating missing Node JS plugin.

the Node plugin installed to work with the code in this project. Open up *"Preferences"* and click on "Plugins" on the left. Inside of the Plugins screen there is a search area. Type "NodeJS" in there to see if the plugin is installed. If the plugin is not there click on the "Browse Repositories..." button and search for a plugin named "NodeJS". Highlight that plugin and click on the "Install" button. After installing the plugin you will need to restart PyCharm.

We can add a Configuration to ... the Electron application. Click on the *"Add Configuration..."* button (or click on the "Run" -> "Edit Configurations..." menu). From the "Run/Debug Configurations" window click the plus button to add a configuration. There should be an option for "Node.js". You can change the "Name:" field to "Electron" and change the "Node interpreter:" value to the `electron` interpreter. You will find this inside of the `node_moudles/.bin/` directory.

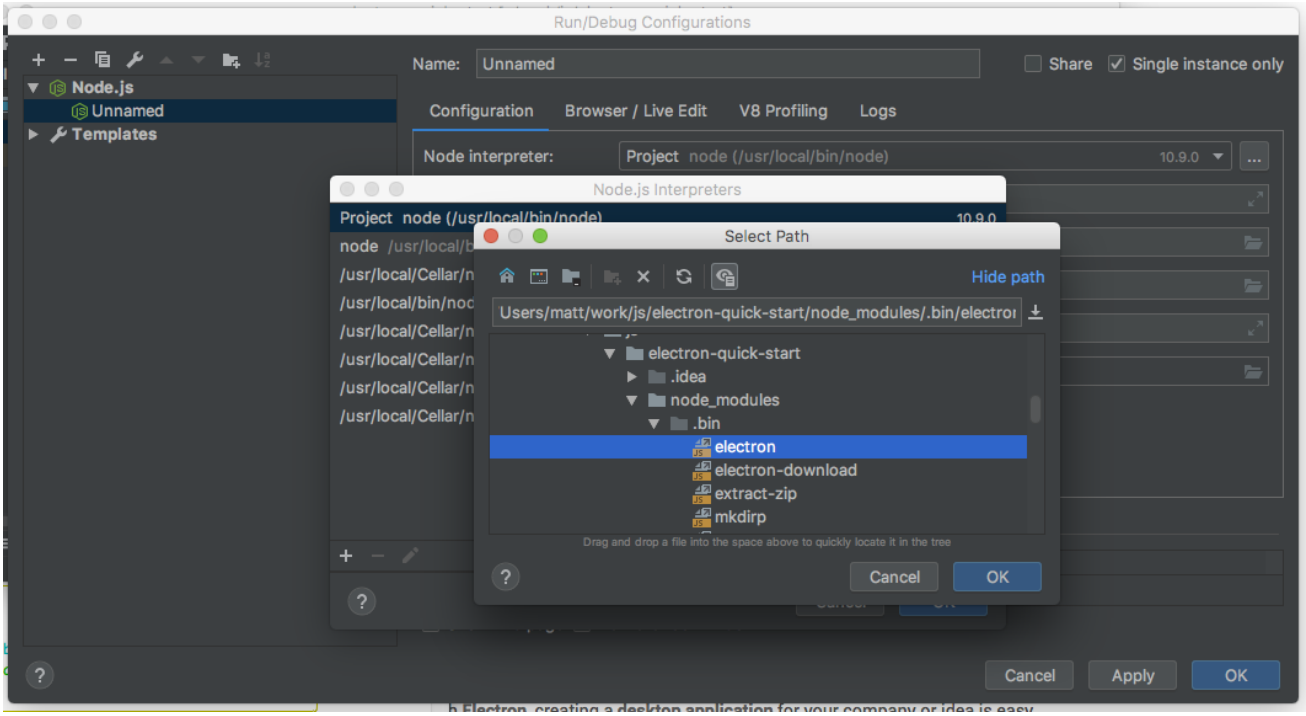


Figure 914: Figure illustrating configuring the interpreter for Electron JS.

The "Working directory:" should have the project directory (`electron-quick-start`). Leave the "JavaScript file:" blank and set "Application parameters:" to `.` (a single period). This tells Electron to run the application in the working directory.

If you run the configuration, the Electron JS application should appear. At this point you should be able to edit `index.html`. You can change the `<h1>` tag to:

```
<h1>Hello World from PyCharm!</h1>
```

If you reload the Electron JS application (View -> Reload) it will show the updated content. You don't even need to stop and restart the application.

9.7 Summary

PyCharm is a powerful editing tool. However it can require some configuration. In this chapter, we walked through how to configure PyCharm so it will run an Electron JS application as well as demonstrated its deep integration with the JavaScript language and common JavaScript frameworks.

9.8 Commands

- *"Extract Method"* - (command-alt-m, or ctrl-alt-m)
- *"Move"* - (F6)

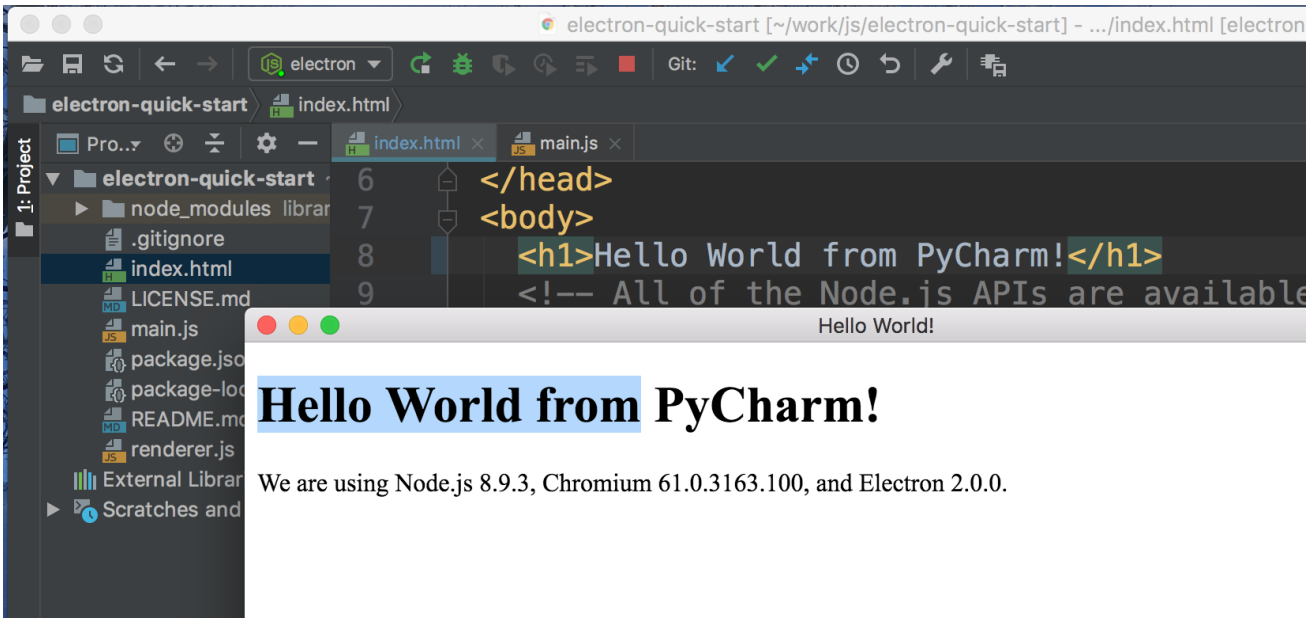


Figure 915: Figure illustrating running an Electron JS application from PyCharm.

9.9 Exercises

1. Create a project that uses Angular. Take advantage of the Angular support in PyCharm.
2. Create a project that uses TypeScript. Take advantage of the TypeScript support in PyCharm.

Chapter 10

Debugging Python applications

Debugging is a common activity for developers. Often times developers will result to using `print` functions to debug their code. Some will use a command line debugger like the `pdb` module, but few editors have the ability to debug and inspect code in the editing environment. PyCharm includes a very powerful debugging environment that is sure to make tracking down bugs faster and easier.

10.1 Debug Tools

In previous chapters, we discussed configuring an environment and running the configuration. Once you have a Python configuration, you can also debug with that configuration. Rather than executing *"Run"*, use the *"Debug"* command (command-alt-F9).

Let's make a simple Python module, `pets.py`, with the following code to illustrate debugging:

```
def remove_pet(kennel, name):
    for other in kennel:
        if name == other:
            kennel.remove(other)
    return kennel


kennel = ['Snoopy', 'Fido', 'Fido', 'Pluto']
result = remove_pet(kennel, 'Fido')
print(result)
```

To quickly create a configuration, you can right-click on the file and click on the *"Run"* or *"Debug"* command. This will create a basic Python configuration and run or debug the module. Let's add a breakpoint by clicking in the gutter left of the line of code. Add a breakpoint where `result` is assigned to a list and then debug the program.

When you debug a program with a breakpoint, the execution should stop at the breakpoint. In addition a "Debug" window will appear. Down the left side of the Debug window are buttons for *"Rerun"* (control-F5), *"Resume"* (F9), *"Pause"*, and *"Stop"* (command-F2). Below those are buttons for *"View Breakpoints"* (command-shift-F8 or ...) and *"Mute Breakpoints"*. Along the top are buttons for *"Show Execution Point"* (alt-F10), *"Step Over"* (F8), *"Step In"* (F7), *"Step In (My Code)"* (alt-shift-F7), *"Force Step Into"* (alt-shift-F7), *"Step Out"* (shift-F8), *"Run Cursor To"* (alt-F9), and *"Evaluate Expression"* (command-F8 or ...)

The "Rerun" command will restart your debugging session. "Resume" is for continuing after you have paused or hit a breakpoint. The "Pause" command will stop execution. Normally we

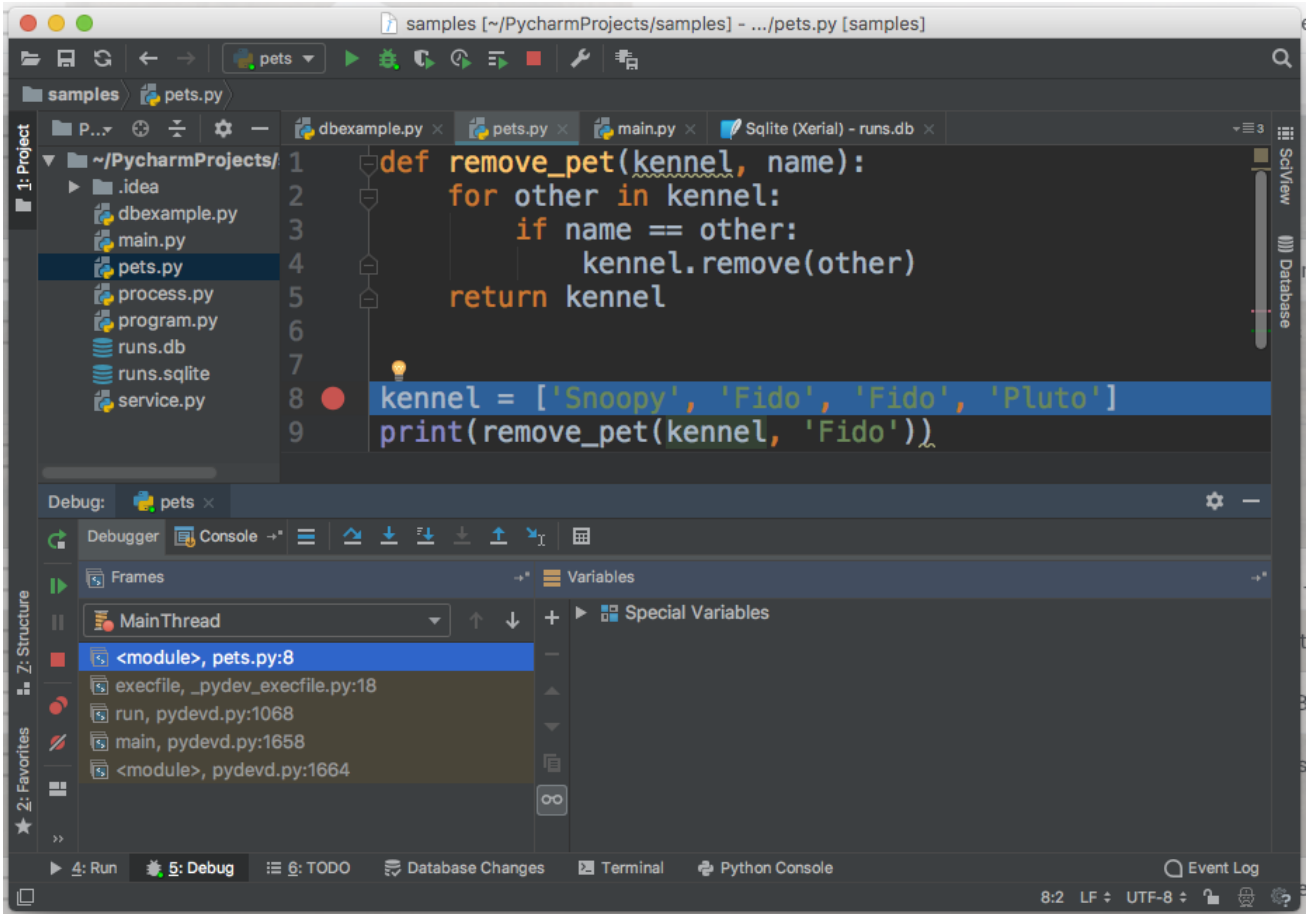


Figure 101: Figure illustrating a breakpoint in the gutter and the debugger window.

set a breakpoint to stop execution and inspect variables. But sometimes you have a long-running program and you don't know where to set the breakpoint. The pause command can help you figure out where execution is getting stuck or what your program is up to. The "Stop" command halts the current program.

The "View Breakpoints" command may seem simple. It will show all the locations where breakpoints are set. But you can also toggle breakpoints, log when they are hit, and set a condition to enable them with this command. For example, you can add a breakpoint to the `print(result)` line and tell it to only stop when `'Fido' in results`. From looking at the code, it would appear that `'Fido'` should be removed from `result`. But the conditional breakpoint will be hit. You can also log when breakpoints are hit and disable them after the first time they are hit. These are nice options that can really help when debugging a problem with code.

The "Show Execution Point" command brings your cursor back to the last line that was executed. This is useful if you are jumping around looking at other files and decide you want to quickly navigate back to the execution point. The "Step Over" command executes until the next line of code. "Step In" will step into a callable (a function or method). While "Step In (My Code)" will jump into your code, but not third-party or standard library code. PyCharm allows you to configure skipping over certain sections of code and step into will ignore those. If you really want to jump into them use the "Force Step Into" command. There is also another option, *"Smart Step Into"* (shift-F7), which doesn't have a button for it. If you have a line with several methods on it, run this command and you can choose the method you want to step into. "Step Out" hops out of

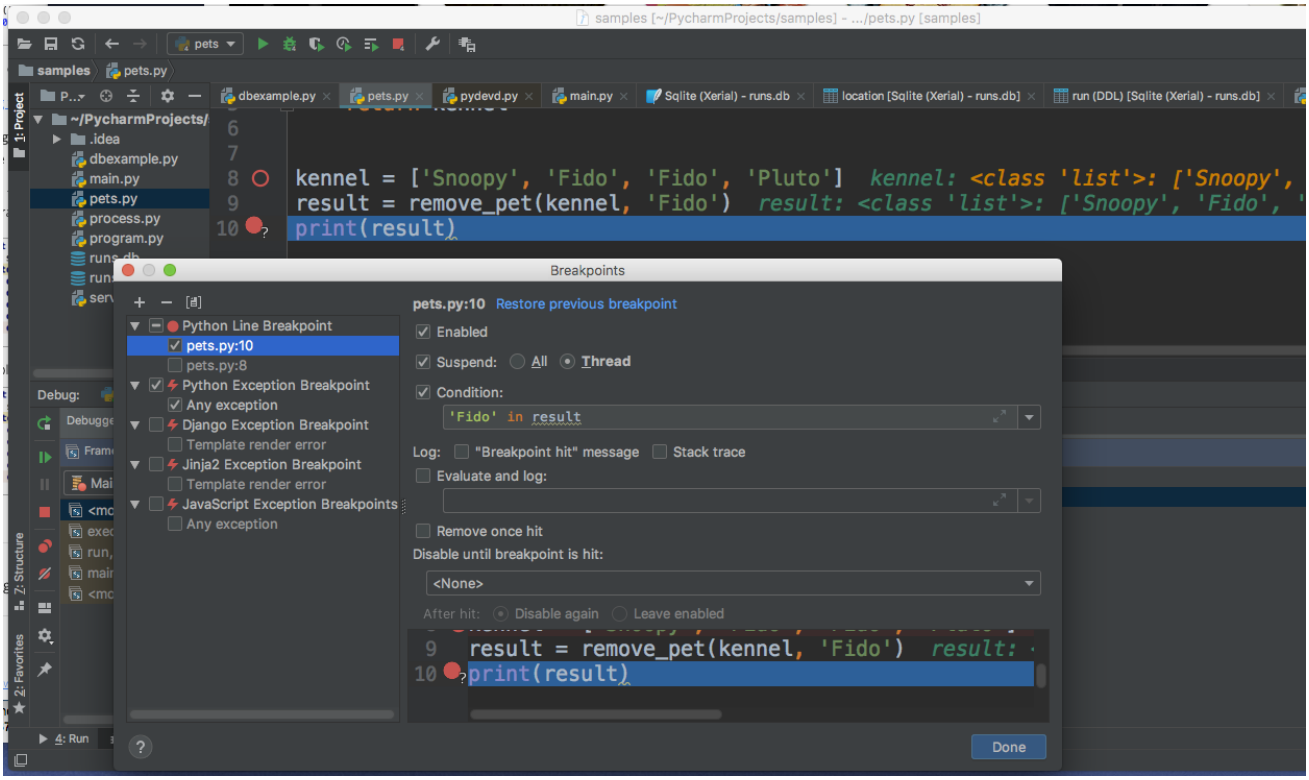


Figure 102: Figure illustrating conditional breakpoints.

the current method or function. "Run Cursor To" will allow you to resume execution to the line the cursor is on. "Evaluate Expression" lets you type in a Python expression much like the REPL.

If you are debugging this program step into the `remove_pet` function. You will see that PyCharm has *inline debugging* and shows the values of the variables next to the line where they are set or changed. The inline debugging values are color-coded. They are green if they haven't changed and change to orange when they are updated or change.

In the Debug window, there are two tabs, "Debugger" and "Console". In the Debugger tab, you can see the stack in the "Frames" section and the objects in the "Variables" section. The variables section shows the values of the variables. But, it also lets us interact with the variables. There is a *"Set Value..."* command (F2) that let's you override a variable. You can navigate to where the variable was created with *"Jump to Source"* (F4).

PyCharm has the notion of *watches*. Watches are special variables that exist inside a function or method. Typically you would set a watch to an expression. In the case of our code, we could create a watch using *"New Watch"* with the value of `'Fido' in result`. PyCharm would show this as a boolean and its value would dynamically change if the `result` variable ever had `'Fido'` in it.

There is also the Console tab in the Debugger window which shows any output to standard out and also allows you to enter input to standard input.

If you step through the loop in `remove_pet` you will see that the code only removes the first `'Fido'`. This is due to the iteration behavior of Python. While iterating over a list, Python keeps track of the index of iteration. If you remove something at the current index, all the other items are shifted to the left. Hence, when we remove the first `'Fido'`, the second `'Fido'` shifts into its place, but then the iteration moves onto the next item, which is now `'Pluto'`. One solution to this situation is to loop over a shallow copy of the list using a slice construct and then update the original list.

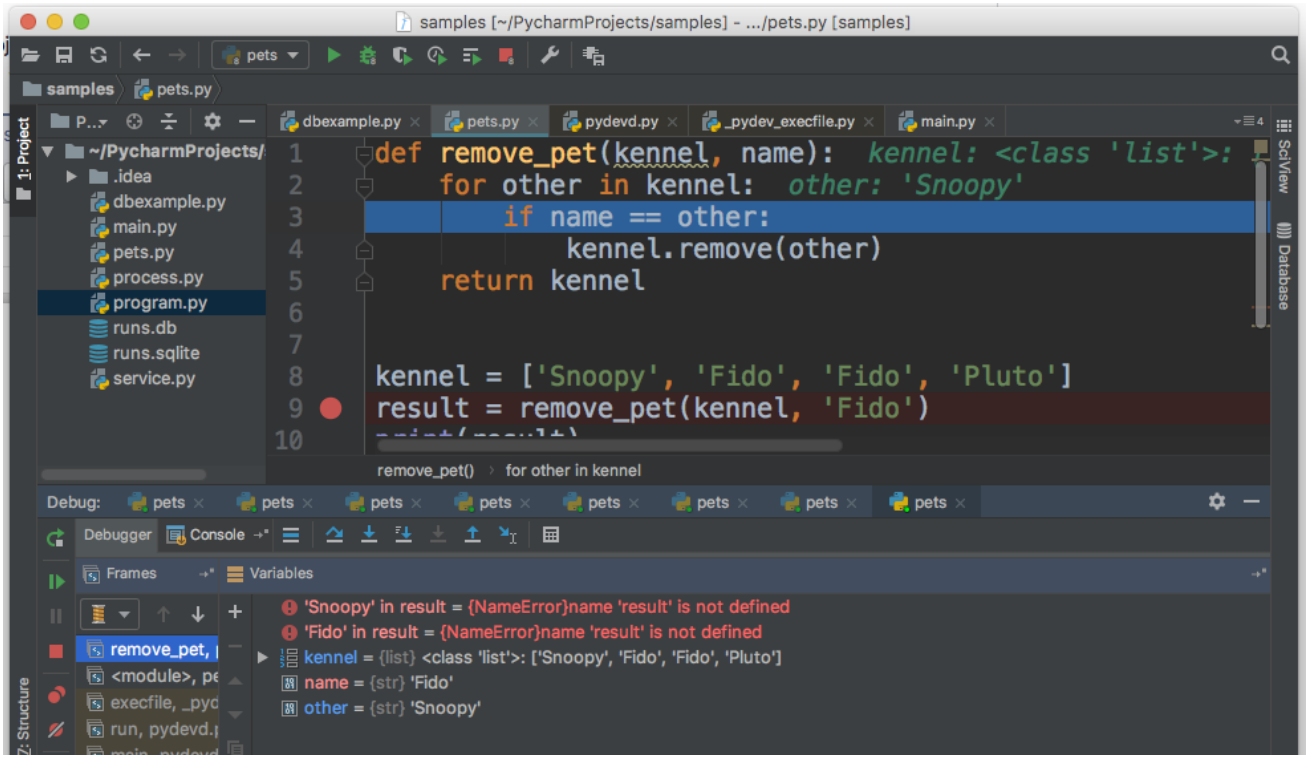


Figure 103: Figure illustrating debugging. Note that we have stepped into `remove_pet` and the value for `kennel` and `other` are shown inline.

The code could read:

```
def remove_pet(kennel, name):
    for other in kennel[:]:
        if name == other:
            kennel.remove(other)
    return kennel


kennel = ['Snoopy', 'Fido', 'Fido', 'Pluto']
result = remove_pet(kennel, 'Fido')
print(result)
```

Another option would be to use a pure function and not mutate the list that was passed in but rather return a new list:

```
def remove_pet(kennel, name):
    result = []
    for other in kennel:
        if name != other:
            result.append(other)
    return result


kennel = ['Snoopy', 'Fido', 'Fido', 'Pluto']
result = remove_pet(kennel, 'Fido')
print(result)
```

10.2 Summary

This section provided an overview of the tools for stepping through code and setting breakpoints. These powerful tools are a great way to debug and understand code. Python is a dynamic language, getting used to tools like the REPL, watches, and conditional breakpoints can simplify and accelerate reading, writing, and debugging code.

10.3 Commands

- *"Debug"* - (command-alt-F9).
- *"Rerun"* - (control-F5)
- *"Resume"* - (F9)
- *"Pause"*
- *"Stop"* (command-F2)
- *"View Breakpoints"* (command-shift-F8 or ...)
- *"Mute Breakpoints"*
- *"Show Execution Point"* - (alt-F10)
- *"Step Over"* - (F8)
- *"Step In"* - (F7)
- *"Step In (My Code)"* - (alt-shift-F7)
- *"Smart Step Into"* - (shift-F7)
- *"Force Step Into"* - (alt-shift-F7)
- *"Step Out"* - (shift-F8)
- *"Run Cursor To"* - (alt-F9)
- *"Evaluate Expression"* - (command-F8 or ...)
- *"Set Value..."* - (F2)
- *"Jump to Source"* - (F4)
- *"New Watch"*

10.4 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/7-debugging` to try out creating breakpoints.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/7-debugging` to try out using the debugger to stop and start control flow.
3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/7-debugging` to try out using the debugger to add a conditional breakpoint.

Chapter 11

Packages

One of the main draws to Python is the extensive third-party libraries available. The Python Package Index [#] has over 160,000 packages and is growing bigger every day. When you need to perform a new task with Python, you should check if one of the available packages provides that functionality. If oner doesn't exist, you might want to build your own packages. PyCharm has support for both.

11.1 Working with Existing Packages

In this section we are going to look at taking an existing package and create a custom version. A common developer activity is to update a package. You may want to add a feature or fix a bug. Let's walk through creating a creating a PyCharm project around an existing package, initializing a virtual environment, and installing the package in developer mode to work on it locally.

Let's show how to create a custom version of the popular Requests HTTP client library. The first thing we need to do is check out the project. Run this command to checkout the project:

```
git clone https://github.com/requests/requests.git
```

Use PyCharm to create a new project with that directory in the "Location:" field. Expand the "Project Interpreter" dropdown and select "New environment using Virtualenv".

If you open the `setup.py` file you may notice that it says "No Python interpreter configured for the project" along the top. If that is the case, point the project interpreter to the virtual environment. After than you should see PyCharm complaining about missing package requirements. Click on "Install requirements".

Let's make a simple change to the `requests/__init__.py` file. We will add a `print` function to show that the file has been changed. Below the docstring in that file enter the line:

```
print("THIS IS OUR CUSTOM VERSION")
```

At this point if you tried to use the Requests library from the virtual environment, Python would complain with a `ModuleNotFoundError` (unless you have it installed in your system library). Even though you created a virtual environment and installed the dependencies for Requests, the Requests library hasn't been installed.

What you want to do is install Requests in *development* mode. This installs it into the virtual environment and also allows you to edit the local copy of the requests package. Any edits will

[22] https://pypi.org

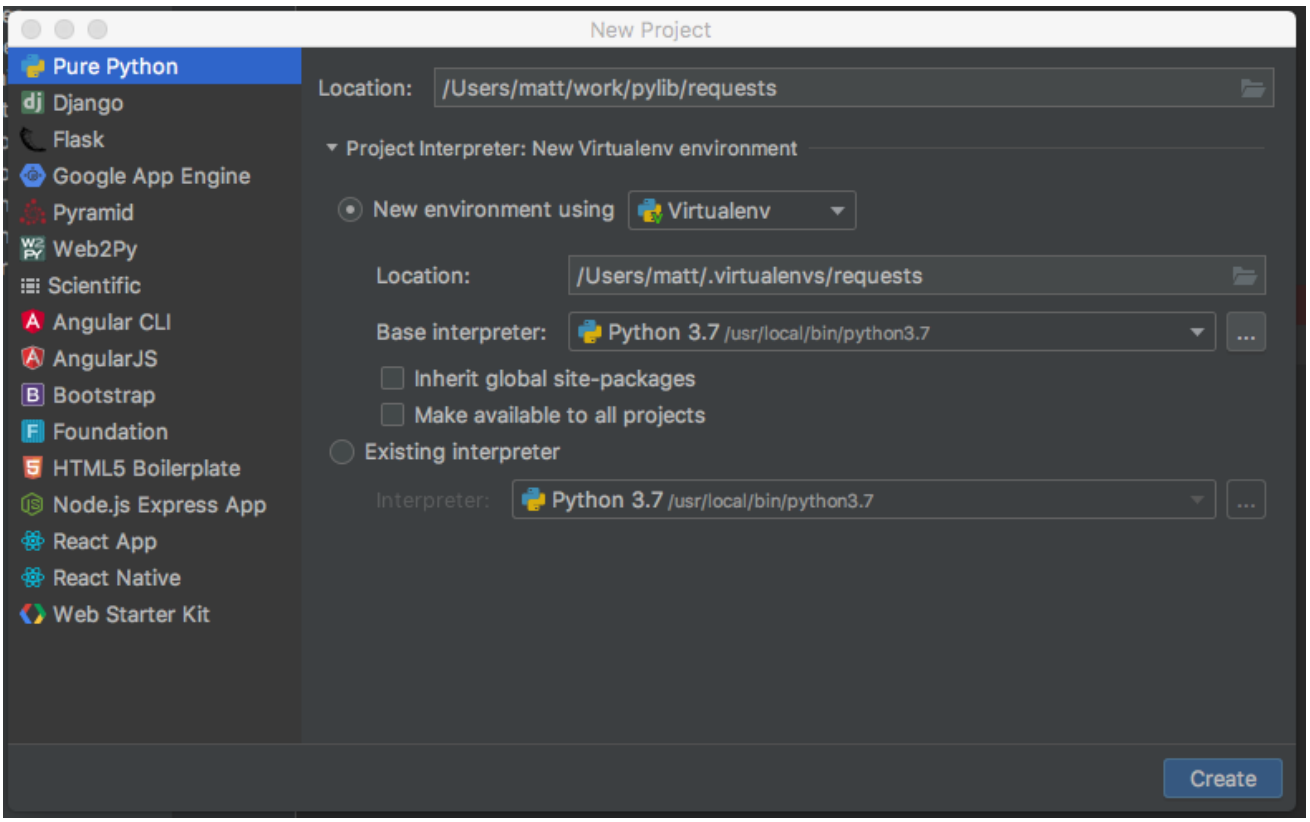


Figure 111: Figure illustrating create a new project with a new virtual environment.

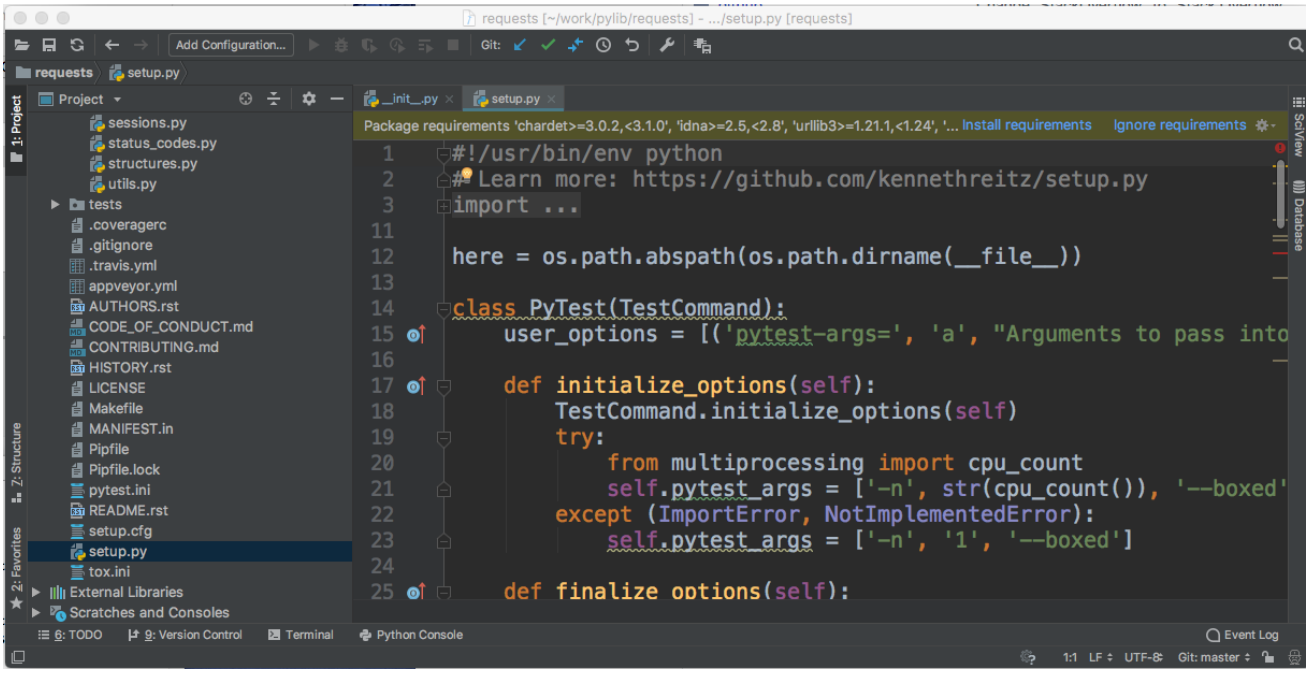


Figure 112: Figure illustrating missing requirements.

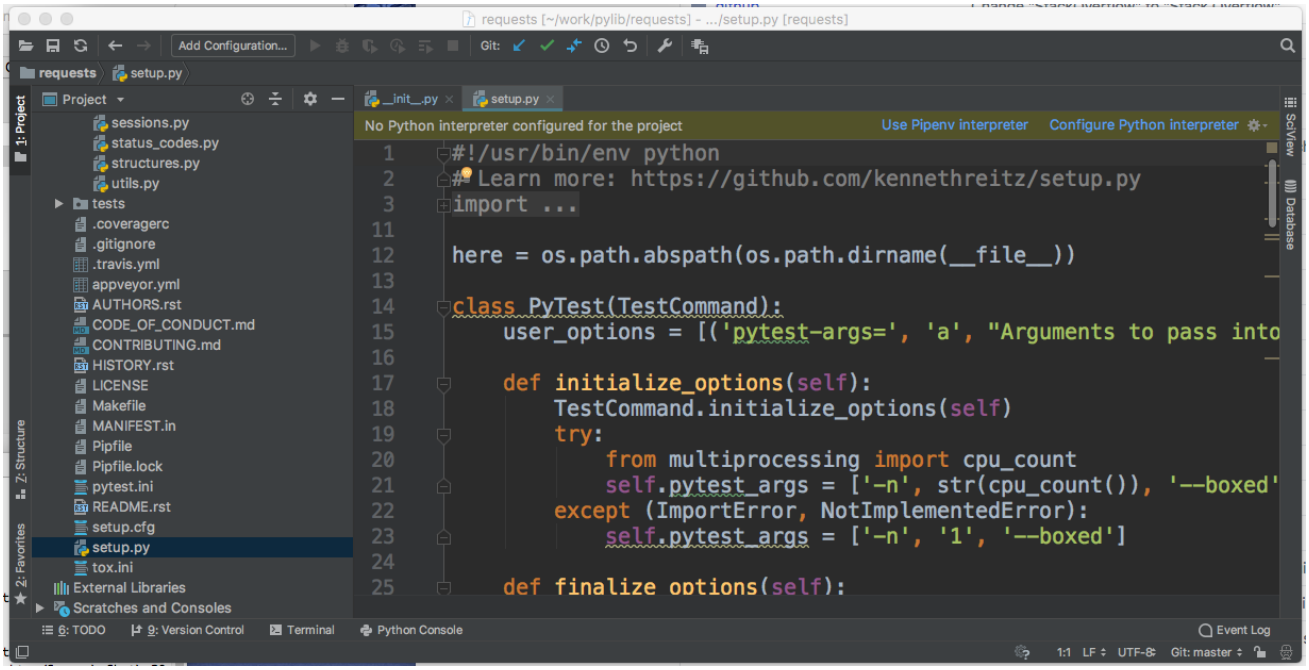


Figure 113: Figure illustrating `setup.py` file not having an interpreter associated with it.

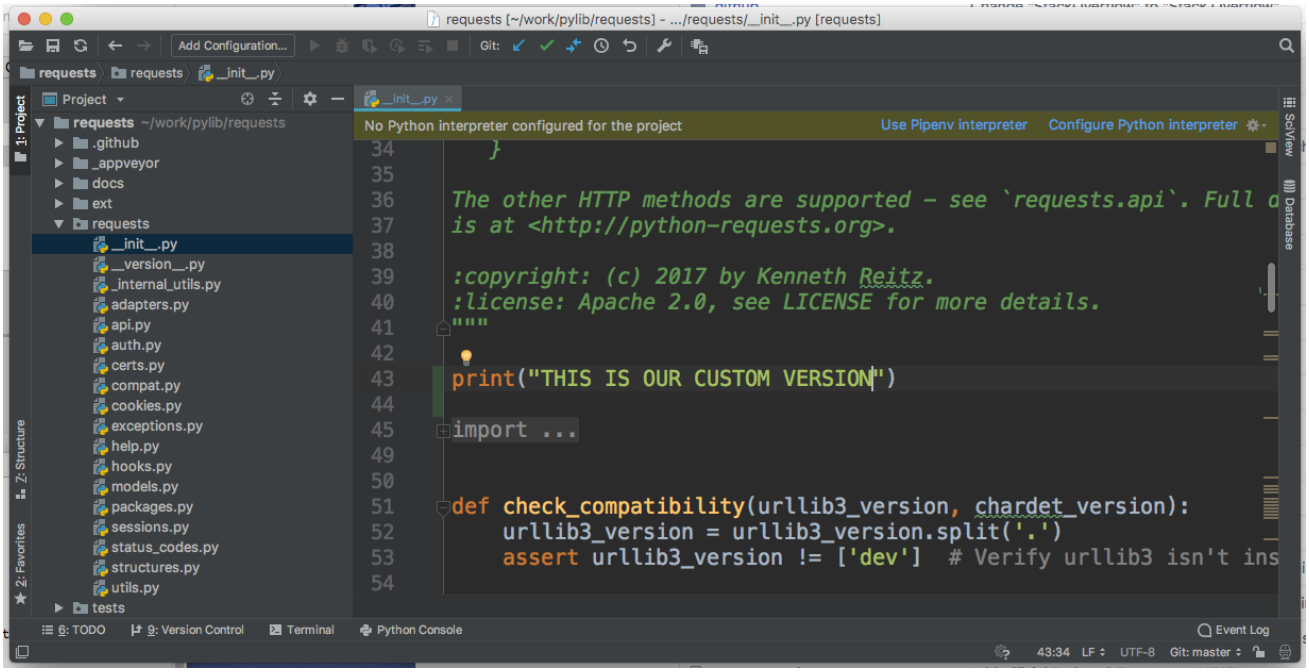


Figure 114: Figure illustrating adding a custom line of code to the Requests project.

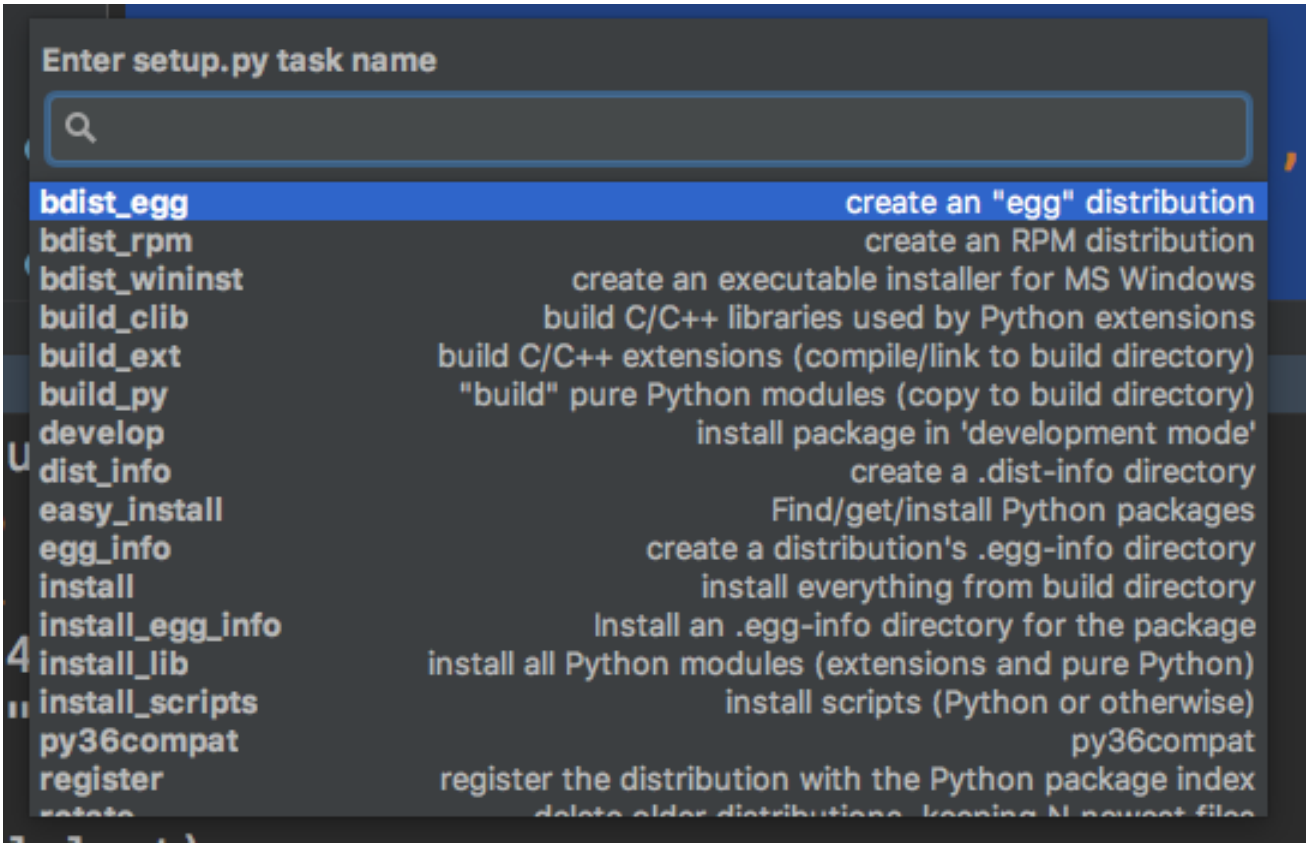


Figure 115: Figure illustrating running a `setup.py` task from PyCharm.

be reflected immediately in the virtual environment. You can do this from the command line by entering the virtual environment and typing:

```
$ python setup.py develop
```

Because this is a PyCharm book, let's do it from PyCharm. Click on "Tools" -> "Run setup.py Task...". Click on "develop".

At this point, when you load the Requests library from your virtual environment, you should have access to the version you are working on. As you make changes to the library you will be able to get these changes as well.

The output from using the new library should look like this:

```
>>> import requests
THIS IS OUR CUSTOM VERSION
```

11.2 Creating New Libraries

In this section we are going to create a library in Python. In order to have a library, you will need a few things. We will need the code implementing the library functionality and a `setup.py` file that declares meta information about the code (how it is installed and packaged). The Python code for a library can reside in a module (just a file ending in `.py`) or a package (a directory with a `__init__.py` file in it).

Let's create a calculator library. Create a new "Pure Python" project called "Calculator". Inside of the project tab there will be a folder call "Calculator". Run the *"New..."* (ctrl + n) command.

From the new popup select "Python Package", and enter "calculator" for the package name (PEP 8 suggests that we name packages in lowercase). If you expand the new package folder you will see that PyCharm created a `__init__.py` file for you. Open that file and add the following code:

```
class Calculator:

    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y
```

If you open the "Python Console" you will have access to the `calculator` package. You can run the following code:

```
>>> import calculator
>>> calc = calculator.Calculator()
>>> calc.add(2, 5)
7
```

If we want to share the `calculator` package so other code can take advantage of it, we have a few options. We can copy the package into the code where we need it. We can also update the `PYTHONPATH` environment variable to point to the directory where `calculator` resides. The former is not recommended as copying code in general is something to be weary of. If you have bugs to fix or features to add, you have to do it in more than one place. The latter is also not recommended as editing the environment variable for each package is tedious. A better option is to create a proper Python library and use the standard Python tools (virtual environment and pip) to install the library.

Let's walk through the necessary steps to convert this code into a reusable library. We are missing a `setup.py` file, so let's create one. This is a file that is rarely created and the format is a little convoluted. Because this file is typically created once you don't deal with them much. Also, the `setup.py` file can be confusing. It is the entrypoint to register and upload to PyPI. It indicates what the package metadata is: who is the author, the version, license, homepage, etc. It controls how source tarballs, binary, and wheel files are created. It specifies which files (both code and data) files belong in a distribution. It also instructs Python on how to install a package. It can specify dependencies as well as console executables that should be installed. In addition it can specify test suites and a test runner.

PyCharm has a command that will help stub out a `setup.py` file. Run the "Tools" -> "Create setup.py" command. This will bring up a popup where you can provide data about the project. Put in a version number (`0.1.0` is a good starting point), url, license, author, email, and description. After you run the command you will have a `setup.py` file that looks like this:

```
from setuptools import setup

setup(
    name='Calculator',
    version='0.1.0',
    packages=['calculator'],
    url='https://calculat.or',
    license='MIT',
    author='Matt Harrison',
    author_email='matt@calculat.or',
    description='A Pythonic calculator'
)
```

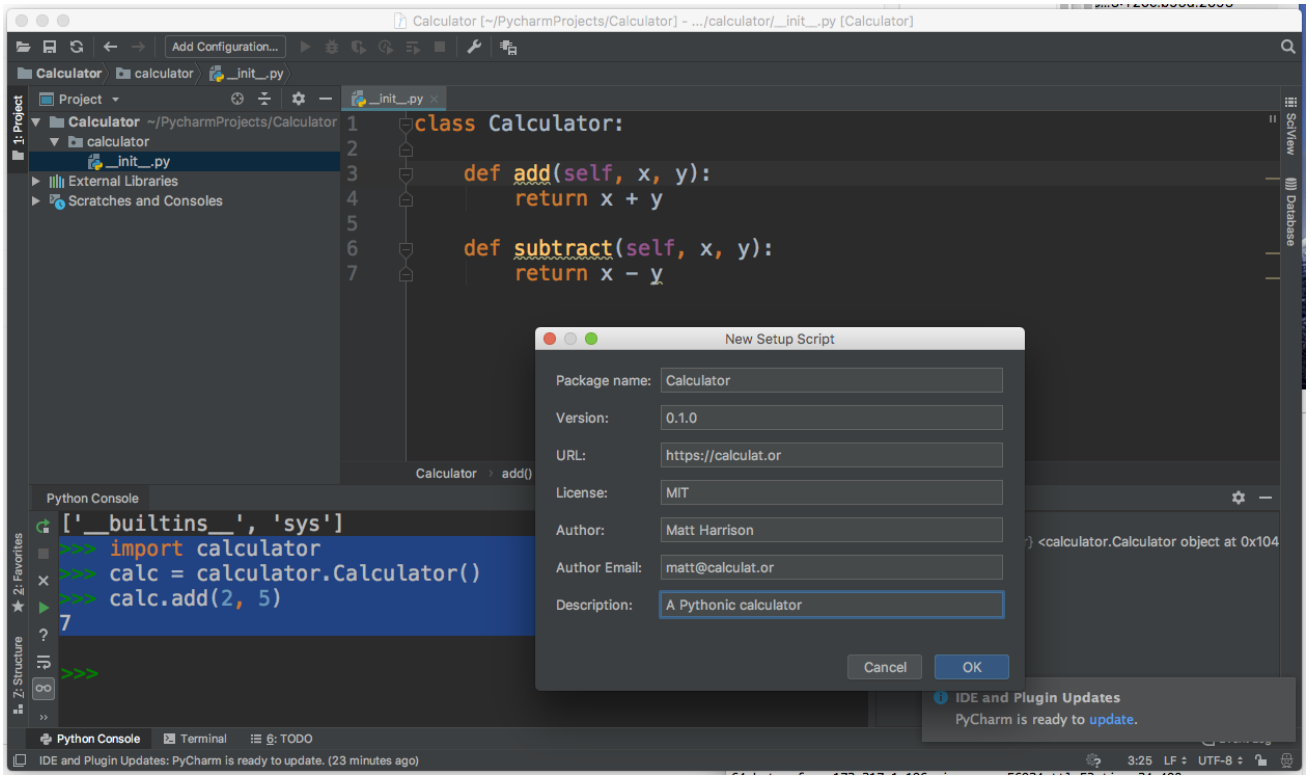


Figure 116: Figure illustrating filling in the "Create setup.py" command window options.

In our case, the important part of this file is that the `packages` parameter is set to a list with the name of our package in it. This parameter indicates which folders will be bundled up in this library.

At this point we could register this project on PyPI, create a source distribution, and upload to PyPI. Since this is a silly example, I wouldn't do that, but the command would be `python setup.py register sdist upload`. (In addition the "Calculator" name is already used on PyPI so uploading would fail). If the name had not been taken we could install the package with `pip install Calculator`.

If you wanted to use this library in another project you were working on, you could go through the uploading to PyPI process and use `pip` to install. Sometimes you might want to skip using PyPI. Here are two options for installing packages without uploading to PyPI. If you wanted to update the Calculator code while working on your new project, a better option is to run `pip install -e .` using the project virtual environment from inside of the Calculator directory. If you don't need to edit Calculator, you can run `pip install .`.

11.3 Summary

In this chapter we discussed editing existing packages from existing code and starting a new Python package from scratch. We explored how we can set up our environment to use packages in development mode. We also looked at creating a new library from scratch. We can use PyCharm to stub out the contents of the `setup.py` file for us. Once we have that file, we can upload to PyPI or install our library locally bypassing PyPI.

11.4 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/8-python-packages` to create a project.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/8-python-packages` to make a package from a project.
3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/8-python-packages` to add a feature to a project.
4. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/8-python-packages` to create a `setup.py` file.
5. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/8-python-packages` to use the library you just created.

Chapter 12

Performance and Profiling

12.1 Introduction to Profiling

Python optimizes for developer speed. You can take advantage of the vast libraries and legibility to create applications very quickly. You may discover that although your code is working, it might not give the runtime performance you would like. You might have a website that you would like to be very responsive, but it is taking over a second to return responses. Rather than going through all of your code and trying to optimize each and every line, we can use PyCharm's profiling tools to figure out where the slowdowns are occurring. Oftentimes our intuition about what is slow is wrong, so taking advantage of these tools can be a great asset.

12.2 Profiling an Application

In this chapter, we will look at the performance of a basic Python application. The application is a sentiment classifier application. It will query the Talk Python website for podcasts that match a given search term. Then using the description that comes back it determines whether that description is positive and negative. It also reports on how long it took to run. This is a simple example but it illustrates both I/O and some calculations that can be used in various programs that may appear when profiling.

The process of calculating sentiment requires downloading three lists. Two contain words that are positive and negative. The final list is a collection of words called *stop words*. These are words that we want to ignore because they are common (e.g. a, and, the) and just take up space. To calculate if the words of the description (called *corpus* by natural language processing engineers) are positive, we strip out any stop words, then calculate the difference, and divide by the number of words. The maximum value, 1.0, would indicate all positive words, while -1.0 would indicate all negative words.

Here is the program to calculate sentiment of Talk Python descriptions. The file is named `sentiment.py`. It is also available on the Talk Python Mastering Pycharm Course repository at GitHub[22].

```
"""
* get stopwords
* get positive words
* get negative words
```

[23]https://github.com/talkpython/mastering-pycharm-course/tree/master/extra_samples/profiling/

```
* get corpus
* score = (count positive - count negative) / (num words)
  bounds [-1, 1]
"""
import os
import re
import time
import urllib

import requests

BASE = 'https://raw.githubusercontent.com/jeffreybreen/'\
  'twitter-sentiment-analysis-tutorial-201107/master/'\
  'data/opinion-lexicon-English/'
NEG_URL = f'{BASE}negative-words.txt'
POS_URL = f'{BASE}positive-words.txt'

STOPWORDS_URL = 'https://raw.githubusercontent.com/'\
  'stanfordnlp/CoreNLP/master/data/edu/stanford/nlp/'\
  'patterns/surface/stopwords.txt'


def get_url(url, fname):
    if os.path.exists(fname):
        return
    fin = requests.get(url)
    data = fin.text
    with open(fname, 'w') as fout:
        fout.write(data)


def get_line_words(fin):
    r"""
    >>> get_line_words([': ."hello world, my name!\n', 'is matt.\n'])
    ['world', 'my', 'name', 'is', 'matt']
    """
    book_words = []
    word_re = re.compile(r'(\w*)')

    for line in fin:
        for word in line.split():
            # don't use match it checks start of line
            valid = word_re.search(word)
            if valid and valid.group(1):
                book_words.append(valid.group(1))
    return book_words


def get_posts(query):
    enc = urllib.parse.urlencode({'q': query})
    url = f'http://search.talkpython.fm/api/search?{enc}'
    res = requests.get(url)
    return [d for d in res.json().get('results')]


def get_score(words):
    if not len(words):
```

```
        return 0.0

    neg = [line.lower().strip() for line in open('neg.txt')]
    pos = [line.lower().strip() for line in open('pos.txt')]
    stop = [line.lower().strip() for line in open('stop.txt')]

    pcount = 0
    ncount = 0
    remove = 0
    for word in words:
        if word in stop:
            remove += 1
            continue
        if word in pos:
            pcount += 1
        if word in neg:
            ncount += 1

    score = (pcount - ncount) / (len(words) - remove)
    return score


def main():
    get_url(NEG_URL, 'neg.txt')
    get_url(POS_URL, 'pos.txt')
    get_url(STOPWORDS_URL, 'stop.txt')
    posts = get_posts('numpy')

    for post in posts:
        words = get_line_words([post['description']])
        print(f"{post['title'][:20]:20} Score: {get_score(words):6.3f}")


if __name__ == '__main__':
    start = time.time()
    main()
    print(f"\ntook {time.time() - start:.3} seconds")
```

Note that this application uses the third-party Requests library. You will need to install that into your active virtual environment to run the code.

Create a run configuration and execute the file. The output will print out the title and the score of the description. On my machine (2015 MacBook) it prints:

```
Python in Finance    Score: -0.045
Serverless software  Score:  0.000
...
PyPy - The JIT Compi Score: -0.034
Effective Python     Score:  0.000
Python in Brain Rese Score:  0.093
took 2.04 seconds
```

The report states that it takes almost two seconds to run. Let's use the profiling capabilities of PyCharm to figure out where our time is being spent.

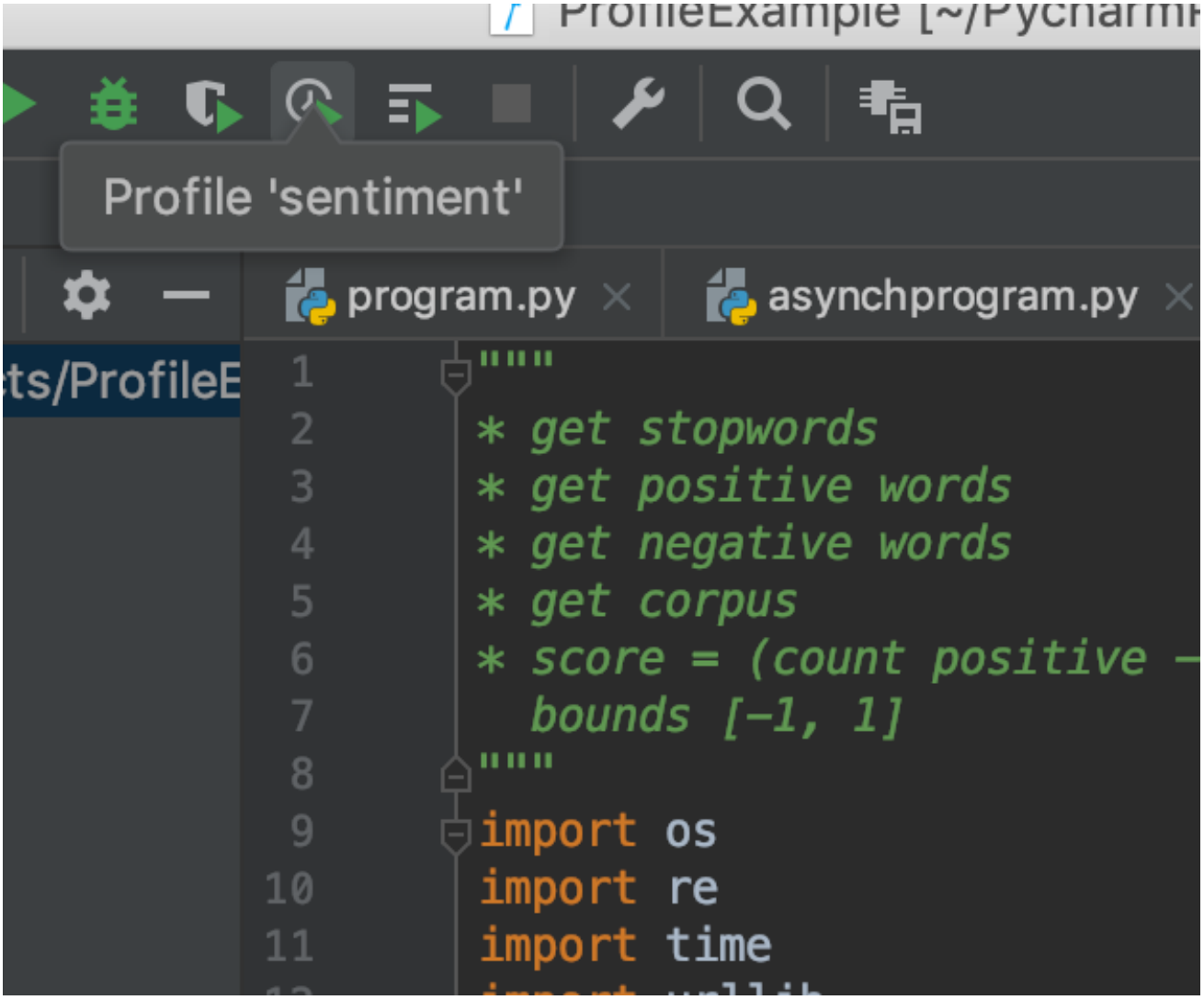


Figure 121: Figure illustrating clicking the profile button

12.3 Capturing Profile Information

To capture profile information, run the "Profile 'sentiment'" command (where sentiment is the name of the run configuration in the PyCharm project that will execute the file above which we have named `sentiment.py`). You will need to click the button for this that has a clock on it.

After running the profiler, you should see a tab open that has two tabs inside of it. The default tab, "Statistics", contains a table of profile information. Along the top of the table are the column names. The name column lists the function or method that is being profiled. The default view should be sorted by function (or method) runtime. The function that took the longest time to run is first. The "Call Count" column indicates how many times that function was called. The "Time (ms)" column shows how much time the code in that function and any functions that it called took. The last column, "Own Time (ms)", indicates how much time was spent in the code in that function. In functions that don't call any other functions the last two columns will be the same.

This table informs us that the `get_score` function took over 72% of the time. Of that, 27% was spent in the `get_score` function itself. The table does not inform us of what code called `get_score` or what `get_score` calls. This is where the "Call Graph" tab comes in. This tab has a graph of

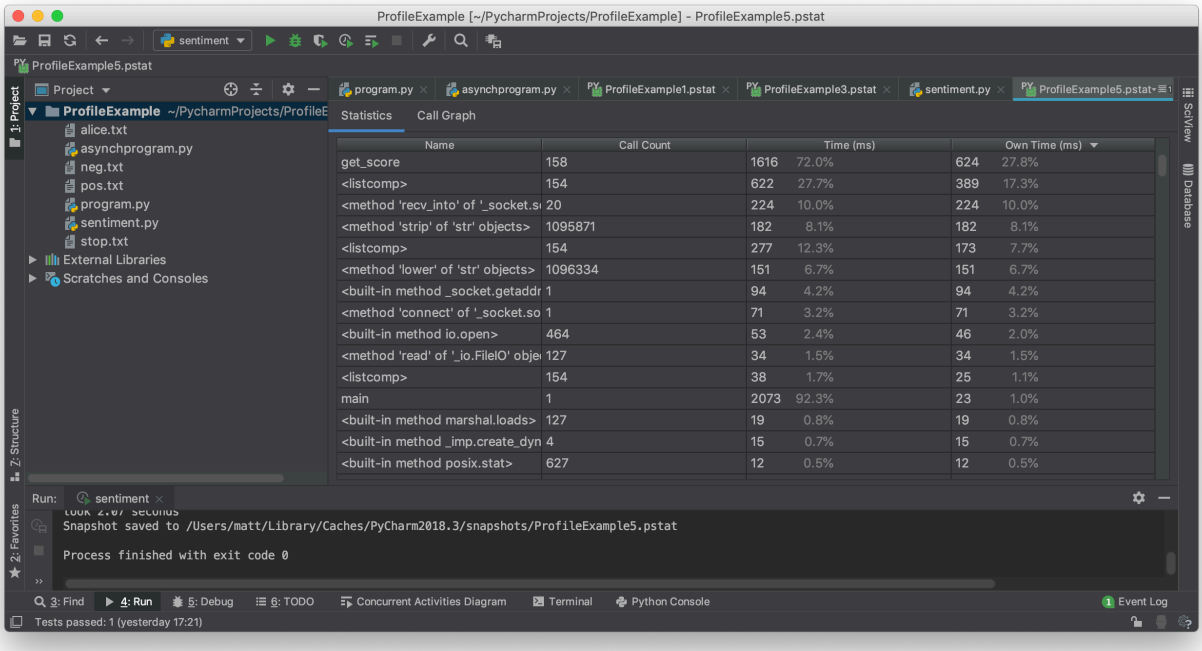


Figure 122: Figure illustrating the profile information table.

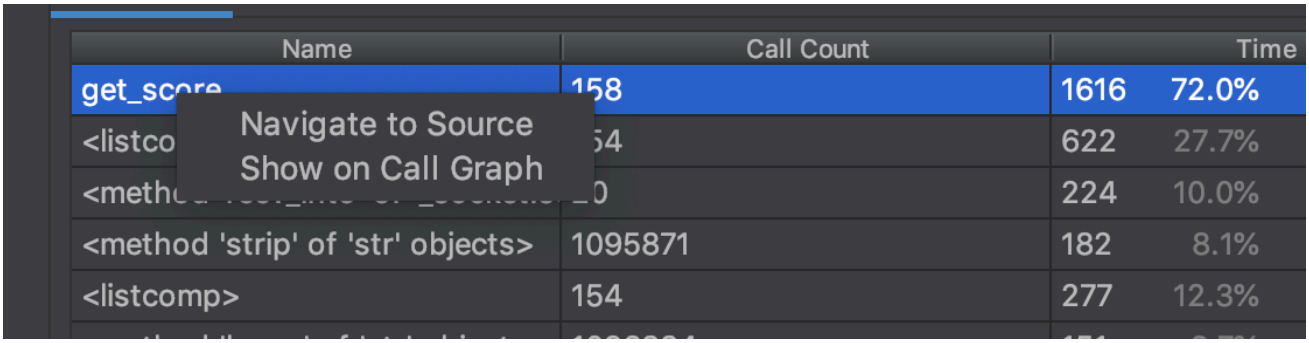


Figure 123: Figure illustrating right-clicking on the function to get to the graph.

which has arrows indicating which functions call other functions. From the "Statistics" tab, we can right-click on a function and select "Show on Call Graph" to view the function in the graph.

It turns out that the `main` function calls `get_score` which uses two list comprehensions for the majority of its work. The rest of `get_score` takes 27% of the time. This code is CPU bound, so it is possible that we can speed it up. Common ways to speed up code are parallelizing it (running on multiple CPUs), using a smarter algorithm, or caching.

If we look around in the graph we will see that there is also some I/O-bound code. The calls to `get_url` and `get_posts` fetch data over the network. This code is I/O-bound, we are waiting for the network to return the results of the page. In order to speed this up, we would need to migrate to asynchronous code, use threading to batch up the waiting, or use caching.

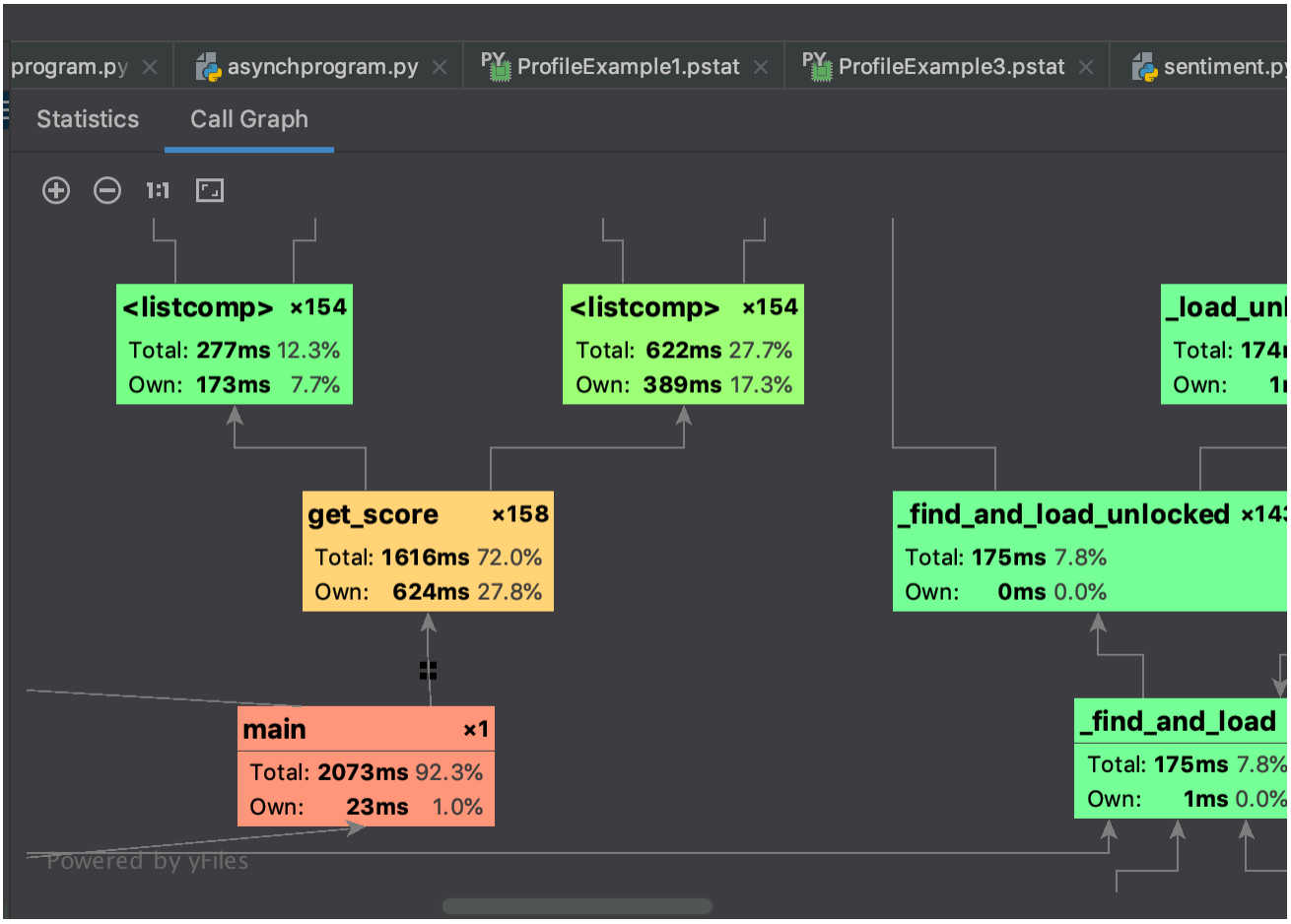


Figure 124: Figure illustrating the call graph information.

Note

If you have yappi installed, PyCharm will use that to profile. By default, PyCharm uses the cProfile library found in the standard library. Yappi has per-thread debug information, profiling decorators, and other features.

12.4 Addressing CPU Bound Code

The `get_score` function is where most of the time is spent. This is the low hanging fruit and the first place we should look to optimize. In that function, we loop over the `words` variable, but we also loop over the `stop`, `pos`, and `neg` lists. The `in` operator in Python works on lists by looping over them. So we have nested loops, which are prime candidates for looking for a better algorithm or data structure. We need to loop over the words in the corpus, but we don't need to repeatedly loop over the stop lists or positive and negative lists. By a simple refactoring, changing the list comprehensions to set comprehensions, we should be able to do a quick hash lookup on the sets instead of looping over lists.

Change `get_score` to the following and re-run the code:

```
def get_score(words):
    if not len(words):
        return 0.0
```

```
    neg = {line.lower().strip() for line in open('neg.txt')}
    pos = {line.lower().strip() for line in open('pos.txt')}
    stop = {line.lower().strip() for line in open('stop.txt')}

    pcount = 0
    ncount = 0
    remove = 0
    for word in words:
        if word in stop:
            remove += 1
            continue
        if word in pos:
            pcount += 1
        if word in neg:
            ncount += 1

    score = (pcount - ncount) / (len(words) - remove)
    return score
```

On my machine, the runtime goes from 2 seconds to 1.4 seconds. A 30% improvement, by changing 6 characters. That's a powerful insight gained from the tooling. But we can do better still.

Rerunning the profiling again, we see that the most time is now spent on the set comprehensions. In fact, the `neg`, `pos`, and `stop` sets are repeatedly created. When we find that we are doing the same thing multiple times, that is an indication that we should cache the results (and only calculate them one time).

Let's do one more refactor to both `main` and `get_score`. We will calculate the sets in the `main` function and then pass them into the `get_score` function:

```
def get_score(words, neg, pos, stop):
    if not len(words):
        return 0.0

    pcount = 0
    ncount = 0
    remove = 0
    for word in words:
        if word in stop:
            remove += 1
            continue
        if word in pos:
            pcount += 1
        if word in neg:
            ncount += 1

    score = (pcount - ncount) / (len(words) - remove)
    return score


def main():
    get_url(NEG_URL, 'neg.txt')
    get_url(POS_URL, 'pos.txt')
    get_url(STOPWORDS_URL, 'stop.txt')
    posts = get_posts('numpy')

    neg = {line.lower().strip() for line in open('neg.txt')}
```

```
pos = {line.lower().strip() for line in open('pos.txt')}
stop = {line.lower().strip() for line in open('stop.txt')}

for post in posts:
    words = get_line_words([post['description']])
    score = get_score(words, neg, pos, stop)
    print(f"{post['title'][:20]:20} Score: {score:6.3f}")
```

The code now takes .44 seconds, 78% better!

At this point, the statistics in the profile output show that the call taking the most time is the `recv_into` method of `_socket.so`. The table view by itself doesn't tell us who is calling this code. But you can right click and select "Show on Call Graph" to find out that the requests library is calling this code. It turns out that what is taking the most time now is waiting for the data to come back from the web server. The code is now IO-bound. We are waiting on an IO operation rather than waiting for the CPU to finish.

Typical methods for dealing with I/O-bound code include using multi-threading or asynchronous code to remove the wait on multiple calls. Caching common calls also remedies this, trading off storing more data for quicker access to it. (The `get_url` code is caching the calls to fetch the positive, negative, and stop word lists.) In this case, we are only making one HTTP call per URL, so caching will not help there.

12.5 Summary

This chapter took a simple application and optimized its runtime performance. We used the PyCharm profiling functionality to quickly view what code is taking the most time. Using the call graph, we were able to see where the code was called. By using the proper data structures and some caching we reduced runtime by 78%. Profiling is not unique to PyCharm, but its tight integration allows you to profile from the same application, quickly, without losing focus and having to juggle between tools. This is a huge benefit for developer productivity.

12.6 Commands

- *"Profile 'program'"* - Click on button with clock
- *"Show on Call Graph"* - Right-click on profile statistics

12.7 Exercises

1. If you want to try your hand at profiling, you can get the original version of the sentiment application covered in this chapter at https://github.com/talkpython/mastering-pycharm-course/tree/master/extra_samples/profiling/

   Try running the profiler on this project and see if you can identify the same issues as we have. See what other performance improvements you might make.

2. Open a recent project and execute it under the profiler. What is taking the most time?

Chapter 13

Unit testing

13.1 Introduction to Testing

Unit testing and testing, in general, aren't always the focus of software development. But they often distinguish amateur from professional developers. PyCharm has deep integration with unit testing and related functionality, like code coverage. In this chapter, we will explore how to leverage PyCharm and the pytest library to test code and ensure quality.

13.2 Logic to Test

In this chapter, we will create a function that takes a string and turns it into a list of numbers. You may have seen something like this in a printer dialog where you can specify that you want to print pages one through three and page 6. You could type in `1-3,6`. We want a function that would take this string and convert it into the list `[1,2,3,6]`.

Let's use Test Driven Development (TDD) to create this function, `parse_int`. The idea behind TDD is to start by writing a test for the functionality you want. The test should fail initially. Then you can implement the logic to let the test pass. Proponents of TDD claim that it focuses development on what you actually need and that you get testing as well. You don't need to use TDD to create code, but we will show an example of it in this chapter.

13.3 Writing a Test

Make a new project in PyCharm with a directory structure like this:

```
TestExample/
   numparser.py
   test_parser.py
```

Because we will be using the pytest library, we named our test file `test_parser.py`. The pytest library recognizes files that start will `test_` or end in `_test.py` as test files. We will follow that convention.

Open up `test_parser.py` and create a test. The easiest way to make a pytest test is to create a function that starts with `test` and use the `assert` statement in it. Here is a basic test:

```
from numparser import parse


def test_basic():
```

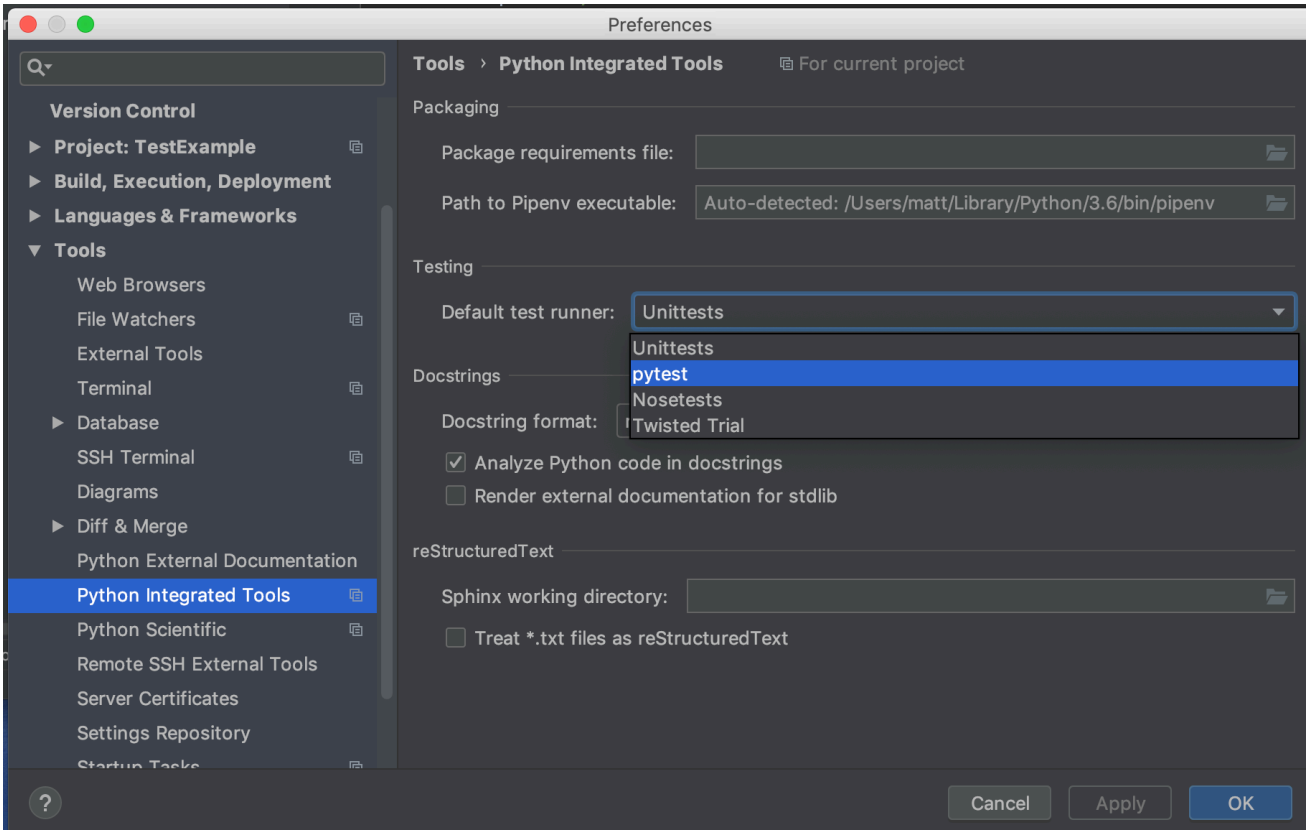


Figure 131: Figure illustrating changing the "Default test runner:" to pytest.

```
result = parse('1,3')
assert result == [1, 3]
```

The default test runner in PyCharm is the unittest library. Let's configure PyCharm to run this using pytest. Open *"Preferences"* and search for "Python Integrated Tools". From that window, you can change the "Testing" -> "Default test runner:" option to pytest.

With the default test runner set to pytest, you should see a green triangle appear in the gutter next to `test_basic`. If you click on that, you can run the tests (or profile them, or collect coverage information). Click on *"Run 'pytest for test_pars...'"* (ctrl-shift-F10) and you will see that a "Run" tab appears below with an error.

The pytest library is not in the standard library. My project does not have pytest installed, so I need to do that. You've got a couple options to install it. The "Python Integrated Tools" window has a warning at the bottom that says "No pytest runner found in selected interpreter". There is a "Fix" button that you can click to install it. Alternatively, you can add the line:

```
import pytest
```

into `test_parser.py` and use the code intention to fix the import.

At this point, we will get an `ImportError` because we haven't implemented `parse`. Let's do that.

Figure 132: Figure illustrating triangle next to test function that appears when pytest is set as the default test runner.

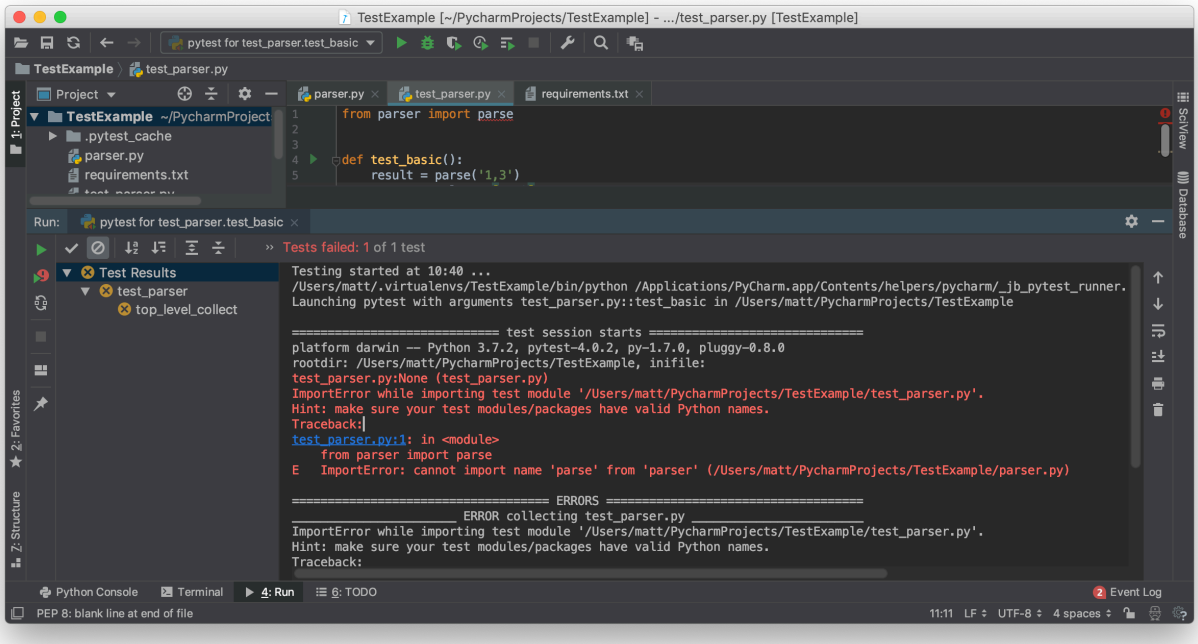


Figure 133: Figure illustrating TDD error. The code we are testing hasn't been implemented yet.

13.4 Implementing Logic

Because we are using test-driven development (TDD), the functionality of the `parse` function is being determined by our tests. We need to implement the ability to specify lists of integers separated by commas. Here is one possible implementation:

```
def parse(txt):
    result = []
    for num in txt.split(','):
        result.append(int(num))
    return result
```

If you re-run the test you should see output like this:

```
==================== test session starts ============================
platform darwin -- Python 3.7.2, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/matt/PycharmProjects/TestExample, inifile:
collected 1 item

test_parser.py .                                                   [100%]

=================== 1 passed in 0.02 seconds ========================
Process finished with exit code 0
```

This output tells you what operating system you are on (darwin is for MacOS), and what version of Python and pytest libraries you are using. (The py and pluggy libraries are dependencies of pytest). The "rootdir" is the directory that pytest uses to search for test files in. The "inifile" is a `pytest.ini` (or `setup.cfg` or `tox.ini` file) used to customize pytest behavior. In this case, there was no inifile. The next line reports that pytest found 1 test. There was a test in `test_parser.py` and it was 100% of the tests. If you look closer at that line, you will see a period. That indicates that the test in `test_parser.py` passed. The pytest library has other statuses for tests, among them, are s (skipped), F (fail), and E (exception).

13.5 Examining Failures

The pytest library tries to make it really easy to debug failures. Let's make the test fail and look at the output. Change the test code to:

```
def test_basic():
    result = parse('1,3')
    assert result == []
```

The output of pytest now looks like this:

```
==================== test session starts ============================
platform darwin -- Python 3.7.2, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/matt/PycharmProjects/TestExample, inifile:
collected 1 item

test_parser.py F
test_parser.py:3 (test_basic)
[1, 3] != []

Expected :[]
Actual   :[1, 3]
 <Click to see difference>
```

```
def test_basic():
        result = parse('1,3')
>       assert result == []
E       assert [1, 3] == []
E         Left contains more items, first extra item: 1
E         Use -v to get the full diff

test_parser.py:6: AssertionError
                                                              [100%]

========================== FAILURES ==================================
_________________________ test_basic _________________________________

    def test_basic():
        result = parse('1,3')
>       assert result == []
E       assert [1, 3] == []
E         Left contains more items, first extra item: 1
E         Use -v to get the full diff

test_parser.py:6: AssertionError
=================== 1 failed in 0.10 seconds ==========================
Process finished with exit code 0
```

Rather than only notifying us that the assertion failed, pytest does some nifty tricks to show you why the assertion failed. There is also integration in PyCharm to pull up the "Comparison Failure" window if you click on `<Click to see the difference>`.

Along the left side of the "Run" tab at the bottom of PyCharm are buttons for *"Rerun pytest"* (ctrl-F5), *"Rerun Failed Tests"*, and *"Toggle auto-test"*. Rerunning failed tests is a great way to quickly fix failing tests without the distraction of passing tests. Auto-test is a mode for running the tests anytime the code changes. There is a slight delay after changing the code, then the tests automatically run.

Note

There are many features that pytest has, that PyCharm doesn't expose, but you can access them from the "Terminal" window. For example running:

```
pytest -x --lf
```

will run the last test that failed first. And it will stop after the first failure. This is a great invocation to focus on fixing one failed test case at a time. When you move code to a different platform or refactor code, this is really handy.

Once you have a pytest run configuration in PyCharm, you can open the *"Edit configurations"* in the top right where you normally select which configuration will run and enter -x –lf in the *"Additional Arguments"* to have PyCharm run with this option.

13.6 Adding More Tests

Let's add another feature to the `parse` function, the ability to accept dashes for ranges. First, we will create a test in `test_parser.py`:

```
def test_dash():
    result = parse('1-3')
    assert result == [1, 2, 3]
```

Be careful, if you clicked the triangle in the gutter to run `test_basic` previously, clicking on the green triangle on either the Run tab or the triangle on the top (the *"Run"* command), PyCharm will only run the `test_basic` test and not the new test. In the name of the run configuration you will see "pytest for test_parser.test_basic", indicating that the run configuration is only for the single test.

You can use the *"Edit Configurations..."* command to change the "Target:" from `test_parser.test_basic` to `test_parser` and then the *"Run"* (shift-F10) command will run all the tests in the file.

Note

In real-world projects we could have multiple test files. If would be nice if PyCharm could run all of the tests with a single click.

If you change the "Target:" to a "Script path" and set it to "." (a period without quotes), PyCharm will run all of the test files found in the working directory.

The output from running all of the tests should look like this:

```
Testing started at 14:12 ...
~/.venvs/bin/python /Applications/PyCharm.app/_jb_pytest_runner.py
--path test_parser.py
Launching pytest with arguments test_parser.py in
/Users/matt/PycharmProjects/TestExample

================== test session starts ==============================
platform darwin -- Python 3.7.2, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/matt/PycharmProjects/TestExample, inifile:
collected 2 items

test_parser.py .F
test_parser.py:7 (test_dash)
def test_dash( ):
>       result = parse('1-3')

test_parser.py:9:
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

txt = '1-3'

    def parse(txt):
        result = []
        for num in txt.split(','):

>           result.append(int(num))
E           ValueError: invalid literal for int() with base 10: '1-3'

numparser.py:15: ValueError
                                                          [100%]

====================== FAILURES ====================================
_____________________ test_dash ____________________________________
```

```
    def test_dash( ):
>       result = parse('1-3')

test_parser.py:9:
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

txt = '1-3'

    def parse(txt):
        result = []
        for num in txt.split(','):

>           result.append(int(num))
E           ValueError: invalid literal for int() with base 10: '1-3'

numparser.py:15: ValueError
========== 1 failed, 1 passed in 0.15 seconds ======================
Process finished with exit code 0
```

Hidden in there is the line:

```
test_parser.py .F
```

which contains a period and an "F". The period is for the passing test and the "F" is for the failing test, which is raising a `ValueError`.

Let's update the implementation in `numparser.py` to support dashes:

```
def parse(txt):
    result = []
    for num in txt.split(','):
        if '-' in num:
            start, end = num.split('-')
            result.extend(range(int(start), int(end)+1))
        else:
            result.append(int(num))
    return result
```

The tests should now run fine.

13.7 Testing Exceptions

We also want to be able to test exceptions. The `parse` function doesn't explicitly raise an exception, but if we pass in bad data into it the function should raise an exception. We can use pytest to assert that an exception is raised. Let's figure out what exception will be raised.

The easiest way to see what exception is raised is to run the code with bad input. I prefer to do that from the interpreter. If you right-click on the source code (in the editor, not on the filename in the Project tab), there is a command, *"Run File in Console"*. If you click that, the "Python Console" will open with the contents of the files loaded.

I typed the following in the console to see the error:

```
>>> parse('bad')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "TestExample/numparser.py", line 8, in parse
    result.append(int(num))
ValueError: invalid literal for int() with base 10: 'bad'
```

It looks like it will raise a `ValueError` if a string is passed in. Let's write a test to handle that.

Until now, nowhere in our `test_parser.py` file was there a reference to `pytest`. When we invoke the tests with pytest, pytest is able to determine what is a test and how to handle `assert` statements. However, the `assert` statement can't handle exceptions. For that, we need to use a special context manager, `raises`, found in `pytest`. Update the `test_parser.py` file to:

```
from numparser import parse


import pytest

def test_basic():
    result = parse('1,3')
    assert result == [1, 3]

def test_dash():
    result = parse('1-3')
    assert result == [1, 2, 3]

def test_bad():
    with pytest.raises(ValueError):
        parse('bad')
```

We can use the `with` statement along with the `pytest.raises` context manager to test that an exception occurred in the block of the context manager.

The pytest library has many other features that could fit in a book of their own. We encourage you to explore the library further and take advantage of it for testing in Python. In addition to the many features it ships with, it has a rich ecosystem of plugins to further enhance it.

13.8 Code Coverage

Code coverage is a report of how many lines of code were executed. It usually goes hand in hand with testing. Because Python isn't compiled like Java or C we don't have the sense of security that compiling our code might give us. We can get that feeling by knowing that we have tested our code. After all, Python code's syntax isn't validated until it executes in many cases. Lines that have been tested necessarily have valid syntax. And it is code coverage that will tell us which lines of code we have executed when running the tests.

PyCharm makes it easy to run code coverage. We need to click the *"Run 'pytest for test_parser' with Coverage"* command. The icon has a shield behind the green run button. Also, once you have a run configuration for a given set of tests, the run with code coverage (same icon) will be near the run and debug buttons as well.

Note

You will need to install a library for coverage. When you the *"Run 'pytest for test_parser' with Coverage"* command, you will have the option to install the `coverage.py` library.

When PyCharm is properly configured for running coverage, a "Coverage" tab will appear after running the coverage command. This tab will show summary statistics for code coverage. Typically, we are most concerned with the coverage numbers for non-test code. Another nice integration with PyCharm is that the coverage results are shown in the gutter. There is a green or red color in the gutter next to the line to indicate if there was code coverage.

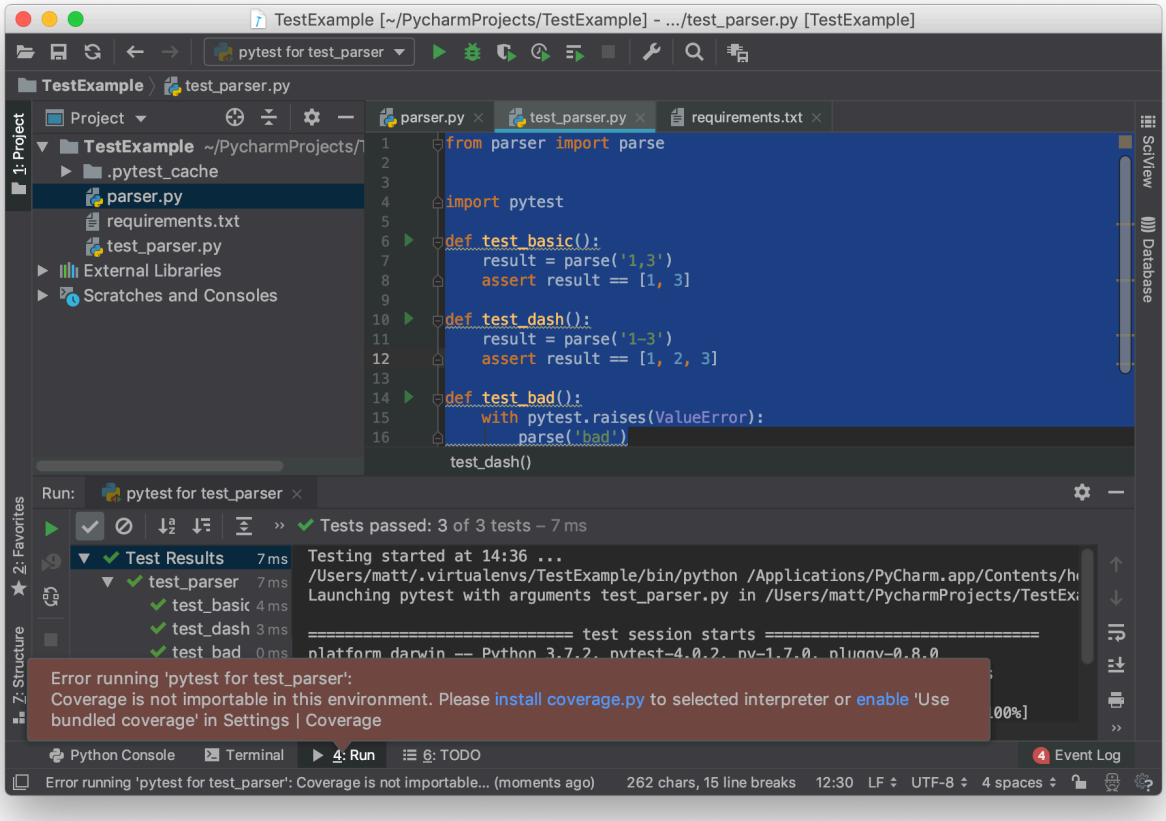


Figure 134: Figure illustrating missing coverage dependency.

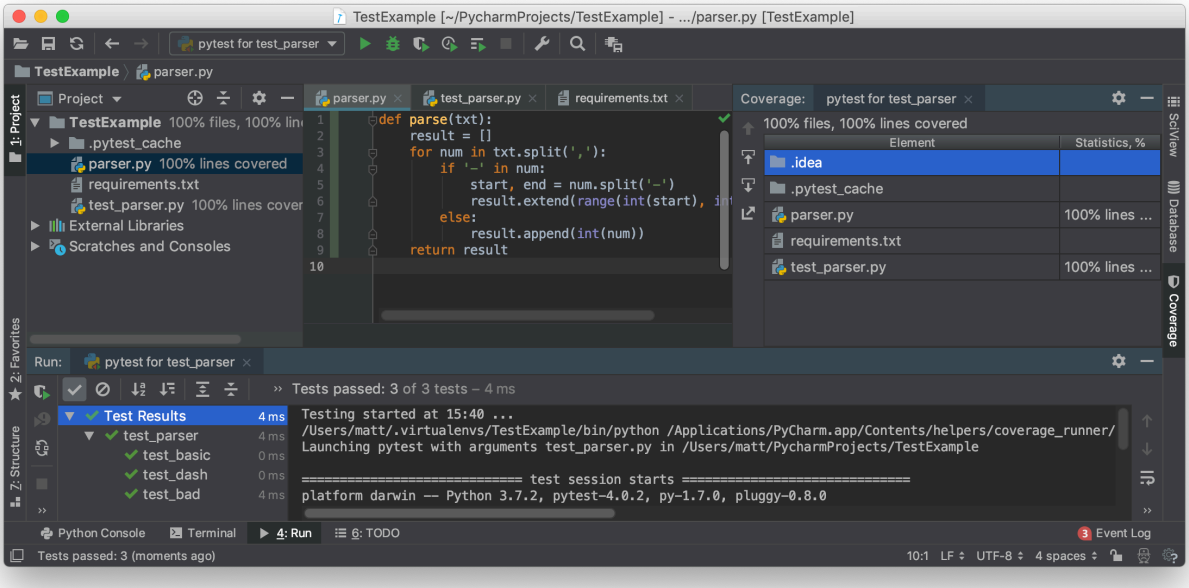


Figure 135: Figure illustrating coverage summary.

For our `parse` function, we have 100% coverage. A few words on coverage numbers. Higher numbers are better than lower numbers. But, 100% coverage only means that the lines were executed, it does not mean that the code is free from logic errors. Also, most code coverage tools only deal with line coverage. As programs are executing, there are "branches" and "paths" through the code that can change depending on the input to the code.

A more exhaustive type of testing and coverage is full path coverage, testing different possible paths through the code. In practice, this type of coverage is difficult because most tools do not offer support for it, but it also requires much dedication to understanding the code and the combinations of execution paths. Most companies that have metrics around coverage pick a number they would like to hit and strive not to have new commits to the code lower that number.

13.9 Summary

In this chapter, we used test driven development to implement a function to parse strings into lists of numbers. We used to pytest library because it makes test creation very easy. We barely scratched the surface of the functionality of this library. Finally, we used the integration in PyCharm to look at code coverage statistics. Having access to these libraries in an integrated environment is a huge boon to Python developers.

13.10 Commands

- *"Run 'pytest for test_pars...'"* - (ctrl-shift-F10)
- *"Rerun pytest"* - (ctrl-F5)
- *"Rerun Failed Tests"*
- *"Toggle auto-test"*
- *"Run File in Console"*
- *"Run 'pytest for test_parser' with Coverage"*

13.11 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/10-testing` to use pytest on an existing project.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/10-testing` to use coverage.

Chapter 14

Data science tools

14.1 Introduction to Scientific View

Python is commonly used for numeric manipulation. Libraries like numpy, pandas, and matplotlib have provided a foundation such that now Python is one of the most desired skills for data science and machine learning. Using Python for analytics requires a workflow that is a little different from application development. Rather than building tools and deploying them, it is usually more interactive and more iterative. You filter, sort, and tweak some data, then look at it. PyCharm has a "Scientific Mode" to aid with this style of usage. It also has Jupyter integration.

14.2 Scientific Mode

When creating a PyCharm project, one of the options alongside a "Pure Python" project is a "Scientific" project. This will turn on "Scientific Mode", which has views for looking at plots and inspecting data. Another option for enabling scientific mode is including the following line in your code:

```
import numpy as np
```

When you include an import to the `numpy` library, PyCharm will sense that and ask if you want to turn on scientific mode. For the next example, we will pull fuel economy data and plot the number of makes a car vendor had for each year. Given the right data, the pandas library can make quick work of this. Create a new project with the following layout:

```
Data/
  explore.py
```

Place the following code in `explore.py`:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

#%%
auto = pd.read_csv(
  'https://www.fueleconomy.gov/feg/epadata/vehicles.csv.zip')

#%%
# Just look at Ford, Lexus, & Toyota
(auto
   .groupby(['year', 'make'])
```

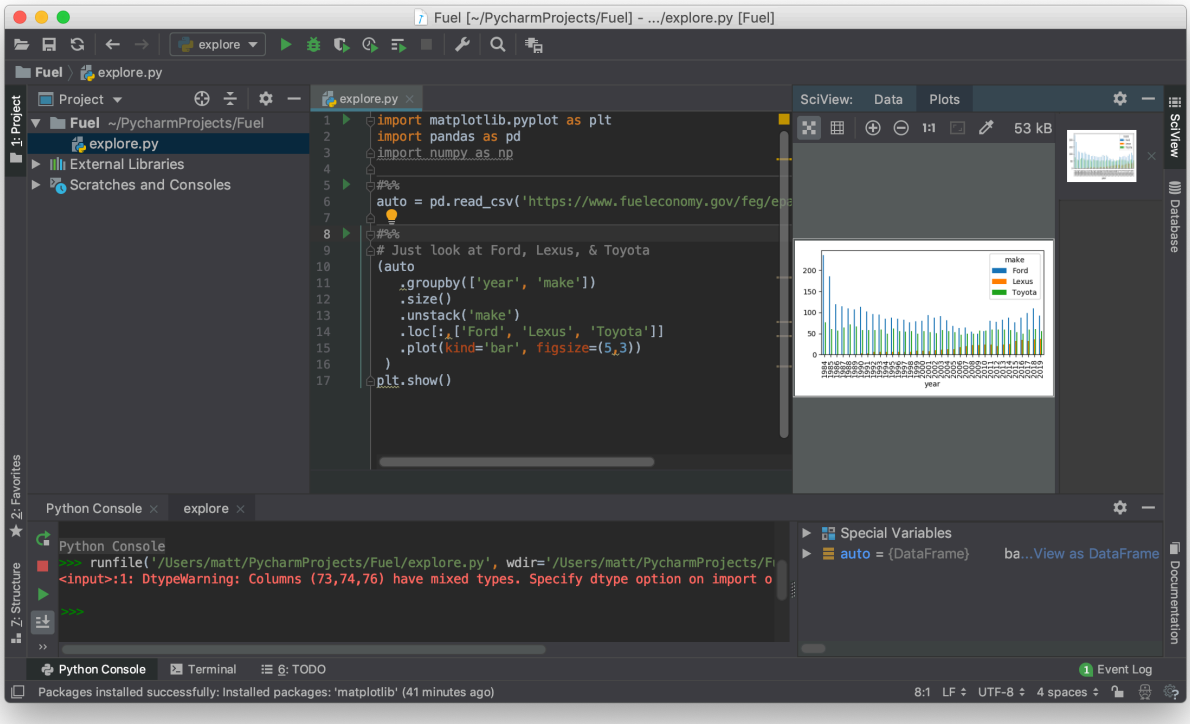


Figure 141: Figure illustrating running code with a plot in scientific mode.

```
    .size()
    .unstack('make')
    .loc[:,['Ford', 'Lexus', 'Toyota']]
    .plot(kind='bar', figsize=(14,10))
  )
plt.show()
```

Note

The parentheses around the pandas code look a little weird to most Python developers. This is common in pandas because we tend to chain operations. With the parentheses, we don't need to have all of the code on a single line. We put each operation in the chain on its own line, and it makes it easier to understand what is happening.

This code will group the data by year and make, then it will calculate the size of each group, then it will pull the make column (really an index) into a row (giving us every make for each year), then it pulls off only Ford, Lexus, and Toyota, and finally plots a bar plot.

If you were to type in this code, PyCharm would ask if you wanted to go into "Scientific Mode" when it sees the `numpy` line. My code imports matplotlib, but doesn't call it. I did that so I could use the code intention to easily install it. (Alternatively, you could open the terminal and `pip install` it). You also need to install pandas.

If you run this code, it might think for a bit, but in the "SciView", there will be a "Plots" tab with the resultant plot.

You will also notice that in the "Python Console" window, there is an "explore" tab with an interpreter. That interpreter has the namespace of `explore.py`. From there we can access the `auto` dataframe, and interact with the data. If you have installed iPython (`pip install ipython` from the "Terminal"), you will get an iPython prompt instead of a normal Python prompt.

On the right side is a list of the variables in the namespace. In this case, only `auto` is there. There is a clickable area titled "View as DataFrame". When you click it, the SciView tab for "Data" will be populated with a table of the data.

One more thing to note. Because our code contained comments starting with `#%%`, PyCharm recognizes this a "cell" of code. The cell consists of the code starting following that comment until the next `#%%` comment. It puts a green execute triangle next to each cell and we can run, or re-run those cells independently. PyCharm conveniently creates a cell for the initial code before the first `#%%`.

Using Python with the interpreter and access to previous variables is great for interactive exploration.

14.3 Notebook Mode

Another feature PyCharm has for scientific computing is integration with Jupyter. Jupyter is an open source tool for making notebooks for running "cells" of code and displaying markup. This tool was used to publish the discovery of gravitational waves and in 2018 the winner of the Nobel prize used Jupyter to work on and publish their efforts. Let's explore the features found in PyCharm.

We can use the *"New"* command (ctrl-n) to create a new "Jupyter Notebook". Let's migrate our previous auto exploration code to a notebook by using the New command and naming the notebook "Auto Exploration". If Jupyter is not installed, it will bring up a dialog to install it. You can click on "Install jupyter package".

A notebook editor in PyCharm 2019.1 shows an editor and a notebook-like pane next to it. PyCharm will insert the code `#%%` into the editor. As you type into the edit, the content appears in the notebook pane as well. Let's add some code to the editor. I like to have the top cell in a notebook only contain the imports for the code. Add the matplotlib and pandas imports to it below the `#%%` line:

```
import matplotlib.pyplot as plt
import pandas as pd
```

If you have created a new virtual environment for this project, you will need to install these libraries. Click on the library name (with the redline under it) and you will get a code intention for installing the libraries. If you click the green triangle, to run *"Execute Code Cell"* (ctrl-enter), PyCharm should start Jupyter and run the code.

After executing the cell. The notebook view will show a number next to a cell, and show any output that was generated. Because this cell has no output, it will say "No output".

To create a new cell, add the comment above where you want the cell to start. Alternatively, you can click on the plus in the gutter to create a code cell or a markdown cell.

Add the following code to the notebook:

```
#%%
auto = pd.read_csv(
  'https://www.fueleconomy.gov/feg/epadata/vehicles.csv.zip')

#%%
```

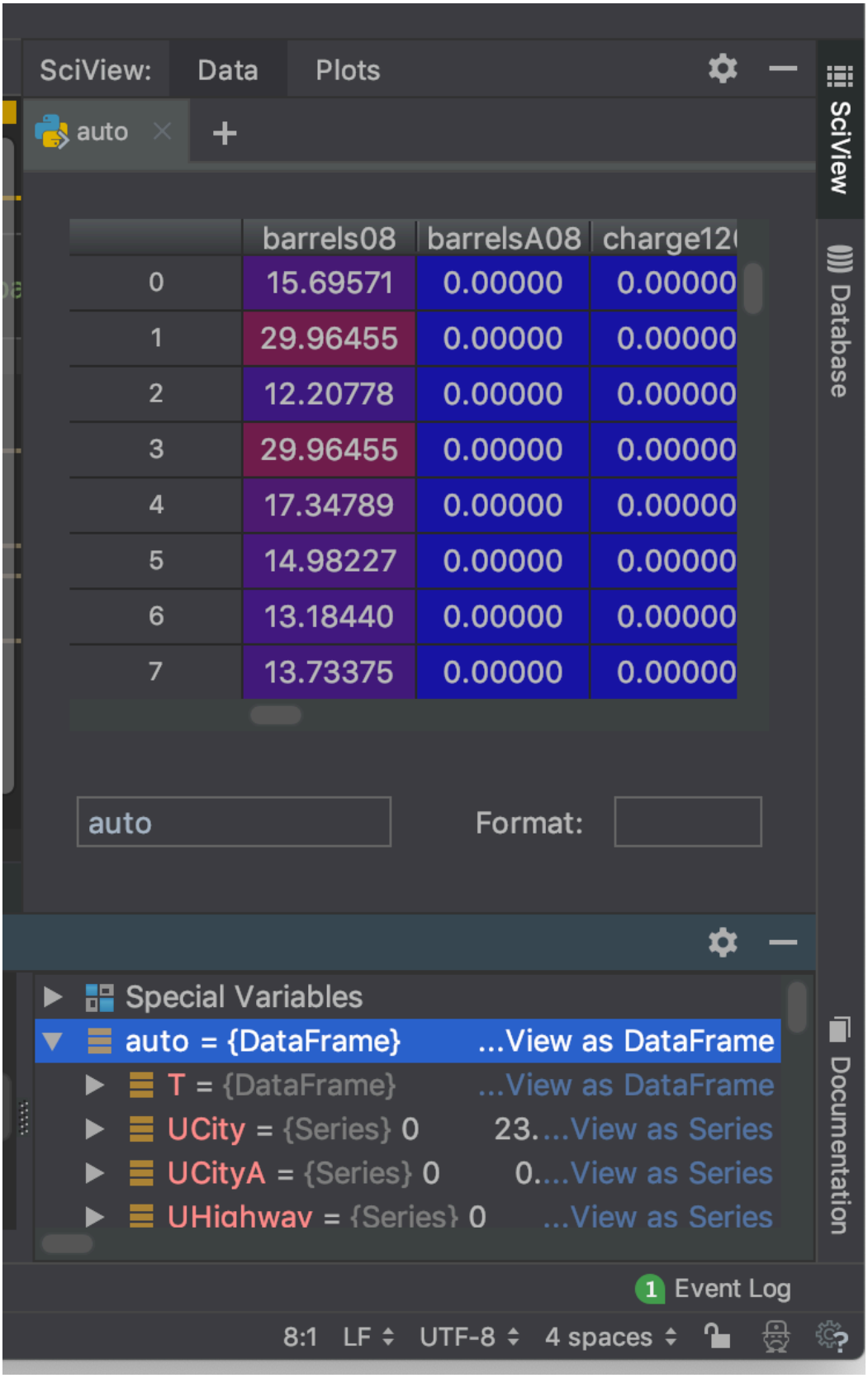


Figure 142: Figure illustrating result of clicking ”View as DataFrame”.

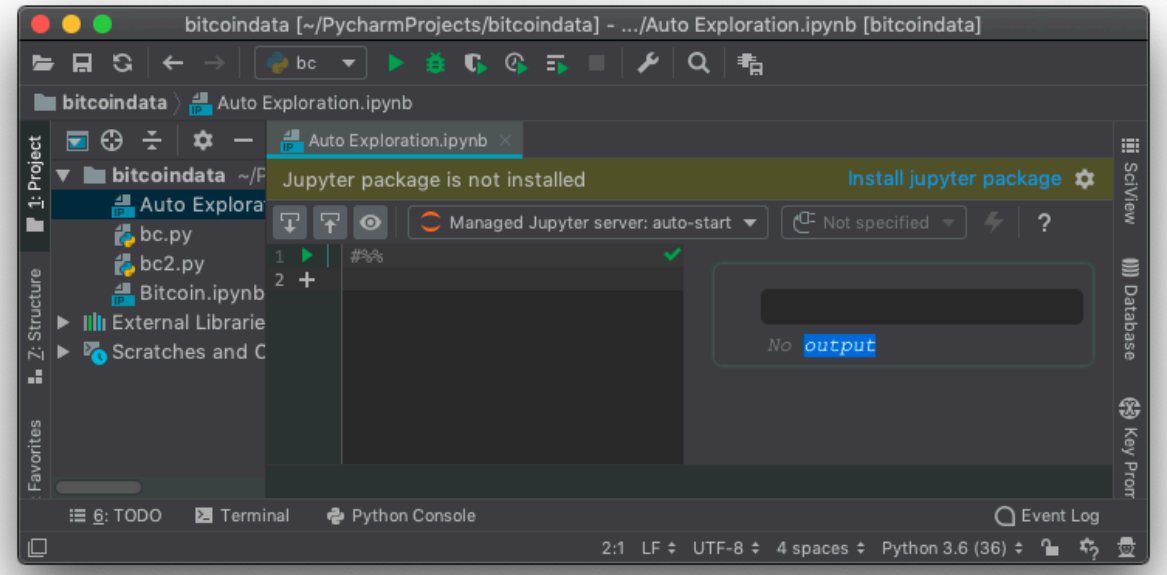


Figure 143: Figure illustrating a new Jupyter notebook, and prompt to install the ”jupyter” package..

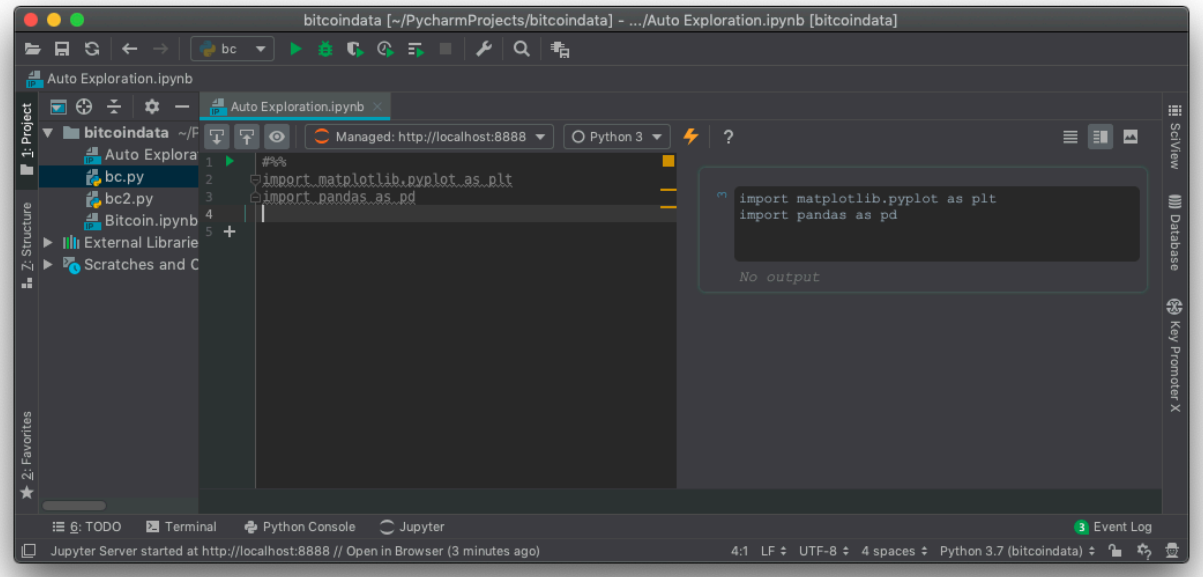


Figure 144: Figure illustrating running a cell when starting a new Jupyter notebook.

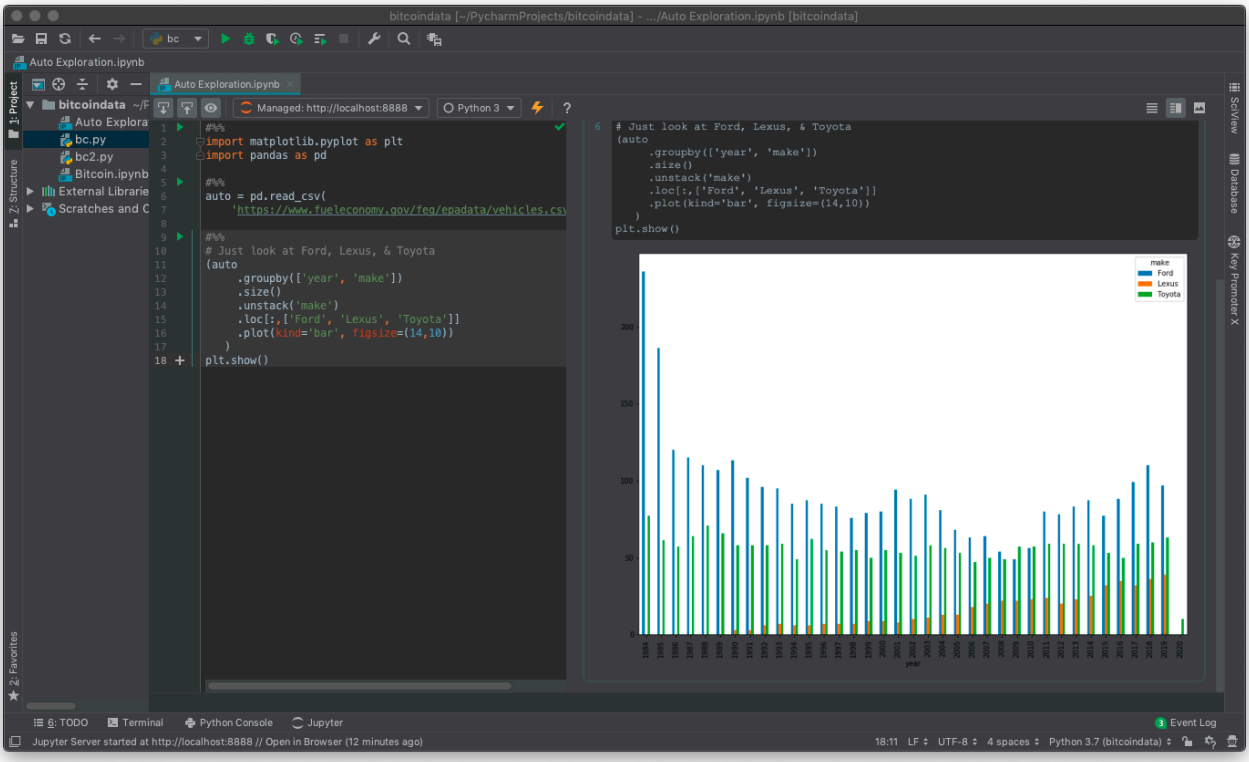


Figure 145: Figure illustrating plot of notebook cell.

```
# Just look at Ford, Lexus, & Toyota
(auto
   .groupby(['year', 'make'])
   .size()
   .unstack('make')
   .loc[:,['Ford', 'Lexus', 'Toyota']]
   .plot(kind='bar', figsize=(14,10))
 )
plt.show()
```

If you run these cells, you will see that PyCharm embeds the plot in the notebook area.

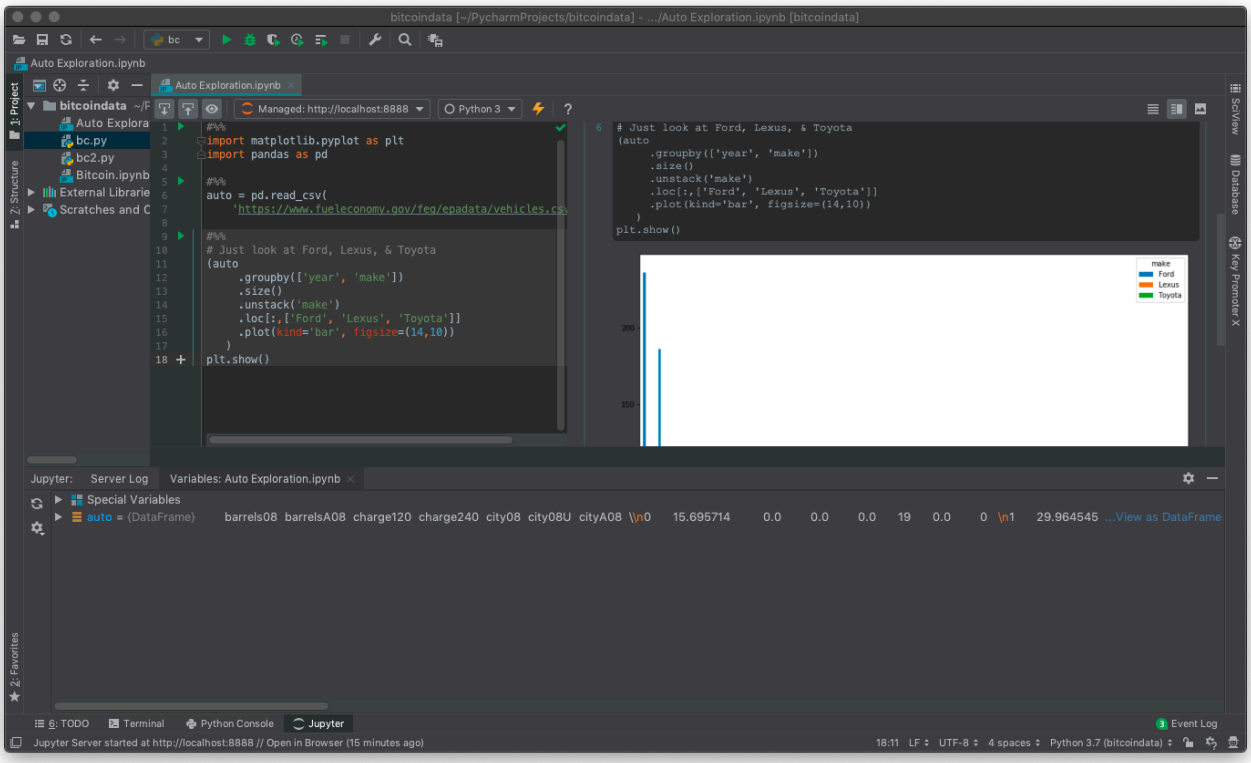


Figure 146: Figure illustrating showing the Jupyter tab of PyCharm.

Note

If you were planning on also using this notebook with the Jupyter browser interface, you should all the *cell magic* for displaying plots in the browser. Cell magics are directives to control how Jupyter behaves. I would add the following line to cell containing the imports:

```
%matplotlib inline
```

Additionally, PyCharm provides a "Jupyter" tab that displays the variables from the notebook. You can click that tab to explore the variables. In this case it only shows the `auto` variable.

14.4 Summary

This chapter introduced Scientific Mode and support for Jupyter notebooks. When used for analysis, it is common to use the REPL and interactive nature to try things out and quickly examine the data. This style of programming is quite different from application development, where we try to avoid using global variables and tend to write functions and classes. This isn't to say that normal Python developers don't use the REPL (they should definitely take advantage of it), or espouse bad programming practices like using global variables. When I'm doing some data science consulting for clients, I use Jupyter to explore and create models. After I'm satisfied with the results, I will refactor the code and apply engineering practices to introduce functions, classes, testing, and packaging to the analysis. PyCharm has you covered here for the full range of development styles.

14.5 Commands

- `import numpy` - Include this is prompt for Scientific Mode
- `#%%` - Include this to divide cells
- *"Execute Code Cell"* - (ctrl-enter or click green button)

14.6 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/11-scientific-computing` to create a plot in scientific mode.
2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/11-scientific-computing` to create a plot using Jupyter.

Chapter 15

Tool Windows

15.1 Introduction to Tool Windows

In this chapter, we will look at a grab bag of tools. These tools and utilities are helpful for various tasks. We will see Todo comments, the Run Tool Window, the Python Console, the Terminal, Favorites, and the Structure Window. These are tools for interacting with your code and environment that don't clearly fall into the categories we've already covered.

15.2 Todo Comments

Sometimes you want to leave notes to yourself regarding your code. Reminding the future you to fix an issue, add a feature, or refactor some lines. You can do this by filing a bug, adding a comment, changing the README file, or jotting it down in a notebook. PyCharm has a "TODO Window" to support this use-case. If you use the word "FIXME" or "TODO" in a docstring or comment, PyCharm will track those, and highlight them in a different color.

Inside of the scroll bar is a clickable line for quick navigation to these items from the editor window. In addition, the is a *"TODO"* (command-6 or alt-6) command that toggles a window showing all the TODO items in a project. You can expand the folders and files the see the items, and click on them to jump to that TODO item in the editor. Along the side of the TODO windows are buttons for *"Previous TODO"* (alt-command-up or ctrl-alt-up), *"Next TODO"* (alt-command-down or ctrl-alt-down), *"Filter TODO Items"*, *"Autoscroll to Source"*, *"View Options"*, and *"Preview Source"*.

By default PyCharm highlights "todo" and "fixme" ignoring case. If you open the settings and search for "TODO", you can add alternative patterns for PyCharm to recognize. If you add a new pattern, you can use the *"Filter TODO Items"* to specify which patterns to highlight.

Note

This is not a substitute for using a proper bug tracker.

15.3 The Python Console Window

The *"Python Console"* Window (Tools -> "Python Console..." or click on "Python Console" tab) is an invaluable tool for Python development. If you run the command, it will start a Python REPL inside of your virtual environment. It will also add the project directory to the `sys.path` attribute.

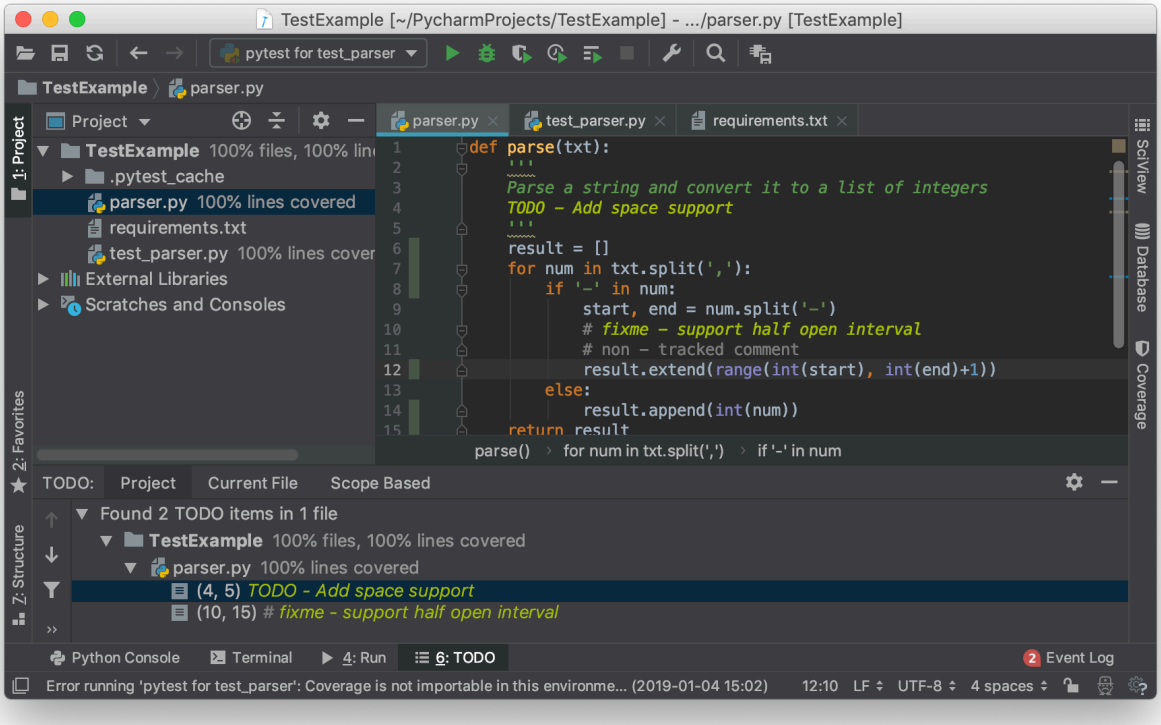


Figure 151: Figure illustrating TODO docstring and fixme comment.

This makes it easy to import source code found in your project and use the dependencies in the virtual environment.

Although this looks similar to the Python REPL you see when you call `python3` from the command line, it has a few tricks up its sleeves. It comes with the code completion included in the Python editor. Simply hit tab after starting to type a variable name or period. You can also pull up *"Basic Code Completion"* (ctrl-space).

You can navigate through the history of commands using the up and down arrow. If you had a typo when typing out a function in the Python Console, you can hit the up arrow, and fix the typo rather than typing out the whole function again.

The tab window also has a variable inspection area. Local variables are displayed there. Imported modules can be found under the "Special Variables" section. The variables can be inspected there. Nested variables such as lists and dictionaries can be expanded to view parts of the variable. In addition, you can run the *"Set Value..."* (f2) command to change the value of a variable or a nested value.

The Python Console also appears when you run a "Run Configuration" (there is an option in the Python configuration for "Run with Python console"). By using a Run Configuration, you will have access to all of the global variables in a file without having to import them.

If you want to run a specific portion of code, you can type it in directly in the Python Console. You can also highlight code in the editor and run *"Execute Selection in Console..."* (alt-shift-e). If you execute the definition of a function by executing a selection, you won't see the function name in the variable inspection area unless you expand the "Special Variables" hierarchy.

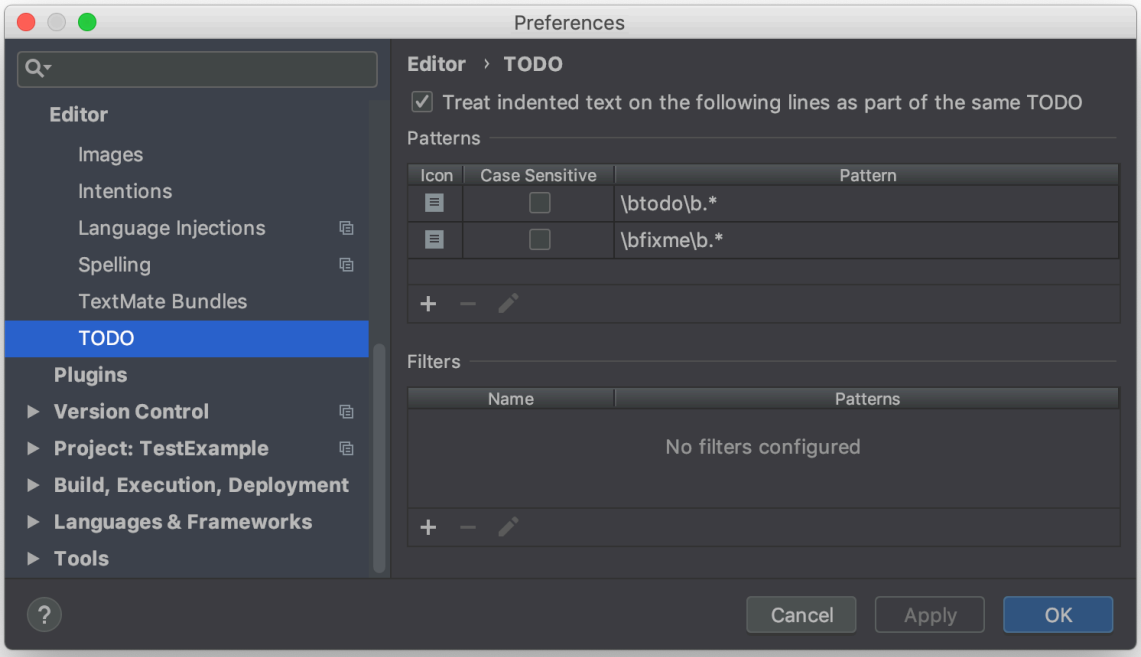


Figure 152: Figure illustrating TODO configuration.

A nice feature of the Python console is that it stays open after execution. This gives you the ability to inspect variables and interact with the global namespace even when the code has finished executing.

15.4 The Run Tool Window

The *"Run"* Window (command-4 or alt-4) toggles the invocation for the last run. This window appears when tests are configured to execute with a test runner. If you use PyCharm to launch a "Jupyter Notebook" configuration, it will use this window as well. The web run configurations (Pyramid, Flask, or Django) also use the Run window. It can also be used for Python execution (by unchecking "Run with Python Console"), though I don't recommend this.

Notable features of the Run window include the *"Rerun"* command (ctrl-f5), *"Stop"* (command-f2 or ctrl-f2), and *"Pause Output"*. The latter is useful if you have a lot of output generated during a server process and you want to stop and examine it. There are buttons along the left-hand side for these commands.

15.5 The Terminal Window

PyCharm includes a *"Terminal"* (alt-f12) that is integrated. On Unix platforms, it launches a bash shell in the project directory. On Windows, it will launch a command prompt (`cmd.exe`). It also activates the virtual environment for the project. This makes package installation really easy and you can use the `pip` command in the Terminal.

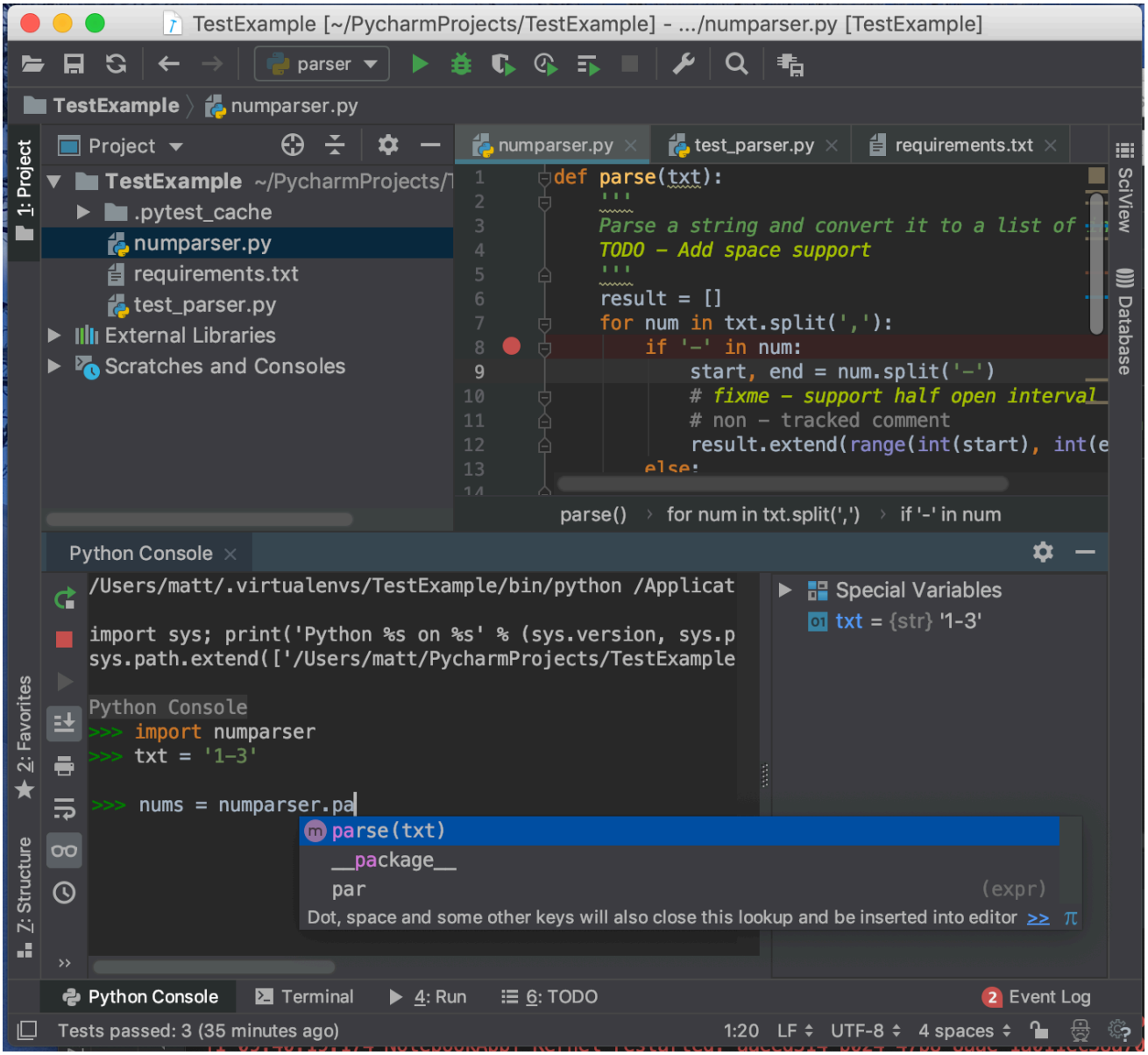


Figure 153: Figure illustrating Python console with code completion and variable inspection.

To view the packages that are installed, you can type:

```
pip list
```

on Windows or Unix.

If you prefer other shells, you can go to the "Terminal" section of the Settings window. Under the "Application settings" area is a "Shell path" entry. You can specify `powershell` on Windows, or `bin/zsh` for Zsh.

If you want another Terminal, click on the *"New Session"* (the plus button) command. This is useful if you have a long-running process going in a terminal and want to perform another action while it is running. A few other tricks include the ability to paste using the middle mouse button (I wish this worked in editor windows), and cycling through previous commands with the up and down arrows.

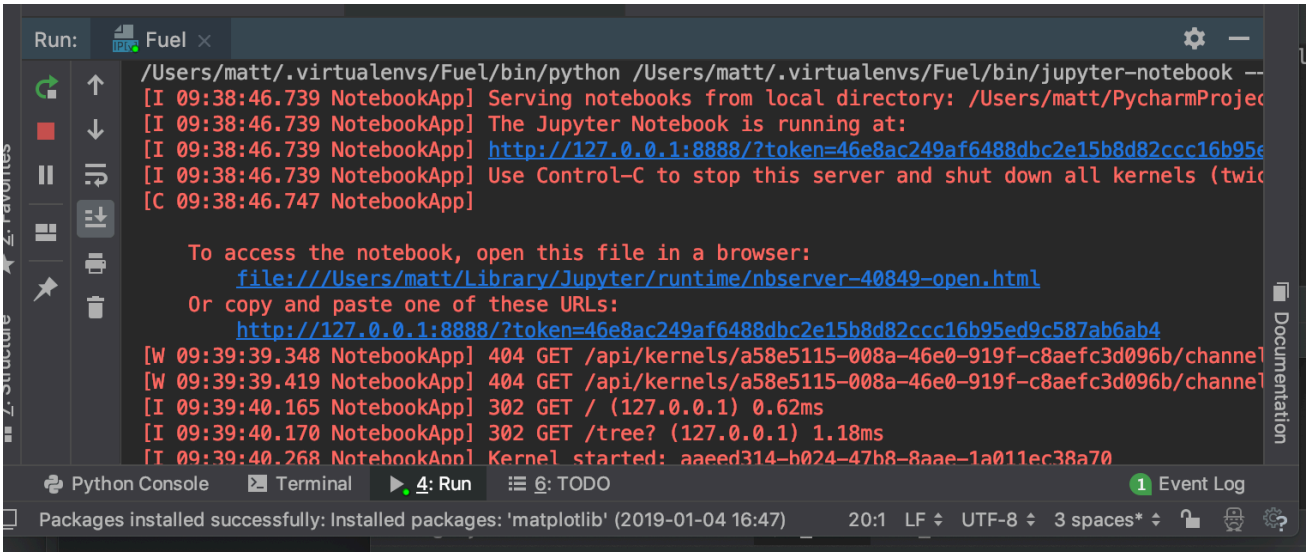


Figure 154: Figure illustrating the Run Tool Window.

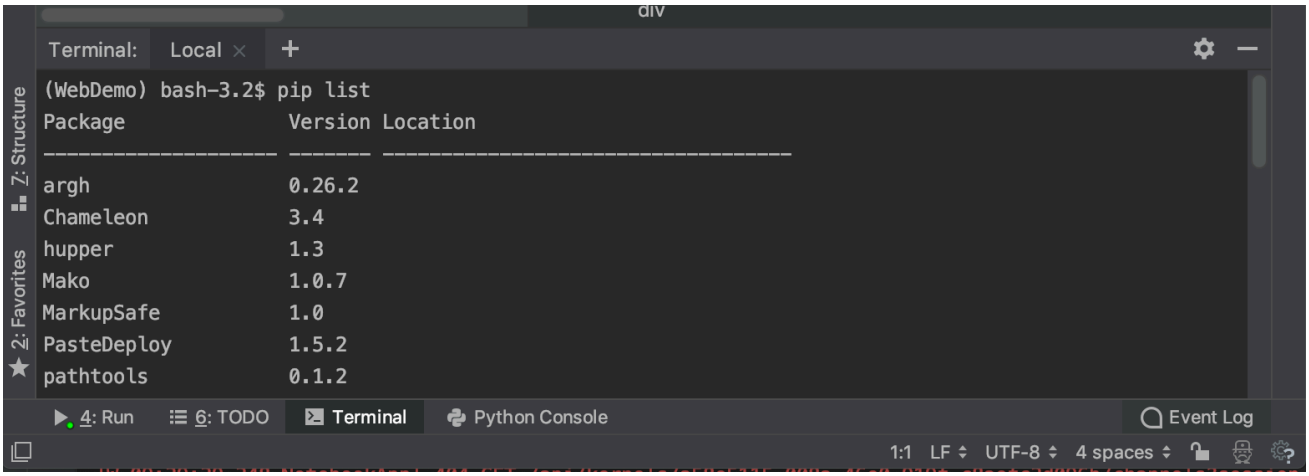


Figure 155: Figure illustrating Terminal running a command in the virtual environment.

The Terminal is tightly integrated for running various command line tools. It works very well for package management and running shell tools (such as Django's `manage.py` script) on all platforms. Minimizing task switching by limiting changing applications is helpful for maintaining focus.

15.6 The Favorites Tool

The *"Favorites"* (command-2 or alt-2) command toggles a window that shows contextual reminders. You can store favorites, bookmarks, and see breakpoints in the window.

A favorite is a file that you want to keep track of. You can use the "Project" window to find files, but if you have a deeply nested project structure with many files, this can be unwieldy. This is also handy when you are working on a bug that affects a few files in different directories.

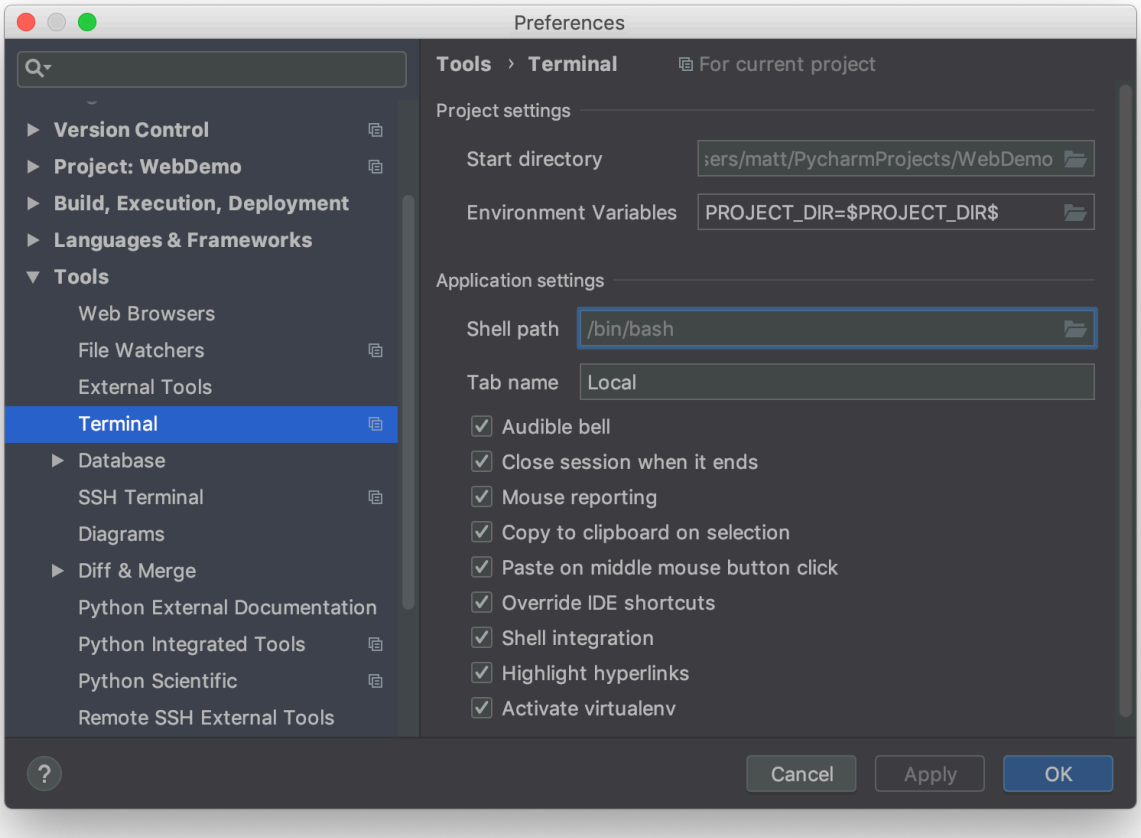


Figure 156: Figure illustrating Terminal settings in the Preferences Window.

In web applications, you might have to edit model and view code in Python, as well as JavaScript and CSS when adding a feature or fixing a bug. You can create a new favorite list and add all of the files to it.

To add a file to favorites, use the *"Add to Favorites"* command (alt-shift-f, File -> "Add To Favorites", right-click on a file in the Project view, or right-click on the tab in the editor). If you want to add all of the open files to a favorite, you can right-click on one of the tabs in the file editor and click *"Add All To Favorites"*.

A bookmark is like a favorite, but more fine-grained. Rather than referring to a file, a bookmark refers to a line in a file. You can *"Toggle Bookmark"* (f11 or Navigate -> "Bookmarks" -> "Toggle Bookmark") when on a line of code.

In addition, you can create a "mnemonic" bookmark. With the *"Toggle Bookmark with Mnemonic"* (command-f11 or ctrl-f11), PyCharm will bring up a dialog to select a cell. The cells are single character alphanumerics. If you select "1" as the cell, there will be a new keyboard *"Go to Bookmark <1>"* shortcut (ctrl-1) to navigate to the mnemonic bookmark. You can also use the "Favorites" window to navigate to bookmarks.

Bookmarks are very useful for tracking particular lines in files. Take advantage of mnemonic bookmarks for quick navigation to commonly used lines of code.

Finally, the Favorites window allows easy viewing and navigation to breakpoints. When you run the *"Toggle Breakpoint"* (command-f8 or ctrl-f8) or click in the gutter next to the code, the

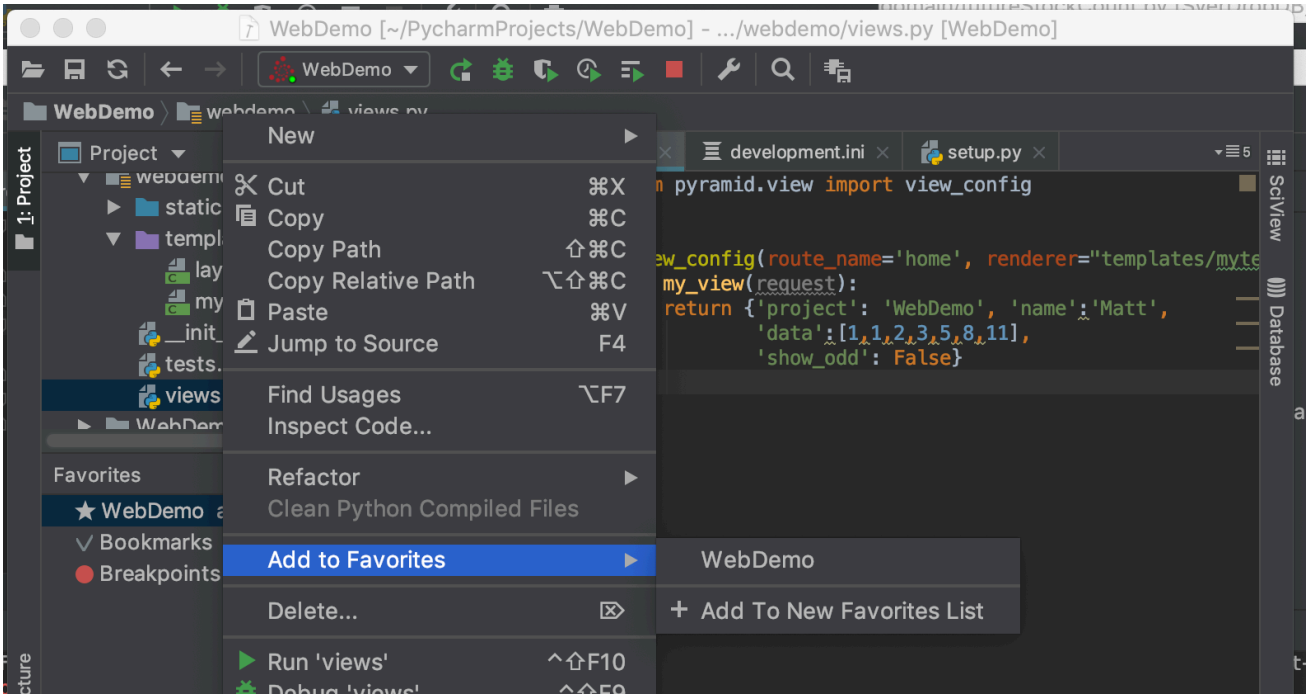


Figure 157: Figure illustrating right-clicking to add a file to the Favorites Window.

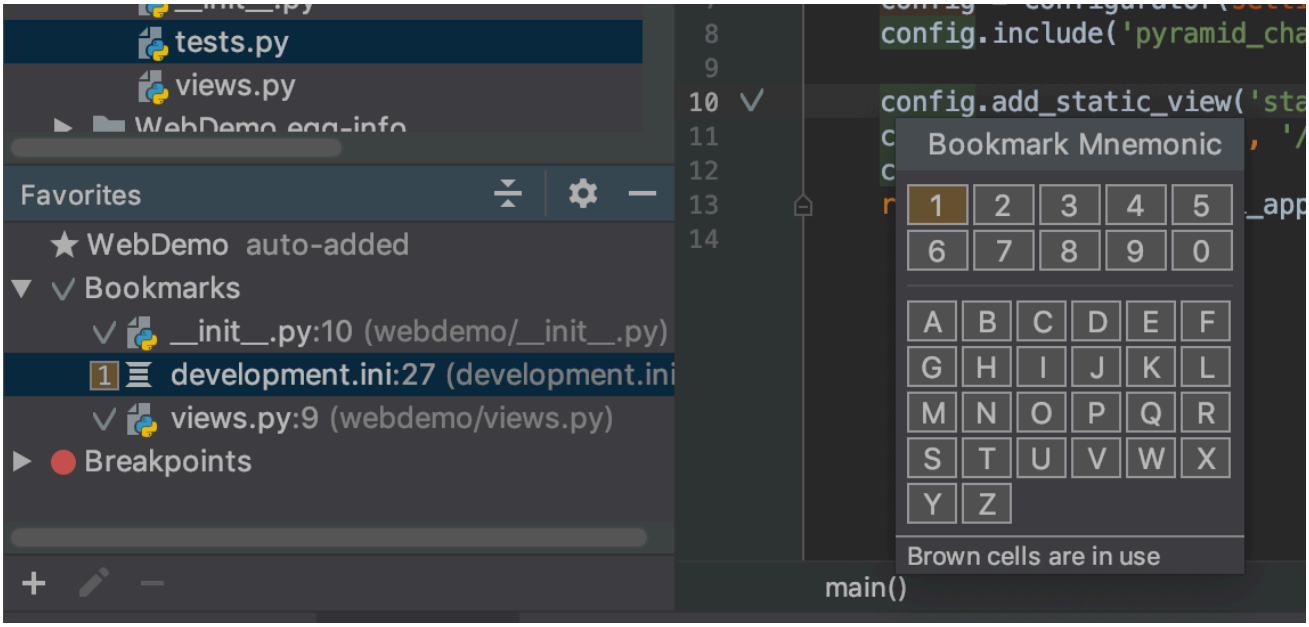


Figure 158: Figure illustrating adding a mnemonic bookmark.

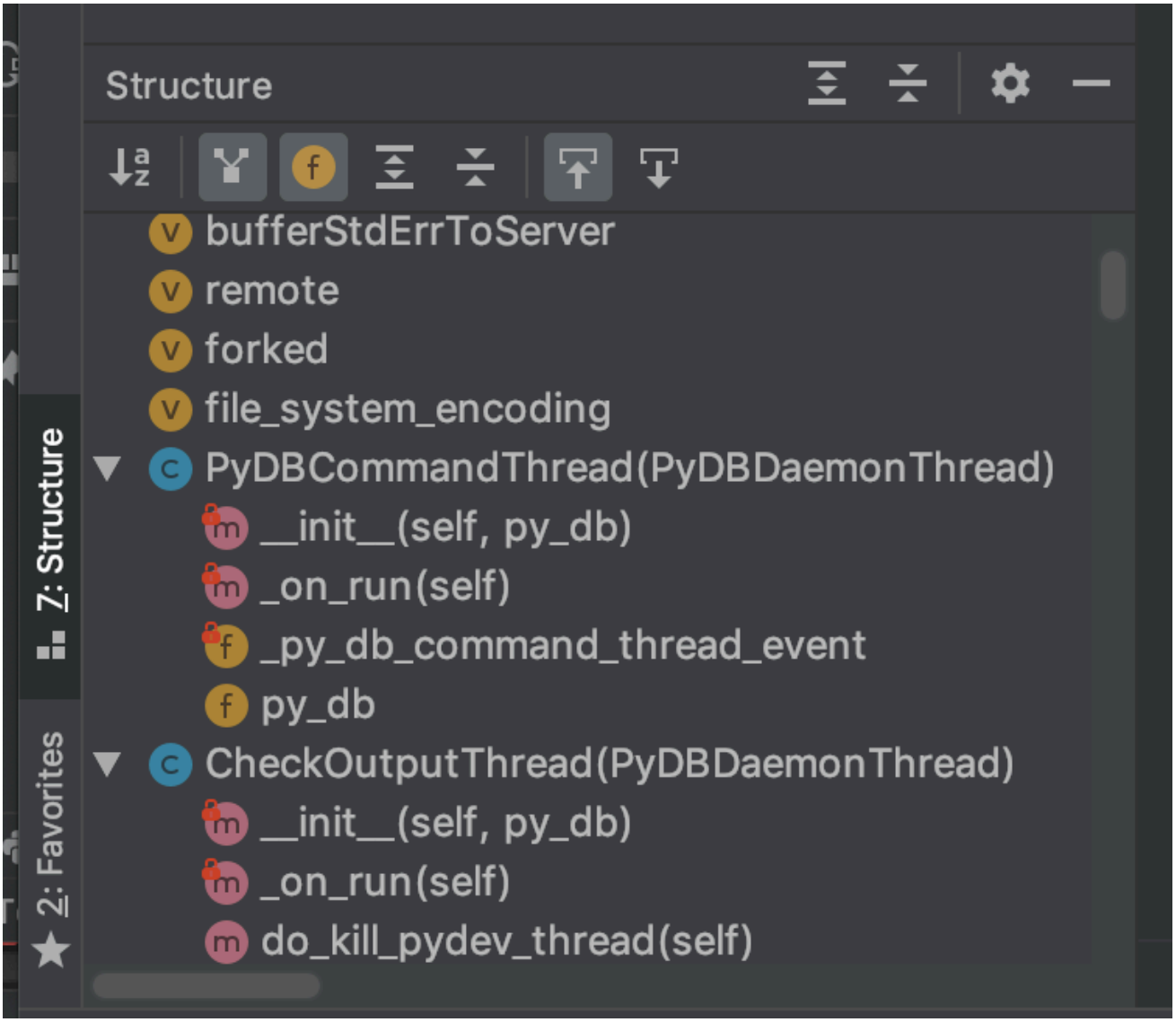


Figure 159: Figure illustrating the Structure Window.

breakpoint will appear in the favorite view. Again, by clicking in the favorites window, you can navigate to a breakpoint.

15.7 The Structure Tool Window

The *"Structure"* window (command-7 or alt-7) shows the contents of the editor window. It will display classes, methods, functions, and global variables. You can navigate to them by clicking in the window.

The structure window is useful for navigating a large file with many classes and functions. It is also useful for navigating to parent members of classes.

15.8 Summary

There are a bunch of tools that while not strictly necessary, can greatly improve your coding experience. The Todo window lets you track areas where you need to update or change the code. The console window has an integrated Python REPL for exploring and experimenting with your project. The Run window appears when you run code and shows you the output of your execution. The terminal window provides quick access to your virtual environment and command line scripts. The favorites window lets you navigate and remember files and code that you commonly use. Finally, the structure window summarizes the code in a file. Take advantage of these windows when working with your project.

15.9 Commands

- *"TODO"* - (command-6 or alt-6)
- *"Previous TODO"* - (alt-command-up or ctrl-alt-up)
- *"Next TODO"* - (alt-command-down or ctrl-alt-down)
- *"Filter TODO Items"*
- *"Autoscroll to Source"*
- *"View Options"*
- *"Preview Source"*
- *"Python Console"* - (Tools -> "Python Console..." or click on "Python Console" tab)
- *"Basic Code Completion"* - (ctrl-space)
- *"Set Value..."* - (f2)
- *"Execute Selection in Console..."* - (alt-shift-e)
- *"Run"* - (command-4 or alt-4)
- *"Rerun"* - (ctrl-f5)
- *"Stop"* - (command-f2 or ctrl-f2)
- *"Pause Output"*
- *"Terminal"* - (alt-f12)
- *"New Session"*
- *"Favorites"* - (command-2 or alt-2)
- *"Add to Favorites"* - (alt-shift-f, File -> "Add To Favorites", right-click on a file in the Project view, or right-click on the tab in the editor)
- *"Add All To Favorites"*

- *"Toggle Bookmark"* - (f11 or Navigate -> "Bookmarks" -> "Toggle Bookmark")
- *"Toggle Bookmark with Mnemonic"* - (command-f11 or ctrl-f11)
- *"Go to Bookmark <1>"* - (ctrl-1)
- *"Toggle Breakpoint"* - (command-f8 or ctrl-f8)
- *"Structure"* - (command-7 or alt-7)

15.10 Exercises

1. Using your favorite project try using the Todo window.
2. Using your favorite project try to run the code with the Console window.
3. Using your favorite project, open the terminal window and run some commands. Try to invoke Python. List the packages that pip has installed.
4. Using your favorite project try using the Favorites window to navigate between functions and files.
5. Using your favorite project try using the Structure window to explore the code in your project.

Chapter 16

Extending PyCharm with Plugins

16.1 Introduction to Plugins

PyCharm is actually built upon a series of plugins upon the IntelliJ platform. By leveraging common code, we get to take advantage of features like DataGrip and JavaScript within PyCharm. In this chapter we are going to look at managing, installing, and updating plugins.

PyCharm also supports writing your own custom plugin. You can do this for personal use or to allow others to leverage it. We won't be discussing plugin development as this requires Java knowledge. JetBrains provides extensive documentation on plugin development[#]_.

16.2 Exploring Plugins

If you open the *"Preferences..."* window (command-, or ctrl-alt-S) and click on "Plugins", you will see the plugins settings. The plugins settings has three tabs, "Marketplace", "Installed", and "Updates".

In the Marketplace tab, you can search for plugins, learn more about them, and install them. If you click on the name of the plugin, you will see more details about the plugin. Let's explore an example with the "Key Promoter X" plugin. We are huge fans of learning the keyboard shortcuts for PyCharm, and this plugin can be a huge boon for that. The Key Promoter X plugin uses the IntelliJ message system to inform you of the keyboard shortcuts for commands that you perform with the mouse.

The details page shows the version, release date, number of downloads, ratings, and any text the plugin author has provided. There is also a button to install the plugin. Click on the "Install" button to use the plugin. You will see a "Third-party Plugins Privacy Note", which you can accept.

Note

Many of these plugins are open source, while some are proprietary. As with many things on the Internet, there can be nefarious code. Buyer beware.

The plugin page in PyCharm will show a "Restart IDE" button. Click that to use the new plugin. Now when you perform a command using the mouse, this plugin will indicate the keyboard shortcut.

[24] http://www.jetbrains.org/intellij/sdk/docs/welcome.html

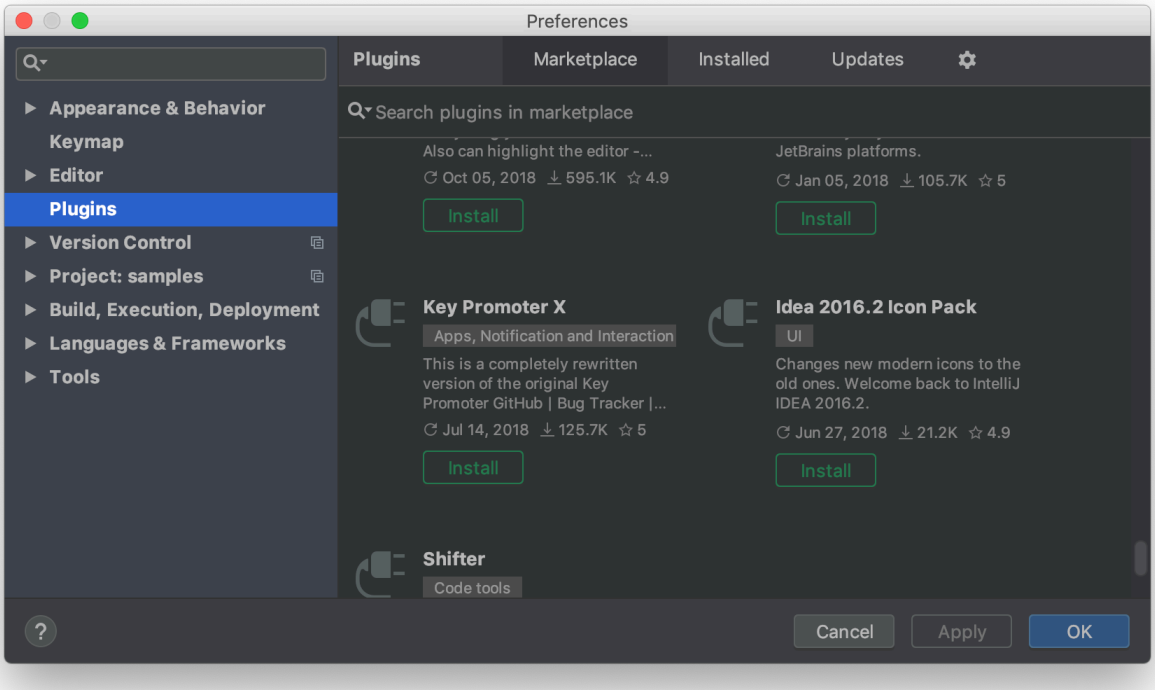


Figure 161: Figure illustrating the Plugins window in the Preferences.

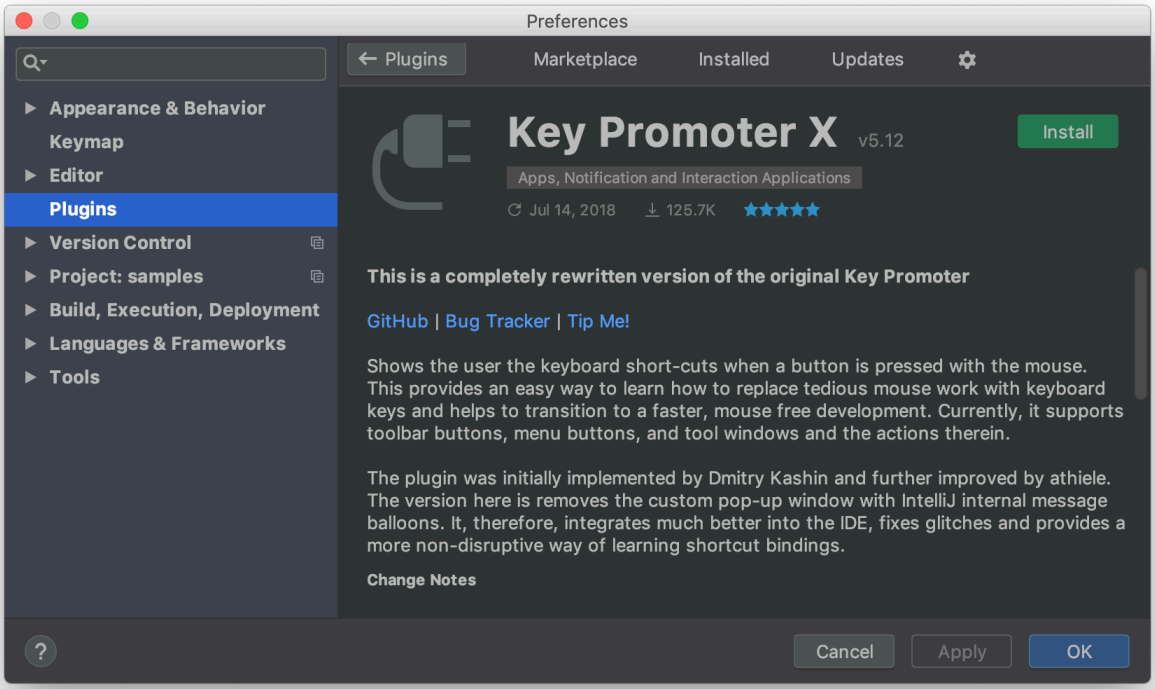


Figure 162: Figure illustrating the details of a plugin.

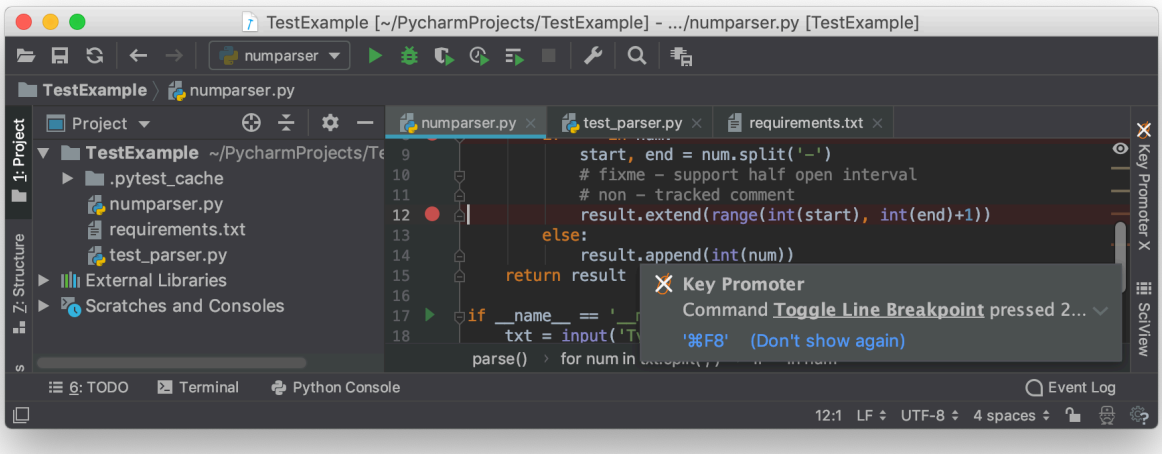


Figure 163: Figure illustrating the Key Promotor X showing the shortcut for the breakpoint command.

16.3 Installed Plugins

The "Installed" tab of the Plugins settings shows plugins that are installed in PyCharm. On this page, you can disable plugins as well. There is a section for "Downloaded" plugins, and "Bundled" plugins. Bundled plugins are preconfigured plugins that give PyCharm its base functionality. Typically you don't want to disable a bundled plugin. The "CVS Integration" might be a bundled plugin you can safely disable. The reason for using PyCharm is to take advantage of the plugins and integrations.

The "Updates" tab show if there are updates for any of the installed plugins. This allows you to use the latest and greatest.

16.4 Summary

PyCharm works by using plugins. The built-in plugins define the behavior for PyCharm. In addition, there are over a thousand plugins that you can install as well. If you need specific behavior for PyCharm, check to see if there is a plugin that supports that behavior.

16.5 Commands

- *"Preferences..."* window - (command-, or ctrl-alt-S)

16.6 Exercises

1. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/13-plugins` to explore existing plugins.

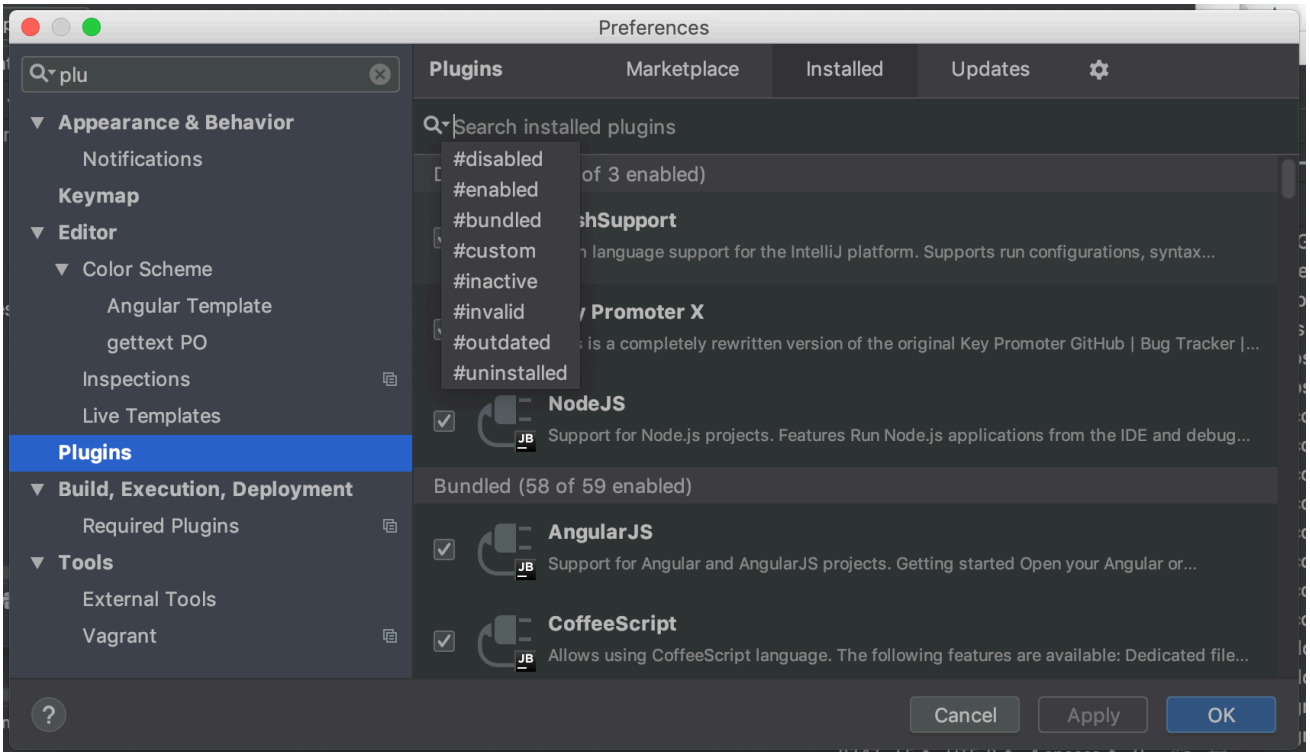


Figure 164: Figure illustrating installed plugins settings.

2. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/13-plugins` to discover new plugins.

3. Check out https://github.com/talkpython/mastering-pycharm-course Follow the instructions for the new project in `tree/master/your-turn/13-plugins` to install a new plugin.

Chapter 17

Conclusion

17.1 You've Done It!

Congratulations! You have explored much of PyCharm's superpowers. PyCharm is a powerful tool for aiding in all aspects of Python development. It can handle simple scripts, complicated applications, website development, and data analysis.

In this book, we have looked into many of the features of PyCharm. We looked at projects. Projects contain the configuration for a group of code. They indicate how PyCharm treats directories and files and how the Python environment is configured. Make sure that your project is properly configured to get the most out of PyCharm.

We explored the PyCharm editor features and learned about code intentions. This powerful feature allows you to clean up code and automatically fix errors. PyCharm has powerful code complete and documentation helpers to make creating code a painless activity. The "Find Action" feature lets us quickly discover or rediscover commands. In addition, we can change the keymaps or customize them to our liking.

PyCharm has excellent Git integration. You can use it to create branches, make commits, and see diffs. It also plays well with GitHub. PyCharm also maintains a local history as well that is independent of version control. There are also plugins and support for other version control systems, but Git is extremely popular.

Cleaning up code and fixing code smells is a common occurrence for programmers. PyCharm, with its deep understanding of code, is able to perform safe refactorings for you. PyCharm has a "Locate Duplicates" feature that lets you see potential refactorings. There is also support for renaming, moving code, and extracting constants and variables.

PyCharm professional includes the DataGrip plugin. This powerful tool provides a common interface to many databases. We showed an example in SQLite, but it works with Oracle, Postgres, MySQL, SQLServer, and more. PyCharm will diagram the structure of the tables and relations. It will let you query and view the data. In addition, you can update schemas from DataGrip. Finally, PyCharm has language injections, so if you start typing SQL in a string or code file, PyCharm will recognize it and provide code completion suggestions for the SQL.

Many Python developers work on web applications. PyCharm is well equipped to support many common Python frameworks, including Flask, Django, and Pyramid. There are templates for creating projects, and deep integration with templating languages. The runner utility lets you quickly try out the application without leaving PyCharm.

PyCharm professional also ships with robust JavaScript tooling. You can refactor JavaScript in PyCharm. You can also create TypeScript from PyCharm. PyCharm has plugins that support Angular, Vue, React, LESS, and more.

Developers spend a large amount of time reading and debugging code. PyCharm has a graphical debugger that enables you to set breakpoints, walk through code, and inspect values. PyCharm also has "inline debugging" and will show you the values of the variables next to the code itself. Using the debugging functionality should allow you to understand code and fix bugs with minimal effort.

PyCharm has support for Python packaging. You can run tasks on existing `setup.py` files. You use the "Create setup.py" command to walk you through creating a new package with minimal pain. In addition to making it easy to create Python code, you can also create packages and share that code with others. PyCharm has you supported through the whole process.

In addition to debugging, PyCharm has profiling capabilities. There is no additional configuration required. If you can run your program, you can profile it. The profile report will show you a table of call count and time spent. You can also view a call graph to determine what code is calling slow code.

Because Python is not a compiled language, testing is very important to ensure that code runs and works correctly. (Not that compiling guarantees that code runs correctly). Python supports the built-in unittest library and also works with the pytest library. Investing some time learning the pytest library will pay dividends in writing better Python code. PyCharm also ships with code coverage support. You can quickly look at the gutter and determine if you have a test that covers a line of code. Again, you get this functionality by clicking on a button. Once you have a run configuration, you get profiling and code coverage for free.

Python is one of the top tools of data scientists. PyCharm has support for workflows that this type of work requires. The scientific mode lets PyCharm get you quick feedback on data analysis and results of models that you run. There is also support for Jupyter notebooks from PyCharm.

There are a bunch of tool windows in PyCharm that will make your life easier. The Todo window tracks items you would like to change in the future. The run tool and Python console let you run code and inspect live variables. The terminal lets you interact with your virtual environment. You can track files and lines in files with the favorites window. Finally, the structure window gives you a high-level view of your classes, functions, and variables.

PyCharm has many plugins that you can take advantage of. If you need extra functionality, make sure that you check for a plugin. We showed how to use the Key Promoter X plugin to aid in remembering keyboard commands.

17.2 Your Turn

We introduced many features of PyCharm. How do you remember all of them? Each chapter in this book lists the commands it discussed. We recommend using copying those commands onto sticky notes and putting them somewhere where you can see them. Learning the commands can be a big benefit to productivity.

Also, each chapter has exercises. Make sure that you take advantage of those. By doing the exercises, you get the muscle memory of PyCharm and you will enhance your learning.

17.3 Thanks and Goodbye

Thanks for reading this book. We enjoyed writing it and hope that you have learned a lot while reading it and doing the exercises. Feel free to follow us on twitter, our handles are `@__mharrison__` and `@mkennedy`. If you enjoyed this book, please consider leaving a review on Amazon. That is the best way for other people to find out about it. Please feel free to link to if from social media.

About the Authors

Matt Harrison runs MetaSnake, a Python and Data Science consultancy and corporate training shop. In the past, he has worked across the domains of search, build management and testing, business intelligence, and storage.

He has presented and taught tutorials at conferences such as Strata, SciPy, SCALE, PyCON, and OSCON as well as local user conferences. The structure and content of this book is based on first-hand experience teaching Python to many individuals.

He blogs at `hairysun.com` and occasionally tweets useful Python related information at `@__mharrison__`.

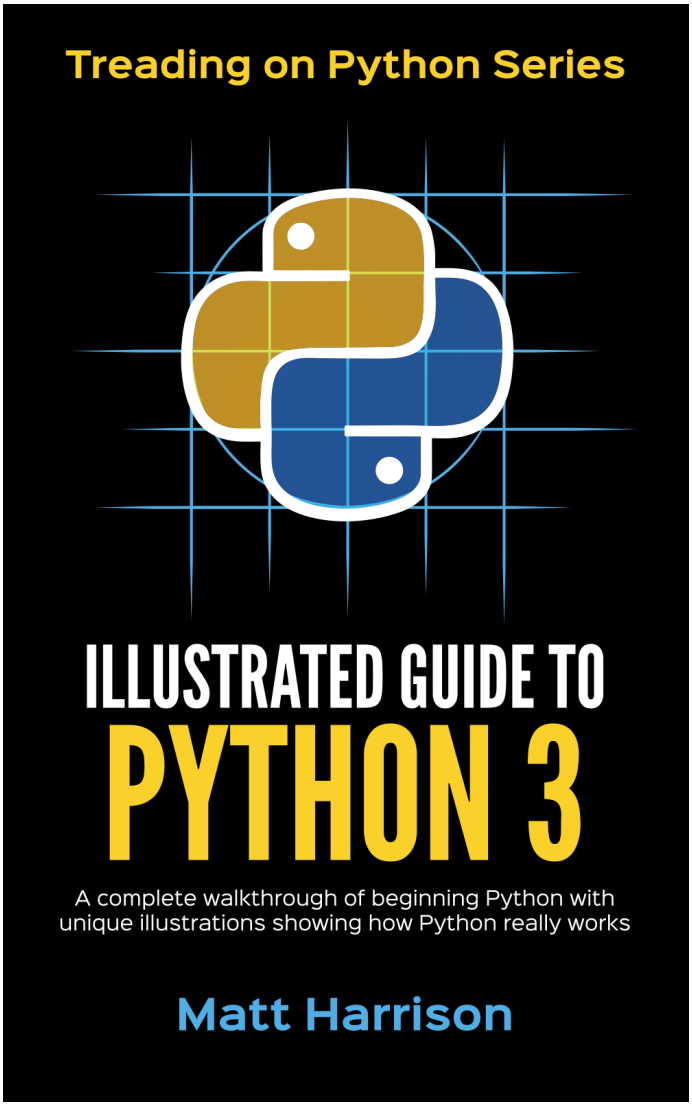
Michael Kennedy is the host of Python Bytes and Talk Python to Me. He is also the founder of Talk Python training and a Python Software Foundation fellow. Michael has been working in the developer field for more than 20 years and has spoken at numerous conferences.

Follow Michael on twitter where he is `@mkennedy`.

Index

Also Available

Illustrated Guide to Python 3



Illustrated Guide to Python 3 by Matt Harrison is the book to teach you Python fast. Based on years of experience teaching Python at small and large companies. Designed to up your Python game by covering the fundamentals:

- Interpreter Usage
- Types

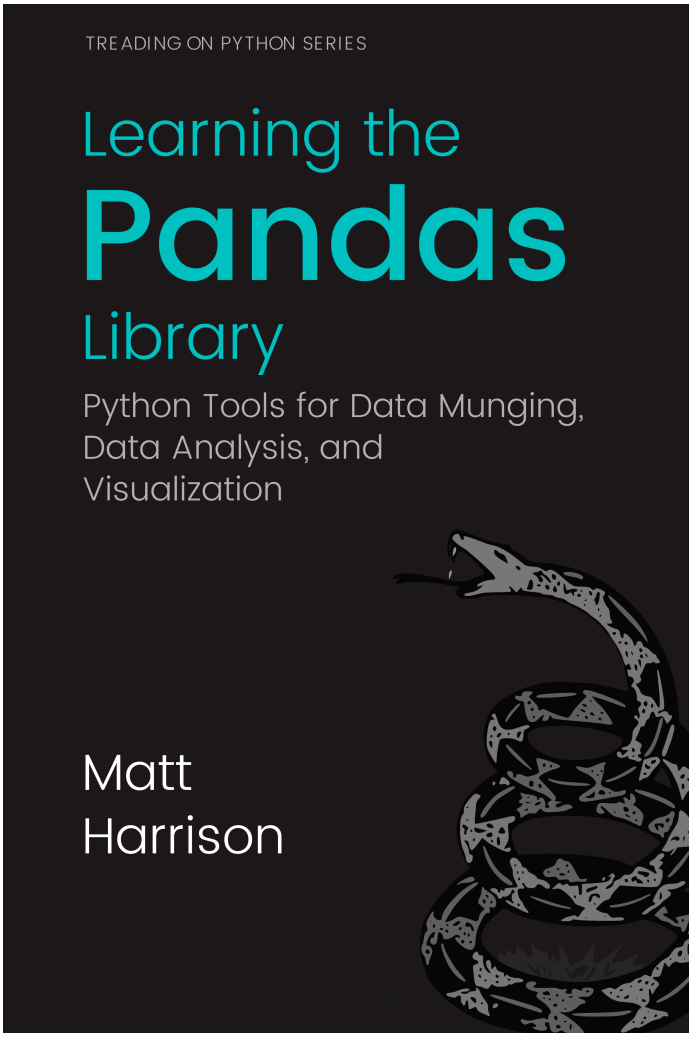
- Sequences
- Dictionaries
- Functions
- Indexing and Slicing
- File Input and Output
- Classes
- Exceptions
- Importing
- Libraries
- Testing
- And more …

Reviews

> If you want to go from being able to write runnable Python scripts to the beginning levels of being able to follow pycon tutorials, this is the book for you.
>
> Ben Jones

> I like this book for people learning Python. I teach Python Programming Fundamentals and this is one of the textbooks I recommend to my students. Very good explanations.
>
> PurpleDiane

Learning the Pandas Library



Learning the Pandas Library: Python Tools for Data Munging, Analysis, and Visualization by Matt Harrison is the book to teach you the ins and outs of Pandas. It includes many examples, graphics, code samples, and plots from real world examples.

- Data Structures
- Series CRUD
- Series Indexing
- Series Methods
- Series Plotting
- DataFrame Methods
- DataFrame Statistics
- Grouping, Pivoting, and Reshaping
- Dealing with Missing Data

- Joining DataFrames
- And more …

Reviews

If you already have some background in python but are not a total expert in pandas, I strongly recommend this book. I walks you and motivates through the data structures that are used in pandas en explains their raison d'être. The style is easy read.

Luis-Durham

This book provides tons of practical solutions that I've been able to implement at work immediately after reading. Definitely a nice guide to have handy.

Darian P

One more thing

Thank you for buying and reading this book.

If you have found this book helpful, we have a big favor to ask. As self-published authors, we don't have a big Publishing House with lots of marketing power pushing our book.

If you enjoyed this book, we hope that you would take a moment to leave an honest review on Amazon. A short comment on how the book helped you and what your learned makes a huge difference. A quick review is useful to others who might be interested in the book.

Thanks again!