 Python Introduction – Technical Notes 1

Table of Contents

[Summary 1](#_Toc2385994)

[Python Installation 1](#_Toc2385995)

[Create Virtual Environments 5](#_Toc2385996)

[Set up IDE – IntelliJ and PyCharm 6](#_Toc2385997)

[Exploration 18](#_Toc2385998)

# Summary

This document is a quick introduction to Python. It is intended to keep the document short and handy. It also serves as a proposal for all future introductions – pick a small topic and present it within 20 pages. It should be handy so that readers can follow steps to produce the same results.

Python has been around since 1991(<https://en.wikipedia.org/wiki/Python_(programming_language)> ). When we talk about Python, in general we refer to CPython (<https://en.wikipedia.org/wiki/CPython>), in contrast to Jython (JVM based implementation) or IronPython (.NET based implementation). In the past 20 years, Java dominants the enterprise world because of JCP and Spring Framework. JCP picks up many best practices of the industry (JEE), such as JDBC, JMS, etc. For example, PyODBC is nowhere close to JDBC in terms of performance (3000 rows/sec insert vs 70000 rows/sec). Spring Framework brings in the ease of use, especially the configuration management and IoC, which is a rare perfect solution. But Python is not at rest during the same time, and it pounds the industry with more tools heavily. With Python 3 readiness (It took 10 years to move from Python 2 to 3) and all the needed tools for enterprise applications and quantitative computing, it is time for Python to take off as the leading language in the next 20 years. There will be some minor hiccups, but they can be resolved with not-too-much hassle. Kotlin is created by JetBrains (IntelliJ’s home company) as an opportunity to replace Java(both in mobile and server) under such pressure. Both Google and Spring Framework embrace it now.

Python is a scripting language in nature since it needs the interpreter to run. It is a common sense to worry about the performance in large scale applications. The resolution is that in general we assume python code is fast. If we find slowness in some segment of code, we rewrite that part in C/C++ and then use SWIG wrapping it and bringing it back to Python (<https://en.wikipedia.org/wiki/SWIG>).

This document will assume readers know nothing about Python and walks through the necessary steps with readers so that at the end readers can start writing their own applications. Like exploring any programming language, we touch the steps for code running, debugging, and profiling because I believe these are the basic building blocking.

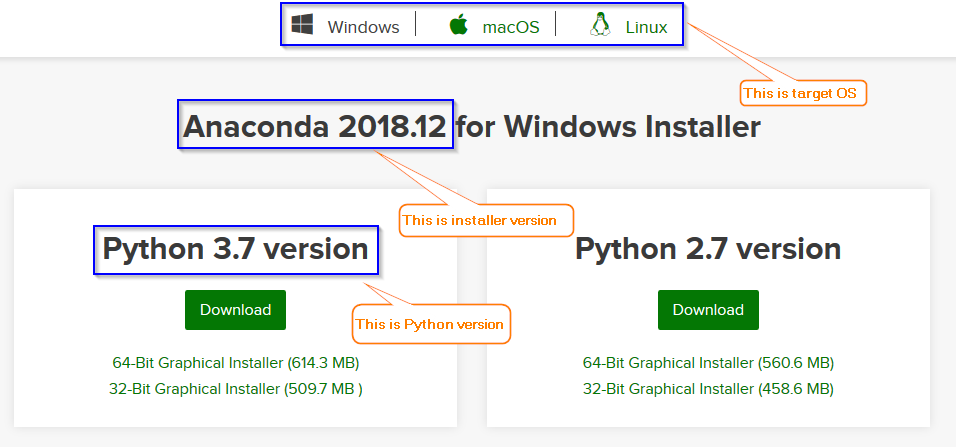
# Python Installation

Back in the old days (and now too), we can install Python with the installer from the official Python site: <https://www.python.org/downloads/>. But because many add-on packages require C/C++ compilation on a particular OS, we use other distributions, or bundle installers, to install pre-compiled packages (We use bundle/distribution indistinguishably in this document). To name a few packages that require C/C++ compilation, numpy, PyQt, python-ldap, pykerberos, etc. The closer we get to math or OS services, the more likely we run into that. 3 well known distributions are:

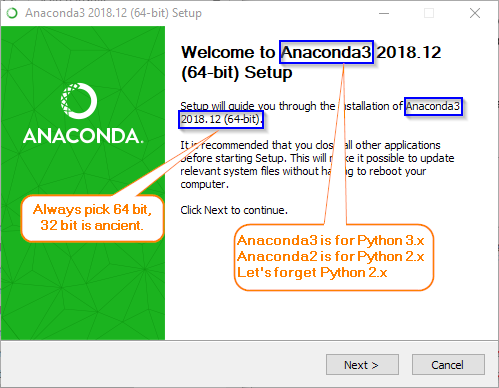
* Ananconda: <https://www.anaconda.com/distribution/>
* Enthought: <https://www.enthought.com/product/canopy/>
* ActivePython: <https://www.activestate.com/products/activepython/>

Using SWIG and C/C++ is an advanced topic in Python and we don’t cross that in 90% of the usages. Ananconda gains a lot of popularity recently because of the ease of use and letting us avoid C/C++ compilation.

First, download the target bundle from: <https://www.anaconda.com/distribution/> or your designated repo:

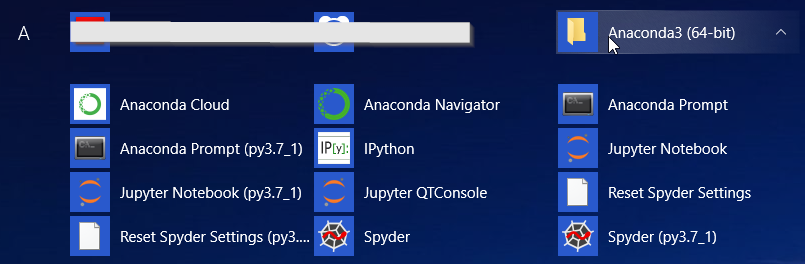


Do not get confused by the versions here: one for Python version and one for installer version. Let’s forget Python 2.x, that’s history. The installer page is also a little bit confusing:



The rest of the installation should be smooth. Please verify the following:

* python.exe is in the installation directory.
* Installation directory is on the PATH.
* The Anaconda menu looks like this:



There are 2 environment variables in Python, PYTHONHOME and PYTHONPATH (similar to JAVA\_HOME and CLASSPATH in Java).

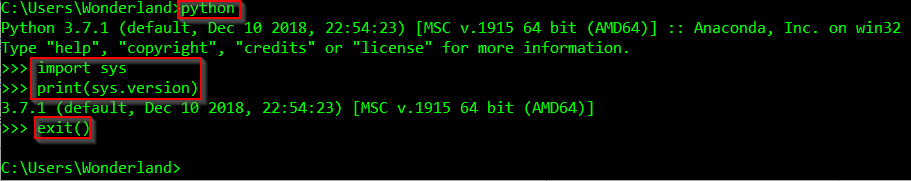
<https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHOME>

<https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH>

Our own code directory should be added to PYTHONPATH before execution (Just like in the Java world, we add our own jars to the CLASSPATH).

PYTHONHOME will be quite useful when creating virtual environments later. Please set PYTHONHOME to the python installation directory where you can find python.exe.

The command line interpreter is interactive:



We could also run Python code in the non-interactive way(bold lines are input, others are output):

C:\>**python -c "import sys; print(sys.version)"**

3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]

Along with the installation, there is a powerful tool, conda, under $PYTHONHOME/Scripts, which should be in your PATH too (set by the installation, there are also other tools in the same folder). To list all the packages installed by the Anaconda installer:

C:\>**conda list**

# packages in environment at D:\0dev\Anaconda3:

#

# Name Version Build Channel

\_ipyw\_jlab\_nb\_ext\_conf 0.1.0 py37\_0

alabaster 0.7.12 py37\_0

anaconda 2018.12 py37\_0

anaconda-client 1.7.2 py37\_0

anaconda-navigator 1.9.6 py37\_0

anaconda-project 0.8.2 py37\_0

asn1crypto 0.24.0 py37\_0

astroid 2.1.0 py37\_0

astropy 3.1 py37he774522\_0

atomicwrites 1.2.1 py37\_0

. . . . .

This shows the package names, versions, build tool, and conda channels. Builds are like python packager or C++ compiler, such as vc\_9 or <pip> (a packager in Python). Channels are default to anaconda, others are conda-forge or pypi. Channel concept is very close to Maven’s repository, for more information, check the official document:

<https://docs.anaconda.com/anaconda/user-guide/tasks/using-repositories/>

Package/library dependency management is a big change since Linux built rpm and Maven was created in Java. The same concept has spread out to other areas:

* C# has Nuget, <https://www.nuget.org/>
* Linux has rpm/apt/etc
* JavaScript has Node.js and npm, <https://nodejs.org/en/> and <https://www.npmjs.com/>
* Windows software installation has chocolatey, <https://chocolatey.org/>

To illustrate how to use conda to install new packages, we are going to install a Linux/Unix command line package so that we could run common linux/unix commands on Windows. If needed, we need to set HTTP proxies first:

C:\>**set http\_proxy=. . .**

C:\>**set https\_proxy=. . .**

C:\>**conda install unxutils**

This is to tell conda to install the package unxutils from the official conda channel, anaconda. It also installs any dependent packages - this is one of the benefits when using conda. Otherwise we have to figure out the dependencies and install them one by one in the right order. This package is installed in the Scripts directory under Python home and contains a lot of UNIX/Linux commands. For example, we could use grep to find out whether a package is installed

C:\>**conda list | grep numpy**

numpy 1.15.4 py37h19fb1c0\_0

numpy-base 1.15.4 py37hc3f5095\_0

numpydoc 0.8.0 py37\_0

Other examples of unix commands are cat, tail, cut, curl, wget, tee, and etc.

Typing in conda without parameter will bring up the help page.

If conda channels do not have the package you need, you could still use the Python standard way to install it, e.g.

C:\>**pip install pyodbc**

Pip uses the official Python repo: <https://pypi.org/>. To update a package with pip

C:\>**pip install –U pyodbc**

To update a package with conda (if a package is installed with conda)

C:\>**conda update pyodbc (or simply use all here)**

Please check documents for more options.

You may also use switch **pip --proxy http://...** to use network proxy.

During installation, Anaconda downloads packages from its channels, which are printed in the console. In case you want to play around with the default conda channels, e.g., you may add the following channels in your <userhome>/.condarc:

channels:

* <File://c:\myrepo>
* http://repo.anaconda.com/pkgs/conda-forge

Then we can use conda install –c conda-forge <package name> to bring in the package. For more information on repos and channels, run **conda -info** or check:

<https://repo.anaconda.com/pkgs/>

<https://anaconda.org/anaconda/repo>

<https://anaconda.org/conda-forge/repo>

<https://conda-forge.org/>

Still the biggest challenge is the package dependency. When installing a new package, often it downgrades some of the existing packages, we need to make a sensible judgement whether we want to do it or not. In general, pip is less destructive.

Occasionally, when using pip to install packages it cannot uninstall the old version installed by conda, we need to simply delete the old package folders and then run pip. Packages are in general installed under $PYTHONHOME/lib/site-packages.

Both conda and pip can take the requirements.txt for bulk installation. This file is similar to Maven’s pom.xml for packages.

C:\>**pip install –r requirement**

With conda

C:\>**conda install -file requirements.txt**

In general, requirements.txt should be in the Python project folder. It has the following form:

psutil>=2.0

python-dateutil==2.2

django>=1.10,<1.11

The general format is specified here: <https://www.python.org/dev/peps/pep-0440/#version-specifiers>

One last comment is about the PATH. Since CPython is implemented in C/C++, and a lot of 3rd party packages are also implemented in C/C++ with Python wrapper, PATH plays a more important role in Python than in Java. In general, we should add the following folders to the PATH:

* $PYTHONHOME
* $PYTHONHOME/Scripts
* $PYTHONHOME/DLLs
* $PYTHONHOME/lib
* $PYTHONHOME/lib/site-packages
* $PYTHONHOME/lib/site-packages/pythonwin (if you are on windows)
* $PYTHONHOME/lib/site-packages/win32
* $PYTHONHOME/lib/site-packages/win32/lib
* $PYTHONHOME/Library/bin
* $PYTHONHOME/Library/usr/bin
* $PYTHONHOME/Library/mingw-w64/bin

These folders are where packages install their C/C++ libs/executables, or some commonly used libraries used by other packages. You may not have all of them for now since you haven’t installed the packages that utilize all of these folders.

# Create Virtual Environments

From time to time, you want to work on different Python versions, or with different packages. Especially when you play around, you don’t want to “pollute” the official installation. That’s when virtual environments come in. You may install different packages on different Python versions in different environments and they are pretty much isolated from each other (except you need to change PYTHONHOME and PYTHONPATH to redirect the folders)

For example, we can create a new virtual environment named p35 with Python version 3.4

C:\>**conda create –n p35 python=3.4**

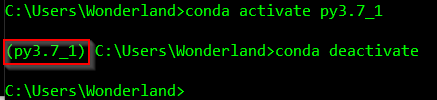
This will create the environment in the default location: $PYTHONHOME/envs. You can overwrite this location with -p

C:\>**conda create –p C:\myenv\p35 python=3.4**

Try conda create –h for help. Once the new environment is created, you can install packages like mentioned before. But before you do that, you need to switch to that environment first.

C:\>**conda activate C:\myenv\p35**

Run deactivate for getting out of that environment. Notice that the environment name is pre-appended to the system prompt once we activate an environment:



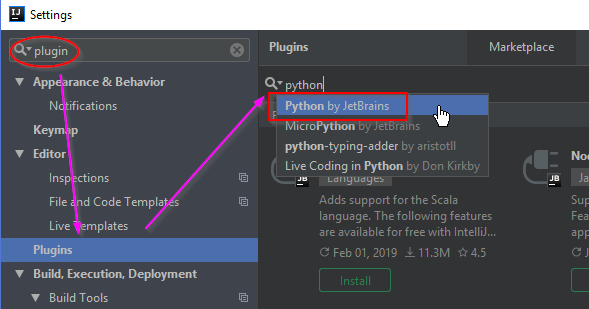
Now the Python SDK is fully set up.

All virtual environments are written into <user home>/.conda/environments.txt

# Set up IDE – IntelliJ and PyCharm

There are many IDEs available for common languages, IntelliJ, VS Code, Emacs, vi, or notepad++, to name a few. Pick up your favorite one. When you go to a battle, you want to have a sharp/effective weapon, like Elf’s swords. So make sure your choice fit you well, powerful and swift. JetBrains has a suite of Elf’s swords. There is minor difference between IntelliJ and PyCharm (with same version, don’t compare pro version in one against community version in another). The Python project setup is not as good as Java project setup. In Java, we can group all IntelliJ files into one folder so that they don’t pollute our source folders. In Python, I am not able to put all of them in one dedicated folder. But it’s a minor hiccup, I just put all IntelliJ files at the root.

First, please install IntelliJ, then install the Python plugin. Open IntelliJ, open the menu File|Settings. Search for proxy in the upper left textbox. Then click on HTTP Proxy in the left panel and set up the proxy in the right panel. Now go back to the search box and search plugin



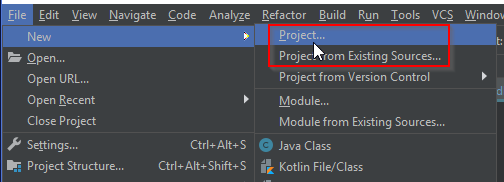
Search plugin on the left, click on the Plugins. Then search python on the right and then select Python by JetBrains. Now click on the upper right corner Install button.

The following steps are in the file, you may use IntelliJ to open the unzipped directory.

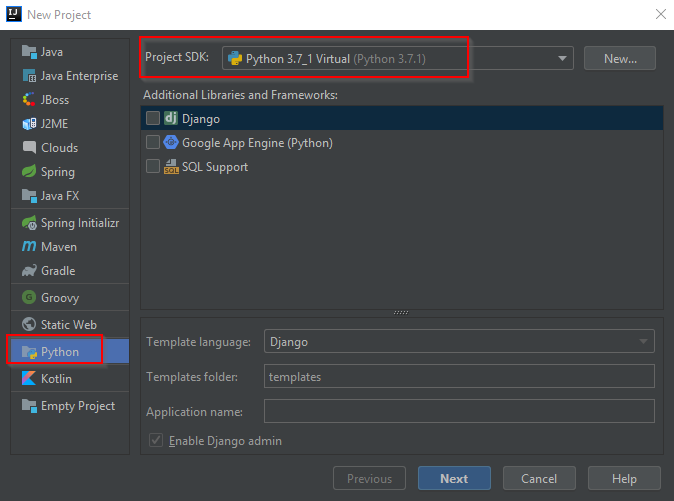


Every Python file is runnable.

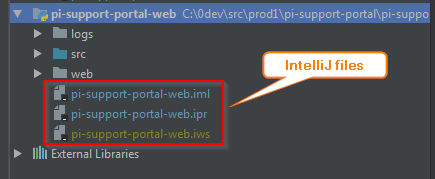
You can now create a new project, or a project from existing sources



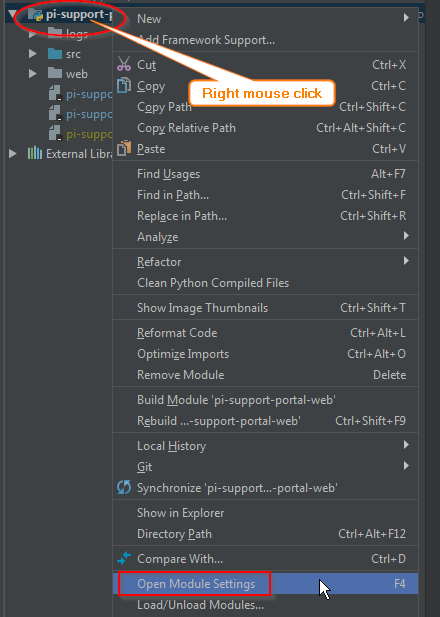
Select Python and SDK in the next screen:



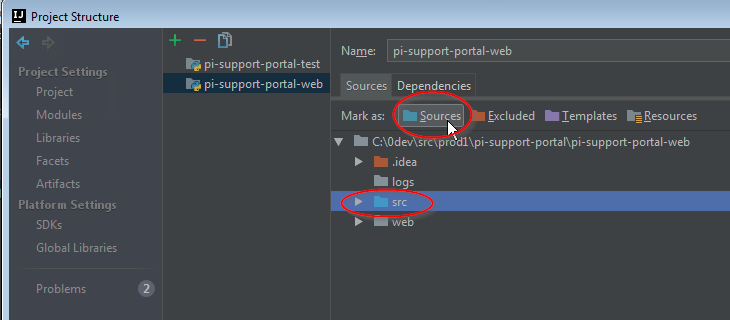
A default module is also created in the project.



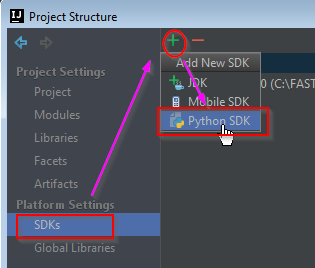
To set up the Python SDK, right click the folder row:



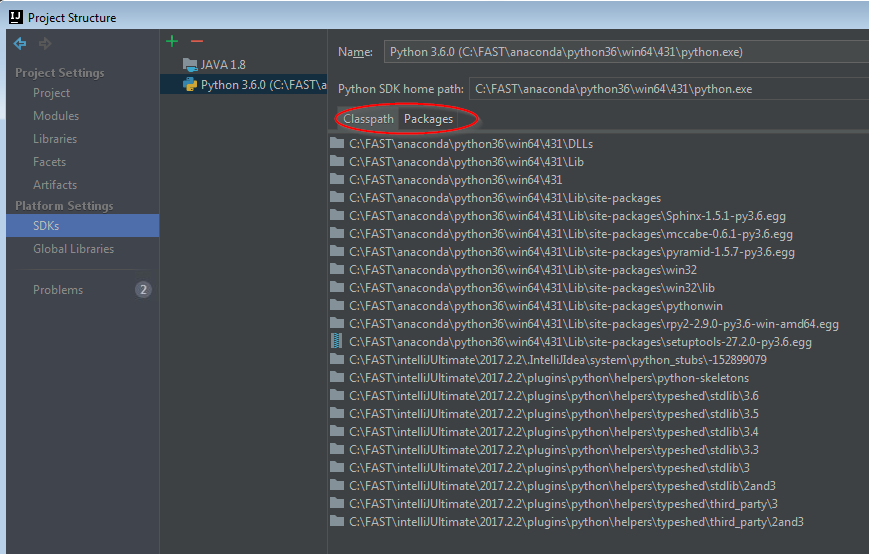
Click on the “Open Module Setting” to bring up the dialog:



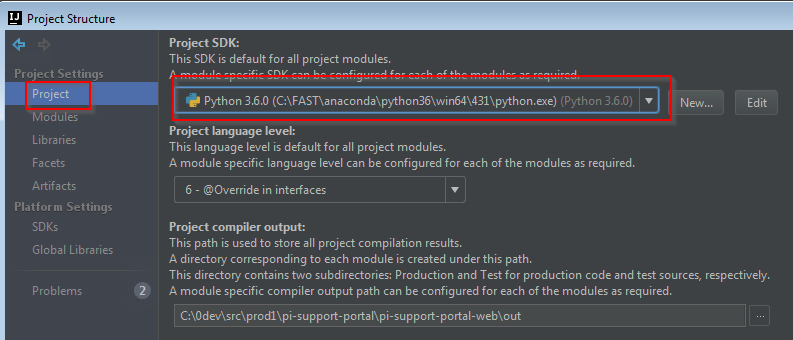
Click src and then Sources, now src should be blue. Next we set up the Python SDK.



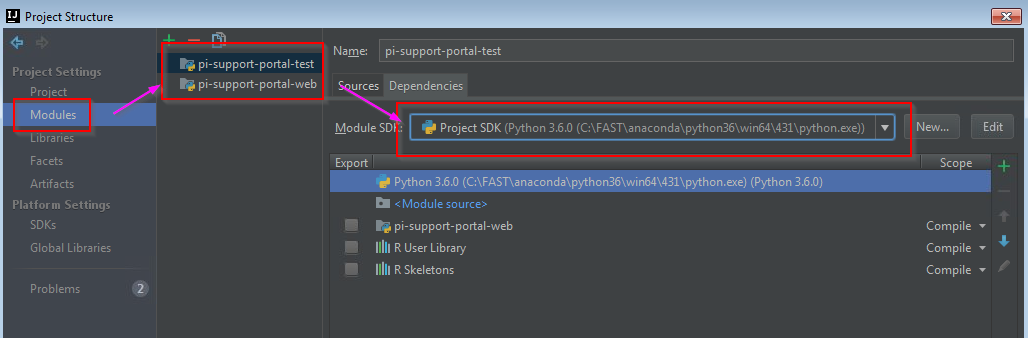
Click “Add local” and select the python.exe in your newly created environment. Then we should see something similar to this (depend on how many packages you install):



It’s going to take a while for IntelliJ to build the index for this SDK. Then we switch to the project to select the Python SDK.



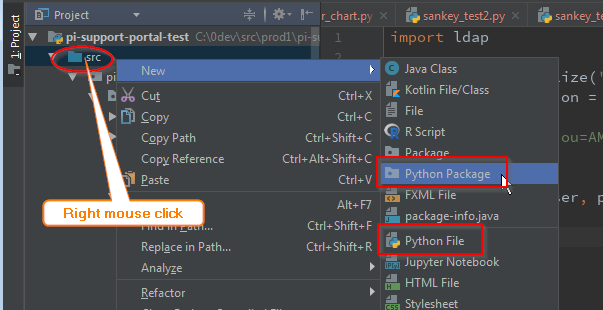
Then in each module we need to map the module SDK to the project SDK:



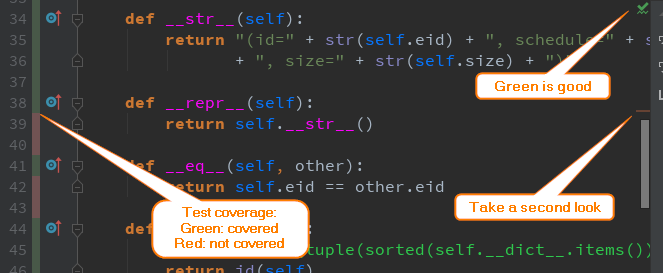
Depends on which virtual environment you use, you need to tweak the PYTHOHOME, PYTHONPATH, and PATH variables. Restart IntelliJ after tweaks. Most of the missing file problems after installation are due to misconfiguration on these 3 variables.

You may add additional modules as you want, and repeat the module setup as above (set src folder to blue and select SDK).

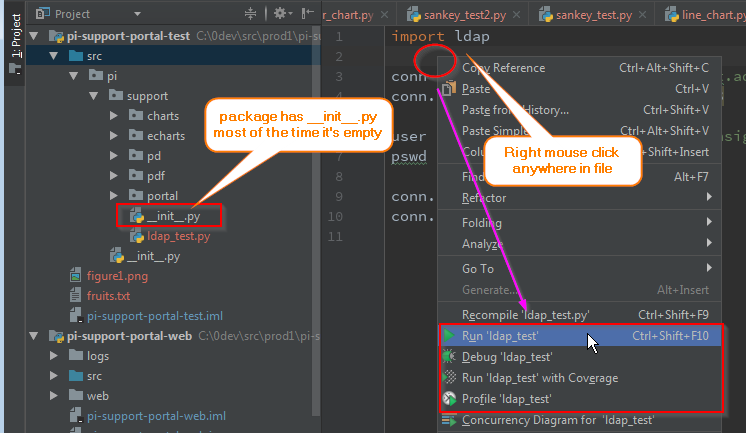
Now it’s time to write code. Right mouse click on the src folder to bring up a new menu where you can add a package or file:



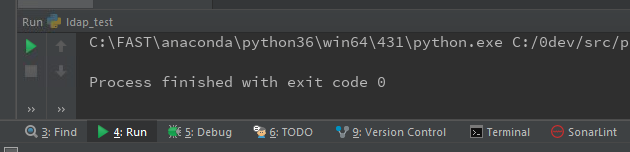
The editor support PEP 8 checks. As in Java, the right scrollbar in the editor is the place for hints.



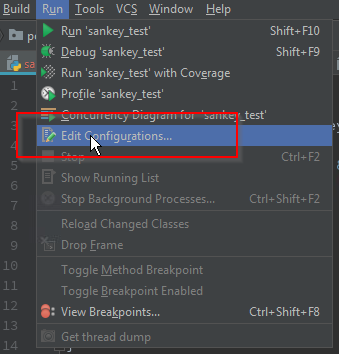
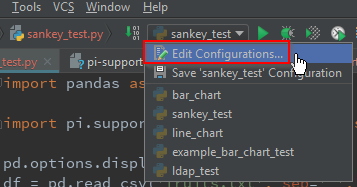
Now we put some python code there and bring up the run menu:



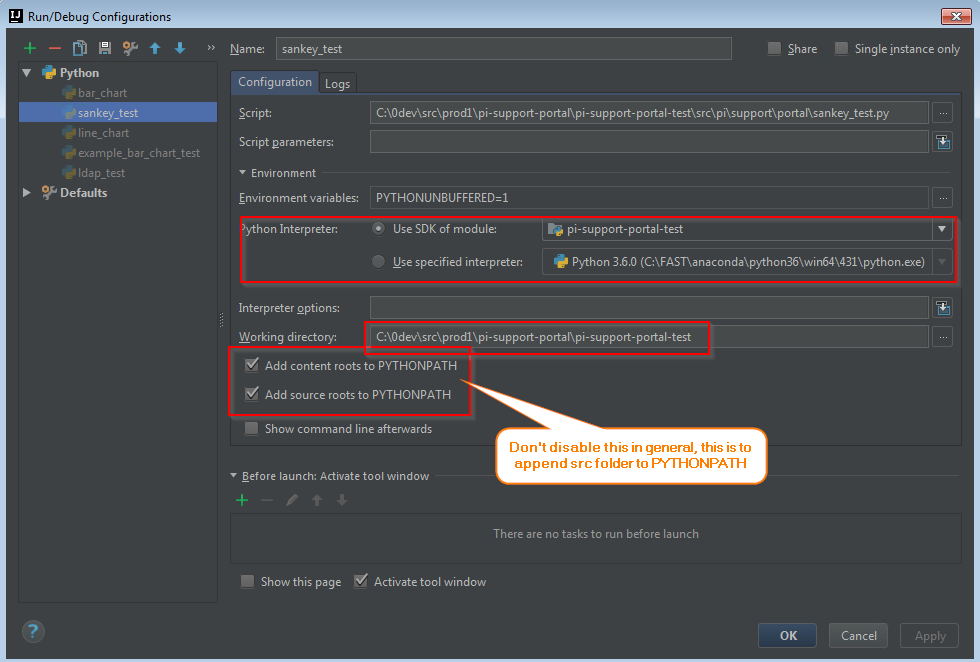
You could run/debug/test/profile the code. The result is at the bottom of the IDE:



We can alternate the default runtime configuration through 2 ways:

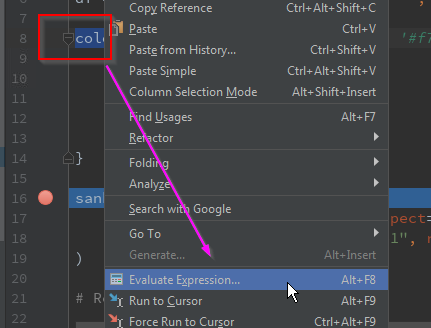


This brings up the configuration dialog:

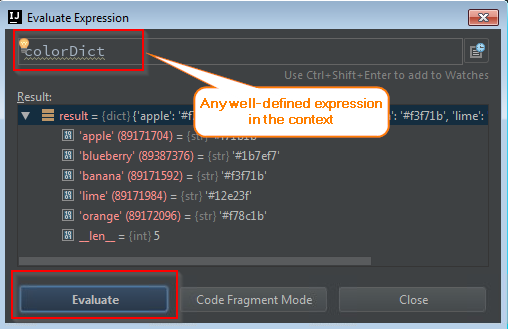


We may change the working directory here if there is any resource location resolved.

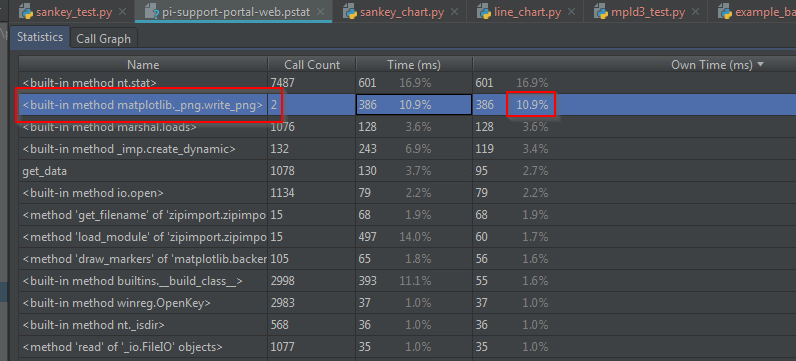
During debug, you can evaluate any expression by highlighting it and then right mouse click:



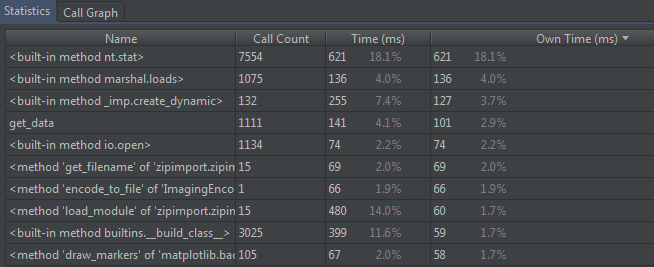
Now we can evaluate any expression(hit enter):



If we run profiling, we get the pstat file. The first row is from profiling, the second row is our calling code. It’s taking a lot of time to write a png file.

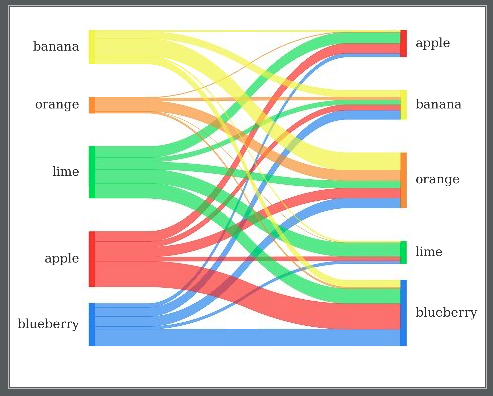


You can right click the line to go to the python code. Now let’s change the png format to jpg format and check the numbers again:



The same method takes no time at all.

The resulting picture is the same:



For quick script play around, you can go to the menu File | Tools | Python Console to bring up a Python console at the bottom. This is very handy if you want to run something quick.

For more information on IntelliJ performance support, check the document: <https://www.jetbrains.com/help/pycharm/profiler.html>.

For more information on Python profiling:

<http://www.marinamele.com/7-tips-to-time-python-scripts-and-control-memory-and-cpu-usage>

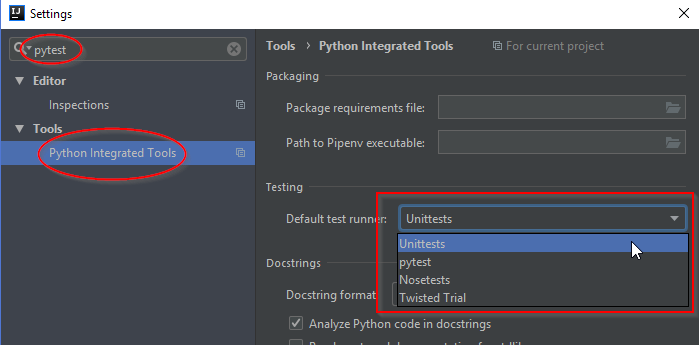
<https://zapier.com/engineering/profiling-python-boss/>

<https://docs.python.org/3/library/profile.html>

<https://stackoverflow.com/questions/582336/how-can-you-profile-a-python-script>

To detect memory leak, check: <https://pythonhosted.org/Pympler/muppy.html>

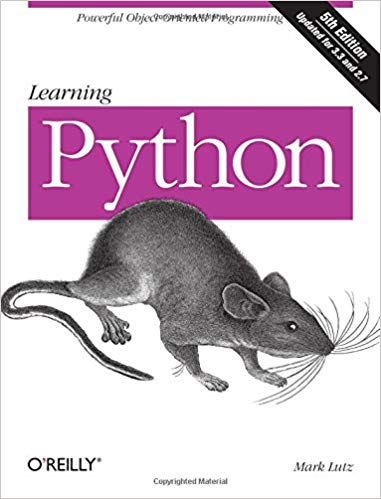
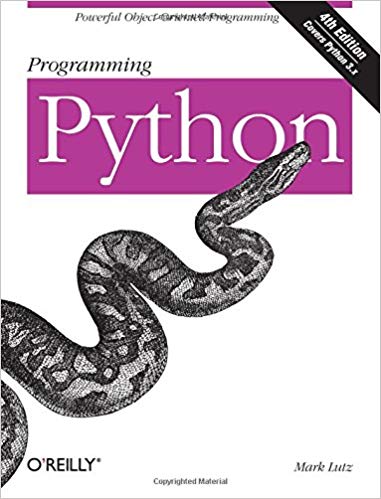
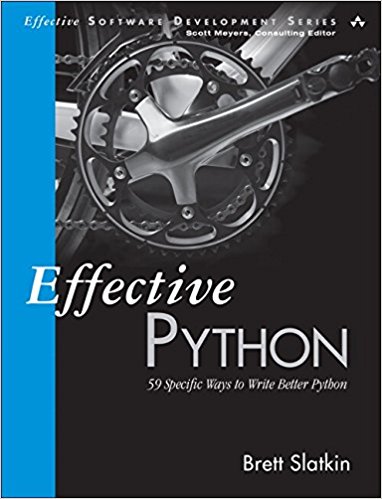
For testing, prefer pytest package since it’s more versatile and convenient than the built-in unittest package. Please install pytest and coverage packages first. There is no surprise if you are familiar with Java’s Junit. Same as test coverage, we can see the stats on the left tree panel and left red/green bars in the editors. IntelliJ supports the following 4 test frameworks:



Now coding should be a pleasant journey with these handy tools.

# New Frontier

Python is very fluid – there are many ways to do one thing, only a handful ways are **pythonic**. It’s easy to get on hand, but it takes a while to write beautiful and maintainable code. Here are some good books to lay a solid foundation for the language (We will mark this level as **L0**, ground).

Amazon’s book ranking: <https://www.amazon.com/Best-Sellers-Books-Python-Programming/zgbs/books/285856>

Online books:

<https://learnpythonthehardway.org/book/>

<https://automatetheboringstuff.com/>

There are thousands of libraries, taking a lot of time even to explore them. The fastest way to learn is to learn from each other! Here are some of the most commonly used general-purpose packages (Keep in mind that there are more than one way to do things):

* Pydot/pyGraphiz: graph plotter
* PyQt/wxPython: for desktop GUI
* Flask/Tornado/Django for web framework
* PyODBC: for database operations, similar to JDBC
* SqlAlchemy: for ORM, similar to Hibernate in Java
* Dask/pyspark/dispy: distributed computing across machines, similar to Spark/Yarn in Java
* xlrd/xlwt: for excel, python-docx for word and python-pptx for ppt.
* request/httplib2: handle HTTP requests
* reportlab: generate PDFs
* pywin32: interact with Windows
* scrapy/beautifulsoup: web scrapping
* Pillow/PIL: handle images
* Twisted: event driven network engine
* Fabric/paramiko: remote SSH
* Cubes: OLAP engine
* rpy2: bridge to R
* NetworkX: graph library

This is **L1**. Picking up the best tools is time consuming and needs team collaboration. Comparing tools in each usage and selecting winners deserve its own document.

Financial libraries:

* Backtesting: <http://pmorissette.github.io/bt/>
* quandl: <https://github.com/quandl/quandl-python>, financial data download
* quantopian: <https://www.quantopian.com>, algorithm testing
* Zipline: <http://www.zipline.io/> backtesting

Here is the list that forms the base for machine learning (This is **L2**):

* Pandas: data frames, similar to Spark in Java/Scala, data frames in R
* numpy: math/stats
* scipy/sympy: math/stats
* matplotlib/seaborn: charting/visualization
* theano: multi-dimensional arrays

Here are my baseless guess for machine learning and data science related packages (This is **L3**):

* Tensorflow: deep learning
* keras: deep learning
* pytorch: deep learning
* caffe2: deep learning
* MxNet
* scikit-learn: machine learning on top of scipy
* imbalanced-learn:
* LightGBM/XGBoost/CatBoost
* nltk: natural language toolkit
* spaCy: NLP
* genism: NLP
* pyflux: time series prediction
* pybrain: neural networks
* neupy: neural networks
* statsmodels

Here are some references on more libraries:

<https://www.kdnuggets.com/2018/12/top-python-libraries-2018.html>

<https://tryolabs.com/blog/2018/12/28/top-10-python-libraries-2018/>

<https://www.edureka.co/blog/python-libraries/>

Finally the grand list of python libs: <https://github.com/vinta/awesome-python>

Other ML references:

Stanford ML/Andrew Ng: <http://www.holehouse.org/mlclass>

CMU ML/Tom Mitchell: <http://www.cs.cmu.edu/~ninamf/courses/601sp15/lectures.shtml>

Google: <http://playground.tensorflow.org>

Microsoft: <https://studio.azureml.net>

Tensor board/Projector: <http://projector.tensorflow.org/>

Kaggle Data Repository: <https://www.kaggle.com/>

UCI Data repository: <https://archive.ics.uci.edu/ml/datasets.html>

