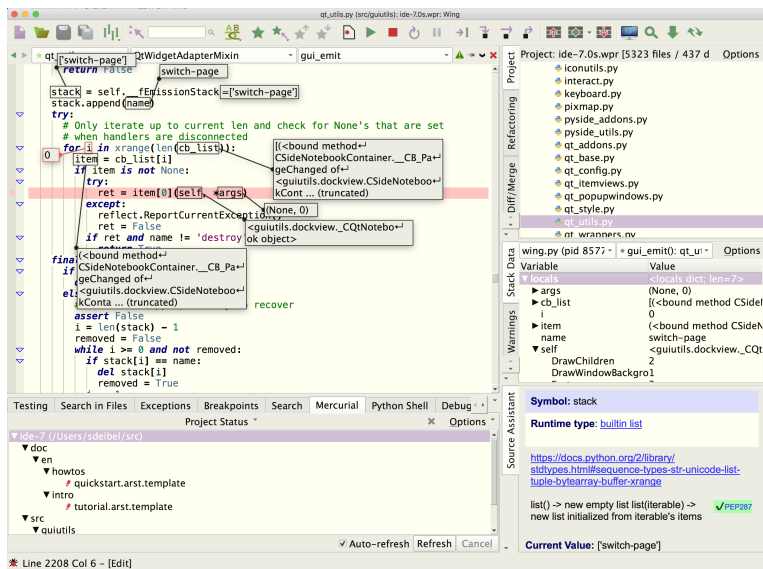# How-Tos

Version 8.3.3

This collection of How-Tos explains how to get started using Wing Pro with specific Python frameworks, tools, and libraries for web and GUI development, 2D and 3D modeling, scientific analysis, compositing, rendering, game development, and much more.

These How-Tos assume that you know how to use the Python framework or tool being discussed and that you are already somewhat familiar with Wing. To learn more about Wing see the Quick Start Guide or Tutorial.

Intelligent Development Environment for Python" are trademarks or registered trademarks of Wingware in the United States and other countries.

Disclaimers: The information contained in this document is subject to change without notice. Wingware shall not be liable for technical or editorial errors or omissions contained in this document; nor for incidental or consequential damages resulting from furnishing, performance, or use of this material.

Hardware and software products mentioned herein are named for identification purposes only and may be trademarks of their respective owners.

```
Wingware
P.O. Box 400527
Cambridge, MA  02140-0006
United States of America
```

Contents

How-Tos for Specific Environments

The following How-Tos explain how to get started using Wing with specific types of Python environments provided by virtual environments, containers, virtual machines, and remote hosts.

1.1. Using Wing with virtualenv
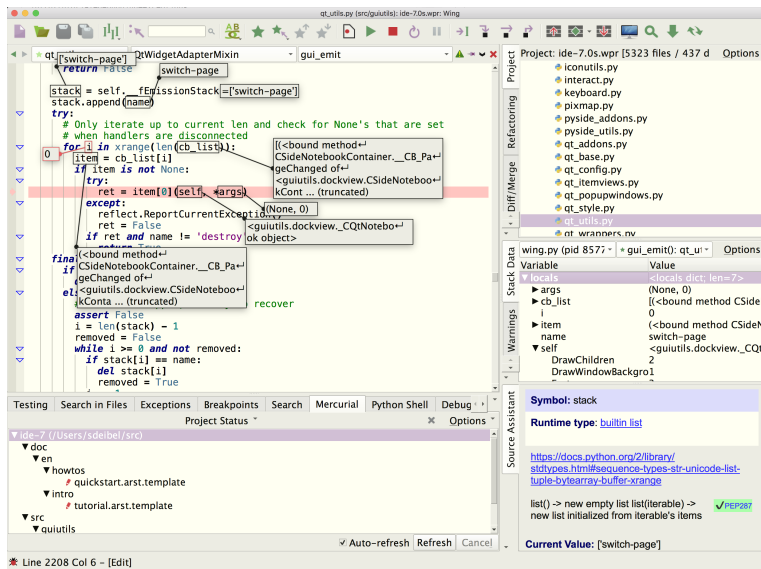
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running in virtualenv.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for virtualenv. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Creating a New Virtualenv

Wing Pro can create a new virtualenv at the same time that you create a new project. To do this, select **New Project** from the **Project** menu, choose the source directory to use with your new project, and then press **Next**. On the second page you will be able to select **Create New Environment** and choose **virtualenv** from the menu of available environment types.

If you are using an existing source directory, you will need to enter the following values:

Name is the name for your virtualenv directory.

Parent Directory is the directory where the virtualenv directory will be created.

If you are using a new source directory, the virtualenv will be created inside that new directory.

You may also specify the following values:

Packages to Install lets you specify packages to install into the new virtualenv. This is either a space-separated list of pip package specifications, or a file that contains one package specification per line. In either case, the package specifications may be anything accepted by pip, such as a package name, **package==version**, and **package>=version**.

Python Executable selects the base Python installation to use. In Python 2, you must install virtualenv into the selected Python first, if it's not already present.

Upgrade pip selects whether Wing should upgrade pip in the virtualenv before installing any packages, to compensate for the fact that virtualenv installs an old version of pip even if the base Python installation has a newer one.

Inherit global site-packages controls whether to use the **--system-site-packages** option when running virtualenv. When checked, the virtualenv will be able to use packages installed into the base Python installation. Otherwise, it will be completely isolated from the base install, other than its use of Python's standard libraries.

After submitting the **New Project** dialog, Wing will create the virtualenv, set the **Python Executable** in **Project Properties** to the command that activates the environment, and add the virtualenv directory to the project.

Now source analysis, executing, debugging, and testing in Wing will use the new virtualenv, as long as the project you just created is open.

Working on a Remote Host

Wing Pro can also create a new virtualenv on a remote host. This is done the same way as described above, except you first need to choose or create a remote host configuration from the **Host** menu on the first page of the **New Project** dialog.

If you are using a virtualenv on the remote host, Wing will update the remote host configuration to use that virtualenv.

Using an Existing Virtualenv

To use an existing virtualenv with Wing, simply set the **Python Executable** in Wing's **Project Properties** or the **New Project** dialog to **Activated Env** and enter the command that activates the environment. Wing uses this to determine the environment to use for source analysis and to execute, test, and debug your code. In this case, Wing starts Python by running **python** in that environment. This does not work, however, if the full path to the activate script contains a space. In that case, use **Command Line** instead, as described below.

Python Executable can also be set to **Command Line** to enter the full path to the virtualenv's **python.exe** or **python**. In fact, this is required if the full path to the activate command contains spaces. The easiest way to find the correct value to set is to launch your virtualenv Python outside of Wing and execute the following:

```
import sys
print(sys.executable)
```

Activating the Virtualenv

If you followed the above instructions, Wing will automatically activate the virtualenv while you're using your project.

An alternative approach is to leave **Python Executable** unset and instead activate the virtualenv on the command line and then start Wing from the command line so that it inherits the virtual environment. However, this is not recommended because the inherited environment may conflict with virtual environments used by other projects.

Package Management

Once you've configured your project to use a virtualenv, you can use the **Packages** tool in the **Tools** menu to list, add, remove, or update packages. See Package Manager for details.

Using Virtualenv with Anaconda

Anaconda implements its own named environments, created by **conda create** but it is also possible to use virtualenv with Anaconda. This works in the same way, except that on Windows Wing will automatically call **conda activate base** before it sets up your virtualenv. This is needed to avoid failure to import some modules as a result of missing environment. See About Anaconda Environments in the Anaconda How-To for details.

Related Documents

For more information see:

- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

1.2. Using Wing with pipenv

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running in a Python environment managed by pipenv.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for pipenv. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Creating a New Pipenv

Wing Pro can create a new pipenv at the same time that you create a new project. To do this, select **New Project** from the **Project** menu, choose the source directory to use with your new project, and then press **Next**.

On the second page you will be able to select **Create New Environment** and choose **pipenv** from the menu of available environment types.

Wing uses your selected source directory as the pipenv directory. You can optionally enter the following values:

Packages to Install lets you specify packages to install into the new virtualenv. This is either a space-separated list of pip package specifications, or a requirements.txt file that contains one package specification per line. In either case, the package specifications may be anything accepted by pip, such as a package name, **package==version**, and **package>=version**. An existing Pipfile or environment.yml may also be used to specify which packages to install.

Python Executable selects the base Python installation to use. You will need to install pip into this Python installation if not already present. Wing takes care of installing pipenv as needed.

After submitting the **New Project** dialog, Wing will create the pipenv, set the **Python Executable** in **Project Properties** to the command that activates the environment, and add the source directory to the project.

Now source analysis, executing, debugging, and testing in Wing will use the new pipenv, as long as the project you just created is open.

Using an Existing Pipenv

To use an existing pipenv with Wing, simply select the pipenv as your source directory using the **Use Existing Directory** option in the **New Project** dialog. Wing will automatically detect that this directory contains a pipenv and use it with your new project.

Working on a Remote Host

Wing Pro can also create a new pipenv on a remote host. This is done the same way as described above, except you first need to choose or create a remote host configuration from the **Host** menu on the first page of the **New Project** dialog.

If you are using a pipenv on the remote host, Wing will update the remote host configuration to use the virtualenv created by pipenv (rather than the base Python installation).

Package Management
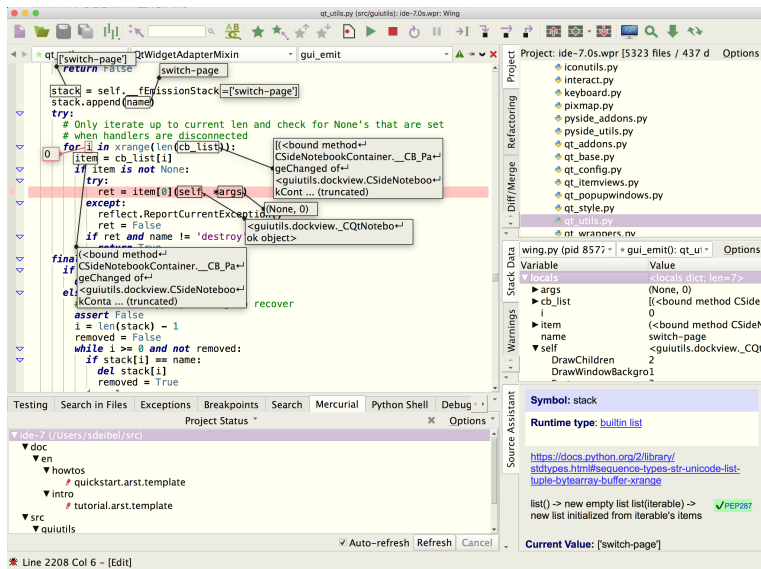
Once you've configured your project to use pipenv, you can use the **Packages** tool in the **Tools** menu to list, add, remove, or update packages. See Package Manager for details.

Related Documents

For more information see:

- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

1.3. Using Wing with Anaconda

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code run with the Anaconda Distribution of Python.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for use with Anaconda Python. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Configuring Your Project

To use Anaconda with an existing Wing project, simply set the **Python Executable** in **Project Properties** to the interpreter that you want to use. There are several options for this:

Command Line can be selected to enter the full path to Anaconda's **python.exe** or **python**. In many cases, Wing will automatically find Anaconda and include it in the drop down menu to the right of the file selector shown for this option. The Python executable for Anaconda is typically located at the top level of the installation on Windows and in the **bin** sub-directory on other OSes. Another way to find the correct full path to use is to start Anaconda outside of Wing and then type the following:

```
import sys
print(sys.executable)
```

Activated Env can be selected to use an existing environment created with **conda create** or virtualenv. This should be the command that activates the environment, for example **activate venv1**. In this case, Wing starts Python by running **python** in that environment. If Anaconda is installed in a default location, Wing will find your existing environments, which can be selected with the drop down menu to the right of this field.

Note that using **Activated Env** does not work if the full path to Anaconda's **activate** contains spaces. In that case, use **Command Line** option instead as described above.

If you are creating a new Wing project and want to use Anaconda, select **New Project** from the **Project** menu and configure **Python Executable** in the **New Project** dialog in the same way as described above. Note that you can create a new Anaconda environment from the **New Project** dialog by selecting **Create New Environment** on the second dialog page and choosing **Conda Env** from the dropdown menu.

In most cases, setting **Python Executable** is all that you need to do. Wing will start using your Anaconda installation immediately for source intelligence, for the next debug session, and in the integrated **Python Shell** after it is restarted from its **Options** menu.

Creating a New Anaconda Environment

Wing Pro can also create a new Anaconda environment with **conda create** at the same time that it creates a new project. To do this, select **New Project** from the **Project** menu, choose the source directory to use with your project, and press the **Next** button. Then select **Create New Environment** and choose **Conda Env** from the menu of available environment types.

You will need to enter the name for the new environment, choose the location to write the new environment, select the installation directory of the Anaconda that you want to use, and specify at least one package to install into the new environment.

Package specifications may either be entered directly into the **New Project** dialog, in a space-separate list, or read from an existing **requirements.txt** or **Pipfile**. In both cases, the package specifications may be anything accepted by **conda install** including just the package name, **package==version**, or **package>=version**:

```
flask
gunicorn
numpy==1.17.4
django>=3.1
```

When the **New Project** dialog is submitted, it will run **conda create** and then configure the project to use the new environment.

Package Management

Once you've configured your project to use an Anaconda environment, you can use the **Packages** tool in the **Tools** menu to list, add, remove, or update packages. See Package Manager for details.

About Anaconda Environments

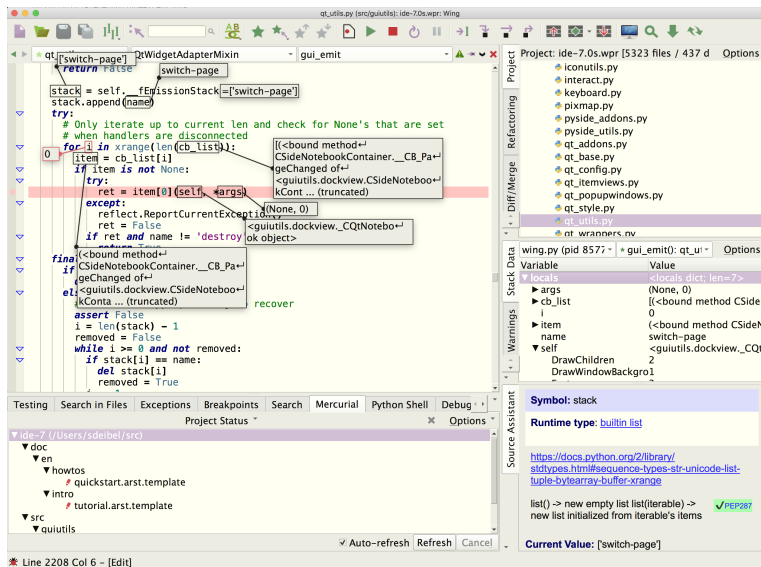On Windows, Anaconda may fail to load DLLs when its **python.exe** is run directly without using a named environment. This is due to the fact that by default the Anaconda installer no longer sets the **PATH** that it needs to run, in order to avoid conflicting with different Python installations on the same system. A typical error message looks like this:

```
builtins.ImportError:
IMPORTANT: PLEASE READ THIS FOR ADVICE ON HOW TO SOLVE THIS ISSUE!
Importing the numpy c-extensions failed.
...
Original error was: DLL load failed: The specified module could not be found.
```

The exact message you see will vary depending on which packages you are using, or you may not run into this at all if you are not using packages that are affected by it.

This may occur when running Anaconda Python outside of Wing without using a named Anaconda environment or when using virtualenv with Anaconda. The solution on the command line is to call **conda activate base** before starting Anaconda or activating the virtualenv.

The problem should not appear in Wing because it detects when Anaconda is being used and automatically activates the base environment before launching Anaconda.

Related Documents

For more information see:

- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

1.4. Using Wing Pro with Docker
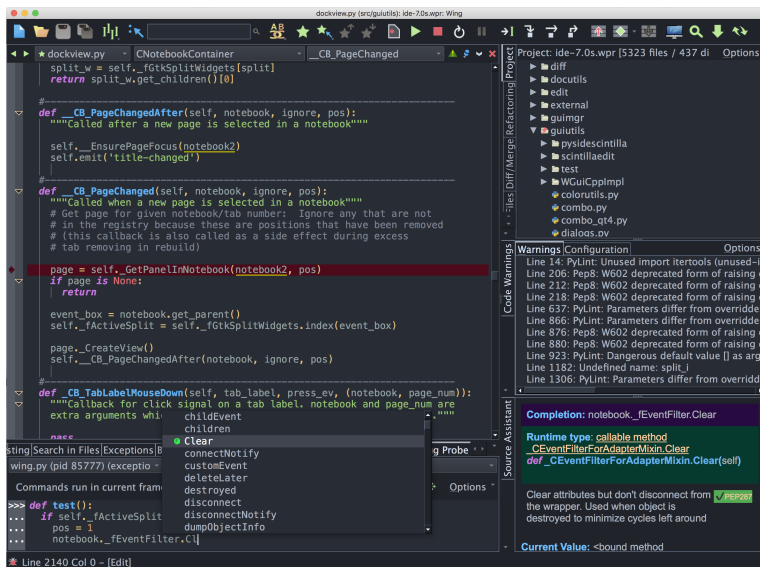
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running on Docker containers.

This document describes how to create a new Wing Pro project that uses a Docker container for Python development.

Getting Started

Before you can work with Docker you will need to download and install it.

On Windows and macOS, downloading Docker Desktop from the Docker website is the easiest way to install it. Be sure to launch the Docker Desktop after installation, so that the daemon is started.

On most Linux distributions, Docker CE (the free community edition) can be installed with the **docker-engine** package as described here.

You should also install Wing Pro if you don't already have it.

Overview of Docker

Docker provides a light-weight form of virtualization that serves to isolate running code from the host system, both for security and so that a replicable controlled environment can be created for developing and running applications.

Docker containers may be used to deploy code into production, and this may include spinning up new container instances on the fly to balance load or service incoming requests in a securely isolated environment.

Docker works by making use of standard preconfigured images that contain particular technologies pre-installed on either a Linux or Windows OS. These images are managed in a curated library and downloaded automatically, as needed to build your containers.

Each container may contain copies of files from the host system, either placed when the container is built or provided by file sharing as the container runs. Containers do not have any access to the host file system unless file sharing is configured, and then they can only write into the shared areas.

Containers can, however, access the host system through a network connection, which is what makes it possible to debug code running within a container from a copy of Wing Pro running on the host system. It is also possible to map network ports from the host system into the container, to facilitate development of network services and websites.

Containers are typically short-lived, persisting for the duration of one run of an application or for the purpose of performing a single computation. They are built, run, and then discarded after optionally retaining some state information or the result of the computation through a file share or network connection. Containers used in this way are often orchestrated into a larger system using Docker Compose or Kubernetes. This is how Docker can automate deploying, load-balancing, and management of containerized applications.

However, how to approach container life-cycle is up to the user. Other containers may be run many times, or over a long period of time, using data shared with the host system. Thus, containers may either be small ephemeral components of an application, or they may act more like a traditional virtual machine or server.

Configuration Overview

Wing Pro offers a number of ways to work with Docker containers:

(1) **Use An Existing Container** -- Wing Pro can configure a new project to use an existing Docker container. This approach requires specifying at least the container image to use and the host-container file mapping used by that container. Wing works with files stored on the local disk and launches debug processes, the Python Shell, unit tests, and optionally OS Commands within the container environment using the container's mapped copies of those files.

(2) **Create a New Container** -- Wing Pro can also create and configure a new Docker container. This approach creates the **Dockerfile** and a working starter application, configures a Wing Pro project file to use it, and automatically builds the container. This case also works with files on the local disk and runs debug and other processes on the container. When Wing creates a new Docker container, it uses the standard preconfigured 'python' container image with the selected Python version and packages installed by **pip**.

(3) **Remote Development via SSH** -- For containers that run OpenSSH and act more like a longer-lived virtual machine or server, Wing can connect to the container is if it were a remote host. This allows working directly with files on the container, rather than using local copies of the files.

These are each described in the following sub-sections of this How-To.

Related Documents

For more information see:

- Docker home page provides downloads and documentation.
- Working with Containers and Clusters for more information on using containers in Wing Pro.
- Quickstart Guide contains additional basic information about getting started with Wing Pro.
- Tutorial provides a gentler introduction to Wing Pro's features.
- Wing Pro Reference Manual documents Wing Pro in detail.

1.4.1. Using an Existing Docker Container with Wing Pro

Creating the Project

To create a new project that uses an existing Docker container, use **New Project** in the **Project** menu, select your source directory (or choose to create a new one), and press **Next**. On the second page, select **Create New Environment** and choose **Docker** from the drop down menu of environment types. Finally, select **With Existing Container** and enter at least the container image and host-to-container file mapping. The configuration options are:

Configuration

Select the style of configuration to use: Either an existing already-built Docker image ID, or an existing **Dockerfile**.

Image ID

Enter the name of the Docker image to use. When **Configuration** is set to **Use Image ID** this selects from those images that have already been built by Docker. When **Configuration** is instead **Specify Configuration** this should be the image name defined with **FROM ... AS** in the Dockerfile or any valid image name to use for the configuration if **AS** is not used. Existing images are listed in the drop down to the right of the entry field.

File Mappings

You must enter at least one file mapping. Each mapping is a pair of directories, one on the local host and the other on the container. This is the mapping set up for the selected container image with the **-v** command line option for **docker run** or using **COPY** in the **Dockerfile**.

Mapping Type

This specifies whether the file mappings are made by using **-v** with **docker run** or with **COPY** in the **Dockerfile**. In general it's easier to use **-v** because the container does not need to be rebuilt and restarted when host-side files change.

Even when the mapping type is set to use **COPY**, Wing will establish an internally defined dynamic mapping with **-v**, in order to make the debugger and other IDE functionality available on the container.

Python Executable

This selects the Python to run on the container. In most cases this is the default, which **python3** if it exists or otherwise **python**.

Connect Hostname

This is the host name that the container uses to make a TCP/IP connection to the host system. In most cases, this is **host.docker.internal**. On Linux, where **host** networking mode is used, set this to **127.0.0.1** instead (see Networking on Linux Hosts below for details).

How It Works

Wing takes the configuration values you enter in the **New Project** dialog and configures a new Wing project as follows:

1) A new container configuration is created and set as the **Python Executable** to use in **Project Properties**.

> 2. The mapped directories are added to the project.
> 3. The **Python Shell** is restarted using Python running in the container.

If you didn't create a new source directory along with your project then you will need to save the project file with **Save Project** in the **Project** menu. Otherwise, it is saved automatically into the new source directory.

Now Wing will run debug processes, unit tests, and the **Python Shell** on an instance of the selected container image. **OS Commands** can optionally run commands on the container or on the local host. This is selected by toggling the **Run in Container** option in the OS Command configuration.

Note that each debug process, each unit test run, and each instance of the **Python Shell** creates its own independent instance of the container.

Networking on Linux Hosts

When Docker is running on a Linux host, it starts containers using a different networking mode than the one used by Docker Desktop on Windows and macOS. This mode prevents connections from the container to the host system, which prevents Wing from inspecting and debugging the containers.

To work around this problem, Wing starts containers on Linux using **host** networking mode. In this mode, the container can connect to the host using **127.0.0.1**.

Currently there is no way to override this behavior. Please contact us at support@wingware.com if this does not work with your container.

1.4.2. Creating a New Docker Container with Wing Pro

To create a new Docker container along with your new project, use **New Project** in the **Project** menu, select your source directory (or choose to create a new one), and press **Next**. On the second page, select **Create New Environment** and choose **Docker** from the drop down menu of environment types. Finally, select **Create New Container** and enter at least the container image and host-to-container file mapping. The configuration options are:

Image ID

This selects the image ID name to assign to the docker container. It is used within Wing to refer to the container configuration, by Docker to refer to the container, and is used as the default new project directory.

Host-side Project Directory

This selects the directory where the new project should be written on the host system. The directory will contain the generated **Dockerfile** and **requirements.txt**, as well as the source directory and other support files.

Container-side Directory

This selects the location on the container where the source directory should be mapped or copied. The source directory on the host side is always called **app** and is created at the top level of your selected **Host-side Project Directory**.

Packages

This specifies any PyPI packages that should be installed into the new container, either as a space-separated list of package specifications, or a file that contains one package specification per line. In either case, the package specifications can be a package name or **package:version** to specify the version to use.

Wing will create **requirements.txt** that is stored with the new project and used to make sure the packages are installed when the container is built. You will be able to add or change packages on the container later by editing the **requirements.txt** file and then rebuild the container from the **OS Commands** tool.

> ### *Note*
>
> Be sure to read about Package Security.

Establish Mappings

When this option is checked (the recommended default), Wing configures the Docker container so that it is run with your host-side source directory mapped into the container with the **-v** option passed to **docker run**. This acts like a file share, so that changes made on the host file system

immediately become available within the container, and changes made within the container are written back to the host file system.

When establishing mappings is disabled, Wing instead uses the **COPY** directive in your **Dockerfile** to make a copy of your host-side source directory when the Docker container is built. In this case, the container needs to be rebuilt and restarted whenever a change is made to your sources. To handle this, Wing sets up the project it creates to automatically run the build each time code is debugged or executed.

Regardless of hether this is enabled, Wing always establishes an internally defined dynamic mapping, in order to make the debugger and other IDE functionality available on the container.

Use wingdbstub for Debugging

When this option is checked, Wing configures the container so that you can initiate debug from your code using **import wingdbstub**, rather than starting debug from the IDE. In most cases, this option should be left disabled. It is useful when Python cannot be started directly in the container.

Python Version Specifier

This option specifies which Python version Wing should install into the Docker container. This is placed after the **FROM: python** directive in the **Dockerfile** that Wing generates. It can be any of the Python image variants that Docker supports, including **3** for Python 3, **2** for Python 2, or a specific version like **3.8.1**. If this option is left blank, the latest Python version at build time will be used. To make your development environment reproducible and uniform, it is best to select a specific Python version.

How it Works

Once you've created your project, Wing creates the selected host-side directory and populates it with the following items:

(1) An **app** directory is created as the place to put source code for the application. This directory is either copied into or shared with the container, depending on whether the **Establish Mappings** option was checked. The directory initially contains a **requirements.txt** file that specifies which third party packages are needed by your application. This is passed to **pip** during the container build process. If it is edited, the container will need to be rebuilt (see below). The **app** directory also a starter code file. If the **Use wingdbstub for Debugging** option was selected, the file **wingdbstub.py** and security token **wingdebugpw** will also be placed here and configured for access to Wing Pro running on the host.

(2) A **Dockerfile** is created that tells Docker to use the standard Python container image and the contents of **app/requirements.txt** to set up the container's environment. The default action of the container, used when **docker run** is invoked without arguments, is to run the application without debug.

(3) The script **build.sh** (or **build.bat** on Windows) is written to build or rebuild the container.

Wing also creates and saves a project file into the host-side directory that is configured to make use of the container. The host-side directory is added to the **Project** tool, a mapping is established

between files on the container and host side so that Wing knows where files are located on each, the **Python Executable** is set up to run the Docker-provided Python, and the project is configured to run the build command automatically before debugging if not using a dynamic mapping to share files.

Wing then builds the container in the **Containers** tool and when that completes saves the project file to disk and restarts the **Python Shell** to contain the Docker-provided Python. Building the container may take a while if you haven't yet downloaded the container image or are installing many packages.

Once this completes, you can start debug from the **Debug** menu or toolbar. Wing should reach a breakpoint it has set automatically in the starter code configured with the project. If you opted to use **wingdbstub** to initiate debug, you will need to instead start the process from outside of Wing or by using **Execute Current File** in the **Debug** menu. Wing will have added **import wingdbstub** to the starter code it generated, so that debug should start and a breakpoint should be reached.

Whenever **requirements.txt** or the **Dockerfile** are changed, you will need to rebuild your container, which can be done by right-clicking on the **Containers** tool.

1.4.3. Remote Development via SSH to a Docker Instance

Docker is sometimes used to host a longer-lived installation of Linux, acting more like a stand-alone system that includes the ability to **ssh** from the host system into the container. In this case, Wing Pro's Remote Development capability can be used instead, to work directly with files and processes running under Docker (no local copies of the files need to exist).

Setting up remote development via SSH to a Docker container works the same as for any other type of remote host. For details, see Remote Python Development (if the debug process can be launched from the IDE) or Remote Web Development (if the debug process is launched from outside of the IDE).

For more information on setting up SSH access into a Docker container, see SSH into a Docker Container.

1.4.4. Docker Configuration Example

If you are new to Docker, setting up a simple container manually may help to clarify how Docker works. This can be done by creating a directory **docker** and placing the following files into it.

Dockerfile:

```
FROM python:3.7
WORKDIR /app
RUN pip install --trusted-host pypi.python.org Flask
EXPOSE 80
CMD ["python", "app.py"]
```

app.py:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "<h3>Hello World!</h3>Your app is working.<br/></br/>"

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, use_reloader=True)
```

Then build the Docker container by typing the following in the **docker** directory:

```
docker build --tag=myapp .
```

You can now run your container like this:

```
docker run -v "/path/to/docker":/app -p 4000:80 myapp
```

You will need to substitute **/path/to/docker** with the path to the **docker** directory you created above; the quotes make it work if the path has spaces in it.

You can now try this tiny Flask web app by pointing a browser running on your host system at it:

If you are using Docker Desktop, then use http://localhost:4000/

If you are using Docker CE, you will need to determine the IP address of your container and use that instead of **localhost**. One way to do this is to type **docker ps** to find the Container ID for your container and then use it in the following in place of **c052478b0f8a**:

```
docker inspect -f "{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}" c052478b0f8a
```

Notice that if you make a change to **app.py** in Wing, then the change will be reflected in your browser when you reload the page. This is due to using both the **-v** argument for **docker run** to mount a volume in the container (so container files are immediately updated if they change on the host), and the fact that **app.run()** for Flask is being passed **use_reloader=True** (so that changed container files are automatically reloaded by Flask).

1.5. Using Wing Pro with Docker Compose
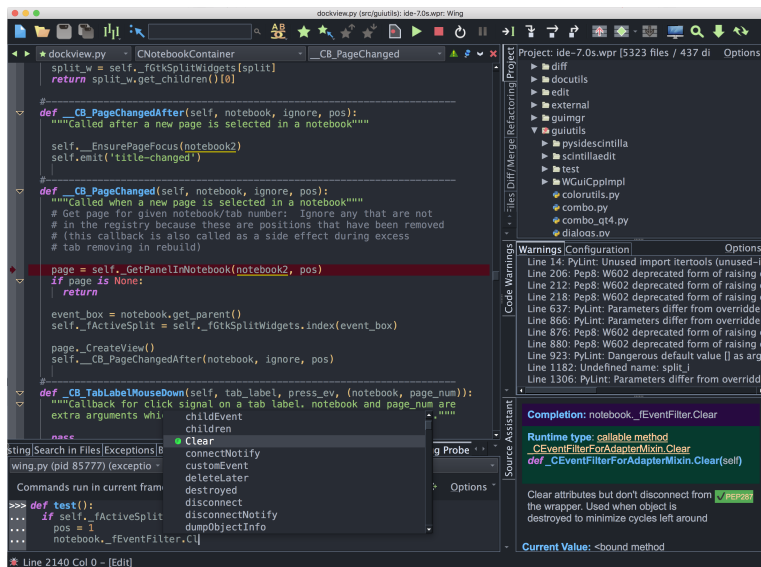
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running on clusters that are orchestrated by Docker Compose.

This document describes how configure Wing Pro in order to use Docker Compose for Python development.

Getting Started

Before you can work with Docker Compose you will need to download and install it. See Install Docker Compose for details.

You should also install Wing Pro if you don't already have it.

Configuration Overview

To configure Wing to use a Docker Compose cluster, select **Cluster** for the **Python Executable**, either in the **New Project** dialog or **Project Properties** (both from the **Project** menu).

In the **New Project** dialog, this is found under **Use Existing Python** on the second dialog page, after you have selected your source directory.

In the **Project Properties** dialog, it is under the **Environment** tab.

After selecting **Cluster** for **Python Executable**, create a new cluster configuration by pressing the **New** button. This displays the cluster configuration dialog. You will need to enter an identifier to use within Wing and point it at the **docker-compose.yml** file for the cluster.

You will also need to select the main service to use as the default place to run your **Python Shell** and unit test processes. Note, however, that both the shell and tests can be configured to run either in synthesized out-of-cluster copies of a cluster service, or on specific services after the cluster as a whole is launched. This is described in more detail in Execution Context below.

Once you have created your cluster configuration, submit the **New Project** or **Project Properties** dialog to complete your project setup.

Controlling the Cluster

You can now control your cluster from Wing's **Containers** tool, found in the **Tools** menu. The top part of this tool shows the overall status of the cluster and provides buttons for building, starting, debugging, and stopping your cluster.

The **Containers** tool also lists the services in the cluster and their status. You can right-click on items in the services list to view their configuration in the **docker-compose.yml** file.

When **Show Synthesized Containers** is checked in the tool's **Options** menu, the list of services will include out-of-cluster containers that Wing synthesizes and starts, for example to run the **Python Shell** or unit tests.

When the **Show Console** item in the **Options** menu is checked, the bottom of the tool shows a console containing the output of the docker-compose commands that Wing is running behind the scenes.

Debugging the Cluster

There are several ways to debug code in a cluster managed by Docker Compose:

(1) The most common way to debug a cluster is to select the services that should be debugged in the services list within the **Containers** tool. Then press the **Debug** button. Wing launches the cluster in such a way that all Python code run in the selected services will be debugged.

(2) Another way to debug code in a cluster is to first start the cluster as a whole without debug by pressing **Start** in the **Containers** tool. Additional processes can be then started and debugged in-cluster by setting up a **Named Entry Point** from the **Debug** menu and configuring those to launch code in-cluster. This is done by setting the **Python Executable** in the associated launch configuration to **Cluster**, selecting the cluster configuration and service, and checking the **In-Cluster** checkbox. See Named Entry Points for more information.

(3) It is also possible to debug code in a synthesized out-of-cluster copy of a container. This is useful if the container environment as a whole is not needed by that code, and can be accomplished in the same way as in option (2) above, but by unchecking the **In-Cluster** checkbox.

Execution Context for Other Processes

Other processes started by Wing may also be run either on a synthesized out-of-cluster copy of a selected container service, or on the live service in-cluster after the cluster as a whole has been started.

Python Shell

By default, Wing starts the **Python Shell** on a synthesized out-of-cluster copy of the service you selected as you main service during cluster configuration. This can be changed with **Use Environment** in the Python Shell's **Options** menu. From here, a launch configuration may be defined that selects **Cluster** for **Python Executable** and checks on the **In-Cluster** checkbox.

Unit Tests

Unit tests work in a similar way: The testing environment can be set up with **Environment** under the **Testing** tab in **Project Properties**, or in the properties set by right-clicking on an individual test file.

OS Commands

The **OS Commands** tool can also run commands either out-of-cluster or in-cluster. For command lines, this is done by setting the **Execution Context** under the **Environment** tab of the OS Command configuration. For Python files, the environment set in the file's properties is used, or you can define a **Named Entry Point** that pairs a Python file with the desired launch configuration.

For more information on using clusters in Wing, see Working with Clusters.

How it Works

Whole-cluster debug is implemented by creating a derived **docker-compose.yml** file that mounts Wing's debugger into the container, and also a **site-packages/sitecustomize** directory that imports **wingdbstub** in order to initiate debug back to the IDE as soon as Python starts.

Wing also takes care of all the associated configuration for the debugger, including establishing the network connection and mapping between local and container side copies of source files.

Related Documents

For more information see:

- Docker home page provides downloads and documentation.
- Working with Clusters for more information on using clusters with Wing Pro.
- Quickstart Guide contains additional basic information about getting started with Wing Pro.
- Tutorial provides a gentler introduction to Wing Pro's features.
- Wing Pro Reference Manual documents Wing Pro in detail.

1.6. Using Wing Pro with LXC/LXD Containers

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running on LXC Linux containers hosted by LXD.

This document describes how to create a new Wing Pro project for Python development on a Linux container.

Getting Started

Before you can work with Linux containers in Wing, you will need to install and configure LXD.

You should also install Wing Pro on your host system if you don't already have it.

Overview of LXC/LXD

LXC implements Linux containers, which provide a light-weight form of virtualization that is useful for its security, standardization of runtime environment, and ability to support scalable deployment. LXD is the next-generation toolset for controlling LXC containers.

LXC/LXD use standard preconfigured container images that are managed in a curated library and downloaded automatically, as needed to run your containers. Each container may contain copies of files from the host system, either copied into place or provided by file sharing as the container runs. Containers do not have any access to the host file system unless file sharing is configured, and then they can only write into the shared areas.

Some LXC/LXD containers, depending on configuration, can access the host system and/or other containers in a cluster through a network connection. It is also possible to map network ports from the host system into the container, to facilitate development of network services and websites.

Wing Pro supports only the newer LXD command line interface. It requires that the container has network access to the host system, which is enabled by default when a container is created. The container must also have Python installed on it, which most Linux systems do.

Creating a Container

You will first need to create a container instance, as follows:

```
lxc launch ubuntu:20.04 demo
```

You can replace **ubuntu:20.04** with any image that includes Python. Type the following to see a list of all available images:

```
lxc image list images:
```

The name **demo** may also be replaced, with any other name for the new container instance.

Configuring Your Project

To set up a Wing project that uses your LXD container, select **New Project** from the **Project** menu, select **Create Blank Project**, and press the **Create Project** button.

After your project has been created, select **Project Properties** from the confirmation dialog or **Project** menu. Then set **Python Executable** to **Container** and press **New** to create a new container configuration. Enter at least the following fields:

- **Identifier** is any short name for the container instance. It does not have to be the same as used in **lxc launch** above, and is used only within Wing to identify the container.
- **Type** should be set to **LXC/LXD**.
- **Image ID** is the image name you gave in your **lxc launch** call (**demo** in the example above).

You may also need to set **Python Executable** under the **Options** tab of the container configuration, if there is no executable called **python** on the container. This is needed on Ubuntu 20.04 (the image used in the example above) because only **python3** exists there. If you used that image, select **Command Line** and enter **python3**.

Once you press **OK** in the new container configuration dialog, the new container will be entered into the **Project Properties** dialog and you can submit that to finish project configuration.

Now is a good time to save your project to disk from the new project confirmation dialog or using **Save Project** in the **Project** menu.

Testing the Container

At this point, Wing should start up the container automatically. The **Python Shell** tool in Wing, available in the **Tools** menu, will also restart using the new configuration, so you can interact with the Python installation on the container.

The status of the container can be seen in the **Containers** tool from the **Tools** menu. It can be restarted by right-clicking in the list of containers found there.

Developing Code

Now you have a working container integration but there is not yet any Python code that can be run on the container. To add that, use **New** in the **File** menu, and enter a simple test file as follows:

```
print('Hello World')
```

Then create a directory on your host system and save the file there as **test.py**.

Next, add a file mapping to your container configuration by selecting **Containers** from the **Project** menu, editing the container, and adding an entry to the **File Mappings** list. The **Host** entry is the full path to the directory where your just wrote your **test.py** and **Container** is the full path where you want to mount that directory on the container. For example:

- **Host:** /home/testuser/demo
- **Container:** /app

Note that if your container already has a mapping set up in its configuration, you will need to uncheck **Establish Mappings** under the **File Mappings** field to prevent Wing from trying to create the mappings when the container is launched. Then that is unchecked, the mappings listed here are used only to determine which files on the host match files on the container. When it is checked, Wing also sets up the mappings for you by temporarily modifying the container's configuration. For the above simple demo, this option needs to remain checked.

When you save your container configuration, Wing will restart the container and your code is now available on the container.

Wing automatically takes care of mapping to and from the container's location for your source files, and knows that code should be launched on the container (and not the host system) when debugged or executed because your **Project Properties** selected the container for **Python Executable**.

To debug it, set a breakpoint in your code by clicking in the leftmost margin in the editor.

Then press the green play icon in the toolbar in Wing or use **Start/Continue** in the **Debug** menu. You should reach the breakpoint. Continuing from there causes "Hello World" to appear in Wing's **Debug I/O** tool and the debug process will exit.

That's all there is to it! You can now develop, debug, execute, and test your Python code in the LXD container environment.

Related Documents

For more information see:

- LXD provides more information documentation for LXD and LXC.
- Working with Containers and Clusters for more information on using containers in Wing Pro.
- Quickstart Guide contains additional basic information about getting started with Wing Pro.
- Tutorial provides a gentler introduction to Wing Pro's features.

- Wing Pro Reference Manual documents Wing Pro in detail.

1.7. Using Wing Pro with AWS
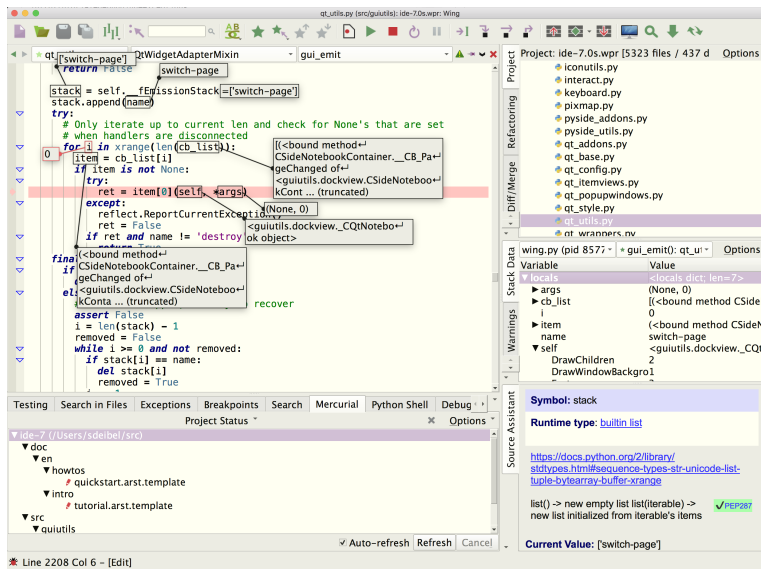
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running on Amazon Web Services (AWS).

This document describes how to configure Wing Pro for AWS. To get started using Wing Pro as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Prerequisites

You will need an AWS instance that has Python installed on it, is running 32-bit or 64-bit Intel Linux, and that you can connect to with SSH. You will need the following in order to configure Wing Pro to use your AWS instance:

(1) The user name and static IP address or DNS name used to connect to the AWS instance. Amazon Lightsail typically uses an IP address while EC2 and other variants of AWS may provide a DNS name as well. The static IP address will work in any case.

(2) The SSH key pair in a ***.pem** file, as downloaded from AWS.

If you do not already have Wing Pro installed, download it now.

Setting up AWS

If you already have an AWS instance to work with, you can skip this section.

Otherwise, Amazon Lightsail is the easiest way to get an AWS instance, in about 5 minutes. You will need to set up an account. Then create a **Linux/Unix** instance, selecting the **OS Only** option and the most recent **Ubuntu** or any other Intel Linux with Python 2 or 3 on it.

While setting up your instance, you can download your SSH key pair under the AWS **SSH key pair manager**. You'll need this on your local machine, where Wing is running, in order to be able to connect to the instance.

After the instance is created, it will remain in **pending** state for a minute or so. Once it is up and running, create a static IP address under the **Network** tab in the AWS Lightsail management area and attach it to your instance.

At this point you have all that is needed to start using Wing Pro with AWS: (1) The SSH key pair that you downloaded, and (2) the user name and IP address, which are shown on the Lightsail instance management page.

Testing the SSH Connection

Before trying to use your new instance from Wing Pro, you may want to first try to connect using **ssh** or PuTTY's **plink.exe** on the command line, to make sure your configuration is working. This is not a requirement, however, since you can also connect to AWS using Wing's built-in SSH implementation.

OpenSSH

On Linux or macOS using **ssh**, you need to make your ***.pem** SSH key pair file readable only by the user running Wing, for example with:

```
chmod 600 aws.pem
```

Otherwise, **ssh** will reject it as potentially compromised.

Once that is done, try connecting as follows, substituting the actual path to your downloaded SSH key pair and your instance's username and IP address or DNS name:

```
ssh -i /path/to/mykey.pem ubuntu@11.22.33.44
```

You will be asked to add the instance's identity to your known hosts file, which you should do by typing **yes**. If this is not done, **ssh** will fail to connect and Wing will also not be able to connect to the instance.

PuTTY

With PuTTY on Windows, you will need to first convert the SSH key to a format that PuTTY can use. This is done by launching **puttygen**, pressing the **Load** button to read the ***.pem** SSH key file you downloaded from the AWS management site, and then using **Save Private Key** to write a ***.ppk** file.

Then you invoke **plink.exe** to connect to the AWS instance as follows, substituting in the actual path to your downloaded SSH key pair and the correct username and IP address or DNS name for the AWS instance:

```
plink.exe -i C:\path\to\mykey.ppk ubuntu@11.22.33.44
```

You will be asked to accept the AWS instance's identity the first time you connect, and this must be done before Wing's remote development support will work with the AWS instance.

Built-in SSH Implementation

If you don't have **ssh** or PuTTY, you can also just use Wing's built-in SSH implementation. You will need to know the full path to your AWS **.pem** private key file and also the username and host's IP address. Once you have these, proceed to the next section to configure your connection to AWS in Wing.

Creating a Wing Project

Now you're ready to create a project in Wing Pro. This is done with **New Project** from the **Project** menu. From the **Host** menu at the top of the first page in the **New Project** dialog, select **Create Configuration**.

This displays the **New Remote Host** dialog where you can configure Wing to connect to AWS. Select **AWS** for the **Remote Host Type**. Then enter an identifier for the remote host (any short string to identify it in Wing's UI) and the user name and IP address or DNS name used to connect to the AWS instance.

In most cases **Python Executable** should be **Use default**, which first tries **python3** and then **python**. If Python is not on the **PATH** on your AWS instance or you want to specify a particular Python executable or activate a virtual environment, you can do this here.

You will also need to point Wing at the SSH key file you downloaded from AWS earlier. This is done under the **Options** tab of the **New Project** dialog, using the **Private Key** field. Select **Use private key file** and enter the full path to your downloaded SSH key.

Pressing **OK** in the **New Remote Host** dialog will create the remote host configuration, so Wing can already connect to your AWS instance. Form here you can select or create your source directory on the AWS instance, and choose or create the Python environment to use. See Creating a Project for details.

After pressing **Create Project** in the **New Project** dialog, Wing will configured a new untitled project. You can save it to local disk from the **Project** menu.

Testing a Hello World

To try out a simple example of editing and debugging code on the remote AWS instance, create a file **helloworld.py** temporarily on the instance. This is done by right-clicking on one of the directories in the **Project** tool in Wing Pro and selecting **Create New File**. Enter the file name (in some key bindings this is in the data entry area at the bottom of Wing's window) and then type or paste the following into the new file:

```
import time
print("Hello World! {}".format(time.time()))
```

After saving the file, set a breakpoint on the second line by clicking on the leftmost margin in the editor. Then select **Start/Continue** from the **Debug** menu to start debug, or use the green play icon ▶ in the toolbar.

There is a slight delay to get the process started, depending on your network distance from the AWS instance, but then you should see Wing stop on the breakpoint. Although there's not much to

see in this context, you can check that it's working by entering the following into the **Debug Console** tool:

```
time.time()
```

Once you continue debugging, the process will exit and print its "Hello World" message to Wing's **Debug I/O** tool.

Related Documents

For more information see:

- Amazon Web Services (AWS) provides documentation and links for creating an AWS account and instance.
- Remote Hosts for details on configuring remote development.
- Quickstart Guide contains additional basic information about getting started with Wing Pro.
- Tutorial provides a gentler introduction to Wing Pro's features.
- Wing Pro Reference Manual documents Wing Pro in detail.

1.8. Using Wing with Vagrant

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running on Vagrant containers.

This document describes how to configure Wing Pro for Vagrant. To get started using Wing Pro as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Prerequisites

This guide assumes you have already installed and started using Vagrant. Wing Pro supports working with Vagrant boxes running Linux (32-bit or 64-bit) or macOS. Other OS types, such as FreeBSD, won't work.

You will also need to make sure that Python is installed in the Vagrant box. If Python is not present, Wing Pro's remote agent installation process will fail.

If you do not already have Wing Pro installed, download it now.

Creating a Project

Creating a Wing Pro project for an existing Vagrant container is easy:

1. Start your container with **vagrant up**
2. Use **New Project** from the **Project** menu and then select **Create Configuration** from the **Host** menu in the **New Project** dialog. Then select **Vagrant** as the **Remote Host Type**
3. Fill in the full path to your Vagrant project directory (which contains your **Vagrantfile**) and press **OK** to create the remote host configuration
4. Only on Windows with PuTTY, you will be asked to convert Vagrant's private key into a PuTTY key. To do this, Wing starts **puttygen** with the private key already loaded into it. Press **Save private key** to save the key as **private_key.ppk** in the current directory. Confirm saving

without password (the original also doesn't have a password) and then quit **puttygen** to continue the project setup process in Wing.

Once this is done, Wing should install the remote agent and confirm that it is working. You can now continue through the **New Project** dialog to select or create your source directory on your Vagrant container and select or create the Python enviroment to use. See Creating a Project for details.

To learn more about Wing Pro's remote development capabilities, see Remote Hosts.

To learn more about Wing Pro's features, take a look at the Tutorial in the **Help** menu or the Quickstart Guide.

How It Works

Wing uses **vagrant ssh-config** to inspect your Vagrant container and fill in the necessary settings in Wing's project file.

To see the settings that Wing created during **New Project**, take a look at **Project Properties** from the **Project** menu. The **Python Executable** was set to point to a remote host named **vagrant**. Click on **Edit** here or use **Remote Hosts** in the **Project** menu to access the remote host configuration. The values that Wing sets up are: **Identifier** and **Hostname** under the **General** tab, **SSH Port** and **Private Key** under the **Options** tab, and **Manage SSH Tunnels** under the **Advanced** tqb. Settings these values manually achieves exactly the same results as using the **New Project** dialog.

Usage Hints

Synced Folders

As far as Wing is concerned, all files and directories are located in the Vagrant container and Wing never accesses local copies of the files maintained by Vagrant's synchronization commands.

If you need to update your local copies of files for some other reason while working with Wing, run **vagrant rsync**, or set up continuous synchronization with **vagrant rsync-auto**.

Password-less Private Keys

Vagrant uses password-less private keys by default. However, Wing can also work with password-protected private keys. You will be prompted as needed for the passphrase to unlock your key. Alternatively, you can load the key into the SSH user agent (**ssh-agent** or **pageant** for PuTTY) and change **Options > Private Key** in Wing's **vagrant** remote host configuration to **Use SSH User Agent**.

Related Documents
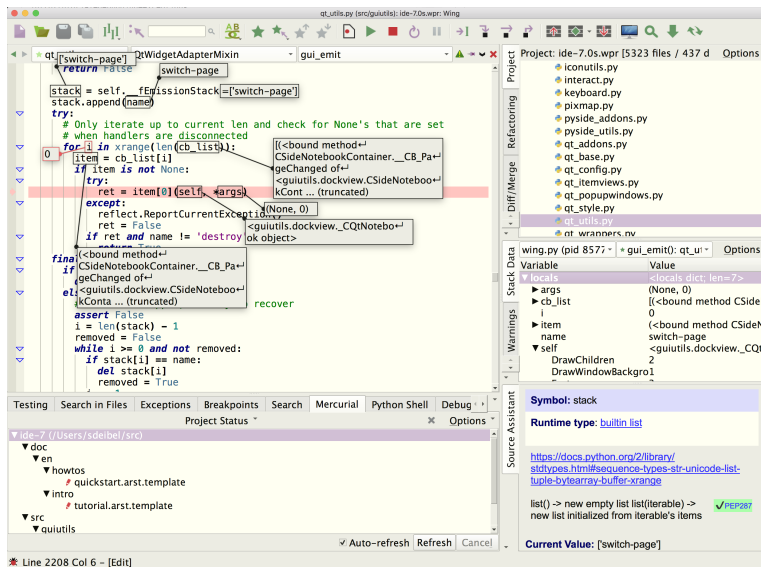
For more information see:

- Vagrant home page provides downloads and documentation.
- Remote Python Development describes how to set up remote development in general.
- Remote Development documents the details of remote development.
- Quickstart Guide contains additional basic information about getting started with Wing Pro.

- Tutorial provides a gentler introduction to Wing Pro's features.
- Wing Pro Reference Manual documents Wing Pro in detail.

1.9. Using Wing Pro with Windows Subsystem for Linux

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running on Windows Subsystem for Linux (WSL), including both WSL 1 and WSL 2.

This document describes how to configure Wing Pro for WSL. To get started using Wing Pro as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Prerequisites

To use Wing Pro with WSL you will need to install Python on WSL and be able to SSH from Windows into WSL. If you don't already have this working, see **Setting up WSL** below before trying to create a project in Wing Pro.

If you do not already have Wing Pro installed, download it now.

Creating a Project

To create a Wing Pro project that accesses Linux under WSL:

1. Use **New Project** from the **Project** menu and then select **Create New Configuration** from the **Host** menu in the **New Project** dialog.
2. Leave the **Remote Host Type** in the **New Remote Host** dialog set to **Generic**.
3. Set **Identifier** to **wsl** or some other short name for the WSL-hosted Linux.
4. Set **Host Name** to **username@ipaddress** where **username** is replaced with the user name running on Linux and **ipaddress** is replaced with the IP address that **ip a** reports inside your WSL system. The user name is needed even if it is the same as the user running on Windows.
5. Set **Python Executable** to **Command Line** and enter **/usr/bin/python3** or the full path to the Python executable you wish to use on Linux.
6. If you are running the SSH server on Linux under a non-standard port, set **SSH Port** under the **Options** tab.

7. If you are using an older WSL version that allows connecting with **username@127.0.0.1** then you will also need to set **Manage SSH Tunnels** under the **Advanced** tab of the remote host configuration to **Never Create Tunnel**. This should *not* be done if using an IP address other than **127.0.0.1** in the **Host Name** field.

8. Leave the rest of the configuration values set to their defaults and press **OK**.

Wing Pro should install the remote agent and confirm that it is working. Then you can continue through the **New Project** dialog to select or create your source directory and select or create the Python environment to use. See Creating a Project for details.

You will be able to edit, debug, test, search, and manage files on the WSL-hosted Linux installation, or launch commands running on Linux from Wing Pro's **OS Commands** tool.

To learn more about Wing Pro's remote development capabilities, see Remote Hosts.

To learn more about Wing Pro's features, take a look at the Tutorial in Wing's **Help** menu or the Quickstart Guide.

Setting up WSL

Here is one way to set up WSL with Ubuntu as the Linux distribution and PuTTY as the SSH client:

Enable WSL and Install Ubuntu Linux:

- Enable WSL in Windows 10+. This is done in the Settings app under Apps > Apps & features > Related settings / Programs and features > Turn Windows features on and off. Restart when prompted.
- Install Ubuntu from the Microsoft Store.
- Install Python in Ubuntu with **sudo apt-get install python3** if not already present.

Fix the SSH server on Ubuntu:

- Some versions of Ubuntu under WSL seem to be initially misconfigured so that connecting to the SSH server immediately drops the connection. If you run into this, you can fix the problem with **sudo apt-get purge openssh-server** followed by **sudo apt-get install openssh-server** and then **sudo service ssh --full-restart**. Type **wsl** on Windows to get into your WSL system without needing the SSH server to work.

Each time you restart Windows or Ubuntu:

- Run **sudo service ssh --full-restart** on Ubuntu to make sure the SSH server is started.

Find the IP Address for Ubuntu

- Run **wsl** on Windows
- In the resulting WSL prompt, run **ip a** to get the IP address for **eth0**. For example, in the following output fragment the IP address is **172.17.134.127** (the number right after **inet** in the **eth0** section):

```
4: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
  link/ether 00:15:5d:a8:d9:33 brd ff:ff:ff:ff:ff:ff
  inet 172.17.134.127/20 brd 172.17.143.255 scope global eth0
    valid_lft forever preferred_lft forever
```

Now you should be able to connect to Ubuntu using **username@ipaddress**, with **username** replaced with your Ubuntu-side user name and **ipaddress** replaced by the IP address you just found. You can try this with PuTTY's **plink.exe** or OpenSSH's **ssh.exe** if you have those, or proceed to creating your Wing project (see above) to use Wing's built-in SSH implementation.

Notes

In the unlikely event that you already have an SSH server running somewhere on your system on port 22, you will need to change the port number used on Ubuntu in **/etc/ssh/sshd_config** and then execute **sudo service ssh --full-restart** again. In this case, you also need to set the port in your remote host configuration in Wing, under the **Options** tab.

Other Linux distributions are also available in the Microsoft Store.

It is also possible to use Open SSH instead of PuTTY. See Working with OpenSSH for details.

Related Documents

For more information see:

- Windows Subsystem for Linux provides information on getting starting with WSL.
- Remote Hosts for details on configuring remote development.
- Quickstart Guide contains additional basic information about getting started with Wing Pro.
- Tutorial provides a gentler introduction to Wing Pro's features.
- Wing Pro Reference Manual documents Wing Pro in detail.

1.10. Using Wing with Raspberry Pi

> **Note**
>
> **"Within a couple of minutes I could fence in and eliminate an error with the handling of a GPRS modem attached to the Raspberry Pi that before I was trying to hunt down for hours."** -- *Robert Rottermann*, redCOR AG
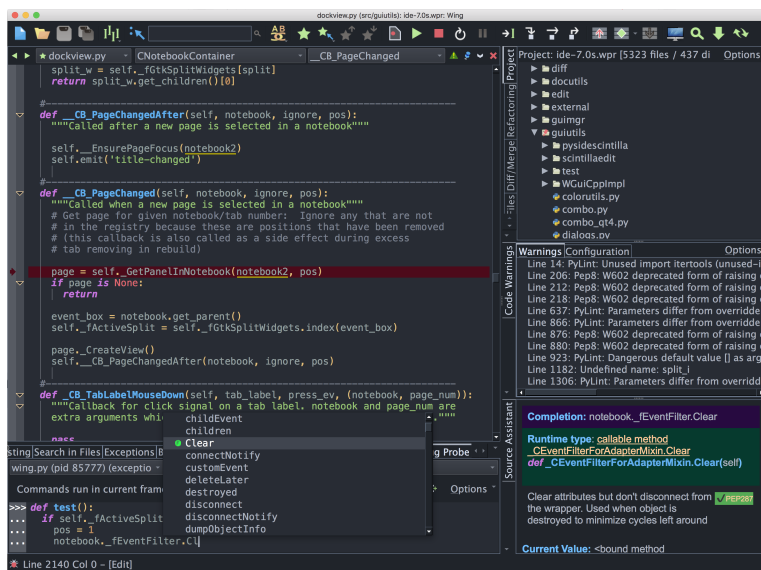
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code running on the Raspberry Pi.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Raspberry Pi. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Configuration

Wing Pro does not run on the Raspberry Pi, but you can set up Wing Pro on a computer connected to the Raspberry Pi to work on and debug Python code remotely. This is configured as follows:

- If you do not already have Wing Pro installed, download it now on Windows, Linux, or macOS.
- Make sure you can connect to the Raspberry Pi from the machine where Wing IDE will be running, using **ssh** (or PuTTY on Windows). If you don't have either **ssh** or PuTTY, you can also connect using Wing's built-in SSH implementation. See Setting up SSH for Remote Development for a detailed description of the available configuration options. However, in most cases all you need to know is the Raspberry Pi's ip address.

- Start up Wing and use **New Project** from the **Project** menu to create a project. Select **Create New Configuration** from the **Host** menu in the **New Project** dialog and then select **Raspberry Pi** for the **Remote Host Type** in the **New Remote Host** dialog.
- Then fill in the **New Remote Host** fields as follows:

    - **Identifier** -- Set this to **rasp** or some other short identifier for the Raspberry Pi. This name is used only within Wing.
    - **Host Name** -- Set this to the string you use to SSH into the Raspberry Pi. In most cases you'll need both a username and IP address, such as **pi@192.168.0.2**.

    Note that you can edit your configuration later, or add remote hosts to any project, from the **Remote Hosts** item in the **Project** menu.

- Next click **OK** to create the remote host configuration. Wing will attempt to install the remote agent and then establish a connection to the Raspberry Pi. If this fails, details of the SSH command's output will be given in the resulting dialog.
- Once you have the remote agent working, continue through the **New Project** dialog to select or create your source directory and select or create the Python environment to use. See Creating a Project for details.

    After you press **Create Project** Wing will create a new untitled project configured to work with your Raspberry Pi. Save it to local disk from the **Project** menu, for example as **rasp.wpr**.

Once this is done, you can open files from the Project tool, with **Open From Project** and in other ways, and work with them as if they were on your local machine. That includes debugging, running unit tests, issuing revision control commands, searching, running a Python Shell or OS Commands remotely, and using other features like goto-definition, find uses, and refactoring.

Related Documents

For more information see:

- Raspberry Pi home page for documentation and downloads.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

1.11. Using Wing with Cygwin

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for cygwin, a Linux/Unix like environment for Microsoft Windows.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Cygwin. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Note: This document was last tested with cygwin 3.6.

Project Configuration

To write and debug code running under cygwin, download and install Wing for Windows on your machine. Wing does not run on cygwin but you can set up Wing for Windows to work with Python code that is running under cygwin.

This is done by creating a project with **New Project** in the **Project** menu, selecting **Create Blank Project**, pressing **Create Project**, and then adding the Windows-side copies of your source files to the project with **Add Existing Directory** in the **Project** menu.

Debugger Configuration

To debug code running on cygwin, follow the instructions for Debugging Externally Launched Code. In this model, you will always launch your Python code from cygwin and not from Wing's menus or toolbar.

When setting this up, use cygwin paths for **WINGHOME** in **wingdbstub.py** because this file will be used on the cygwin side.

File Paths

It is often easiest to configure cygwin pathnames to be equivalent to the Windows pathnames. An example would be to set up **/src** in cygwin to point to the same directory as **\src** on Windows, which is **src** at top level of the main drive, usually **c:\src**.

If this is not possible, you should be sure to add all the sources you need to work with to your project in Wing. This way, Wing can automatically find all your files and use a hash on the contents of the file to identify which Windows-side files are the same as the cygwin files. See File Location Maps for details.

Related Documents

For more information see:

- Cygwin home page, which provides links to documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

1.12. Remote Python Development
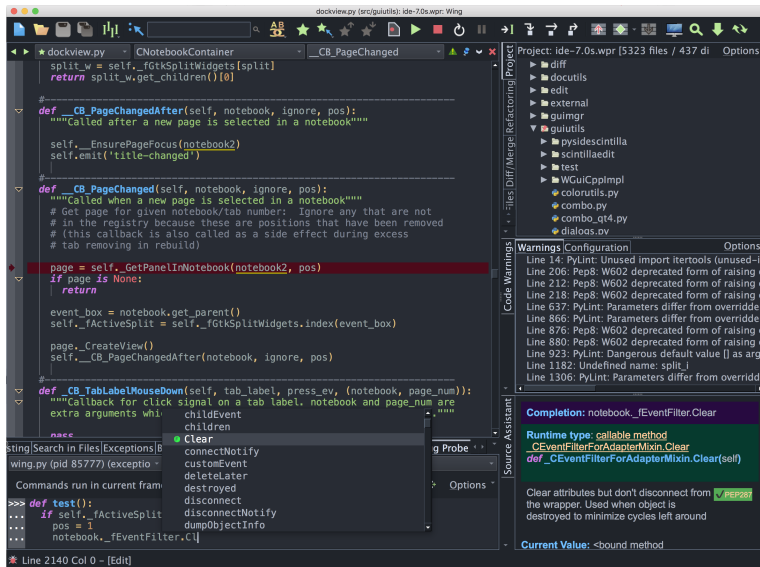
Wing Pro can connect securely to a remote host, VM, or container, in order to work with files on the remote system in the same way that Wing supports working with files on your local system. Editing, debugging, testing, searching, version control, Python Shell, OS Commands, and other features all work with remote systems.

Currently, Wing can work remotely to macOS and Intel or ARM Linux systems. This includes any PEP 513 compatible Intel Linux system and ARM systems like Raspberry Pi and Jolla phone. We are still expanding the range of remote systems that we support. For a detailed list of the remote host types supported in the current release, please see Supported Platforms. If you try a device and cannot get it working, don't hesitate to email support@wingware.com for help.

Configuration

Before you can set up remote development in Wing, you need to be able to connect to your remote system using SSH. This can be tested outside of Wing with **ssh** on Linux and macOS or either OpenSSH (provided by **Cygwin**, **Git Bash**, or similar) or **PuTTY** on Windows. If you don't have either **ssh** or PuTTY, Wing falls back to using its own built-in SSH implementation. See Setting up SSH for Remote Development for a detailed description of all the available configuration options. However, in most cases all you will need to know is the user name and the ip address or host name of the remote system.

Creating a Project

To set up a new project that works with a remote host, select **New Project** from the **Project** menu and choose **Create New Configuration** from the **Host** menu in the **New Project** dialog. Then enter an **Identifier** to use for the remote host and the **Host Name** or ip address (optionally in the form **username@hostname**). You only need to specify **Python Executable** if **python** is not on the **PATH** on your selected remote host or you want to select one of several Python installations.

For example, here is a configuration to access a Linux system on a local network from macOS:
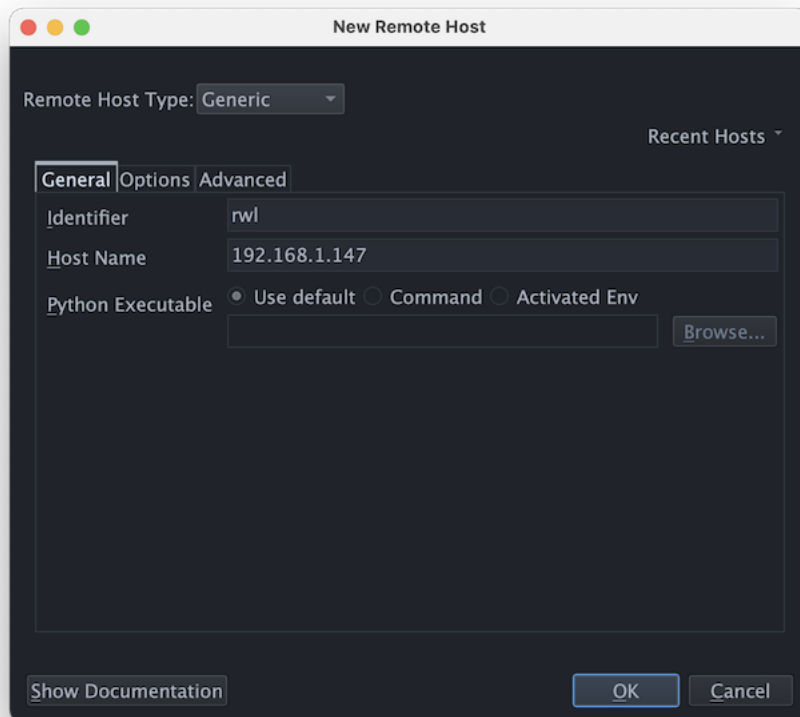
The press **OK** to create the remote host configuration and install the remote agent if necessary.

If the remote agent needed to be installed or upgraded then Wing will briefly show a status dialog during installation and inspection of the remote system, followed by a confirmation dialog:
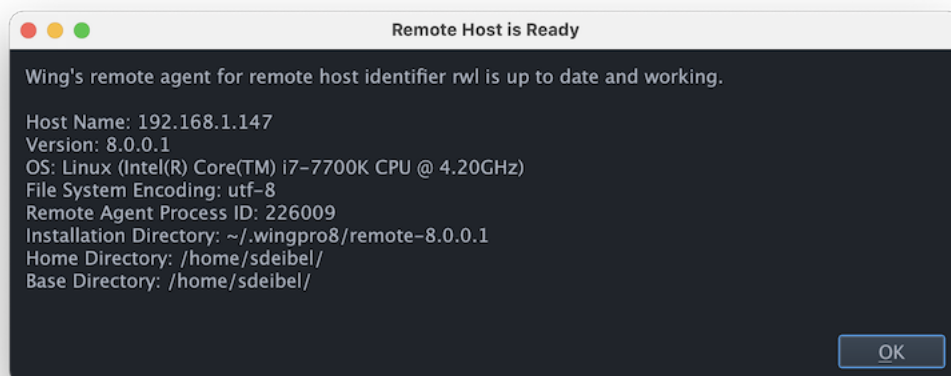
Close this dialog and you will see that the new remote host configuration has been selected for your new project:
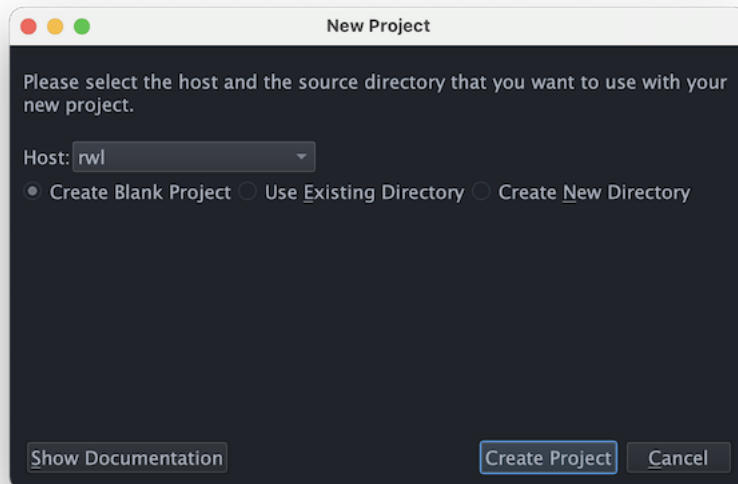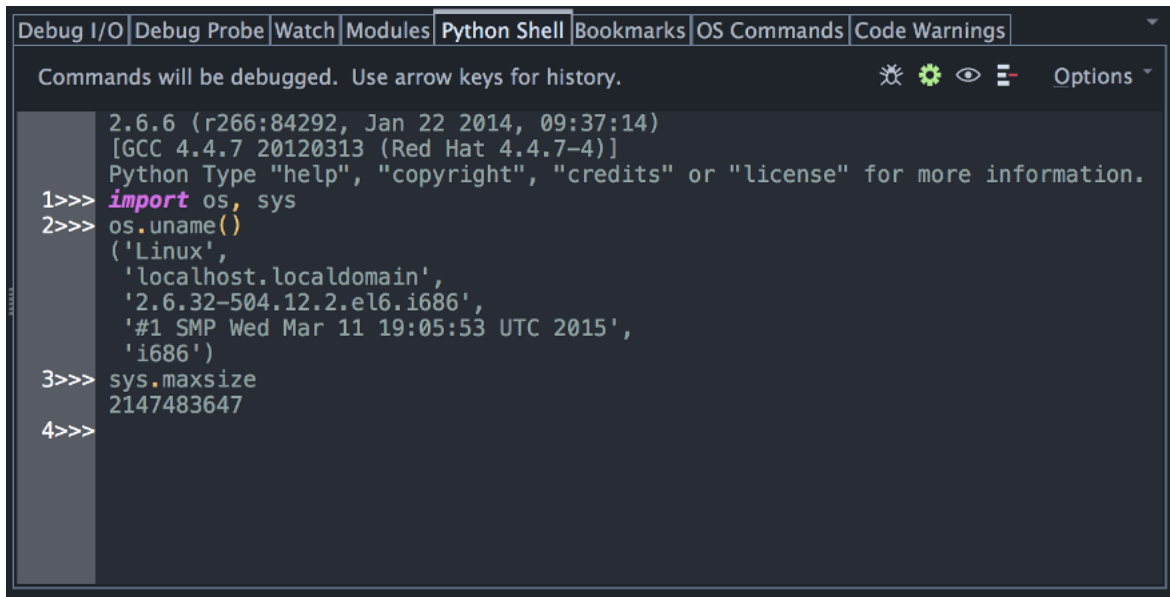
You can now continue by selecting or creating your source directory and selecting or creating a Python environment to use with the project. See Creating a Project for details on the configuration options.

After you press the **Create Project** button in the **New Project** dialog, Wing will set up your project. You can then save the project to local disk with **Save Project** in the **Project** menu.

Using Your Project
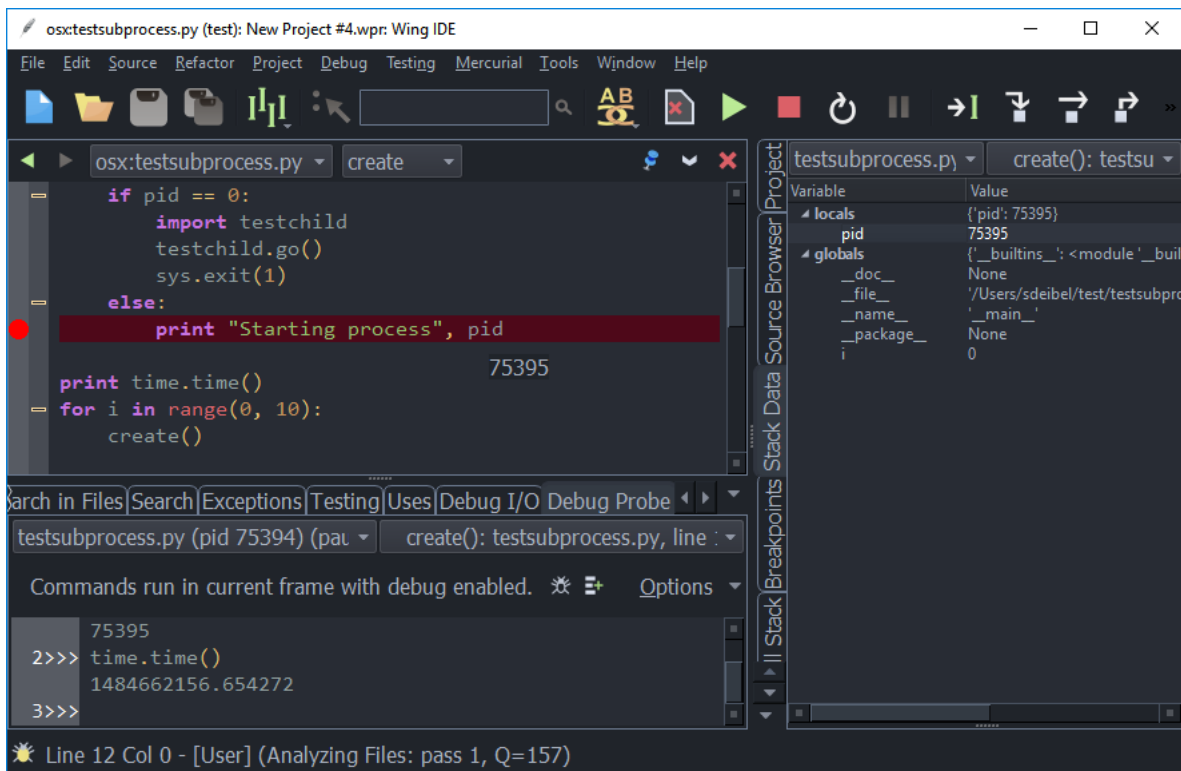
If you bring up the **Python Shell** from the **Tools** menu, you should be able to interact with the Python environment you selected on your remote host. For example, here is Python running remotely on a CentOS 6 system from Wing on macOS:

To debug, open a file from the directory you added to the project and select **Start/Continue** in the **Debug** menu. Wing launches the file in the debugger on the remote host and will reach breakpoints and exceptions. Debugging a remote file works the same way as for local files. You can use the **Debug Console**, **Stack Data**, **Watch** and other tools to inspect and debug your code.

Other tools, including the **Testing** tool for unit testing, the version control integrations, and **OS Commands** for executing non-Python command lines all work on the remote host.

Details

The remote host configuration you created along with your project is stored inside your project file. You can view and edit the configuration, or create other remote host configurations, from **Remote Hosts** in the **Project** menu:
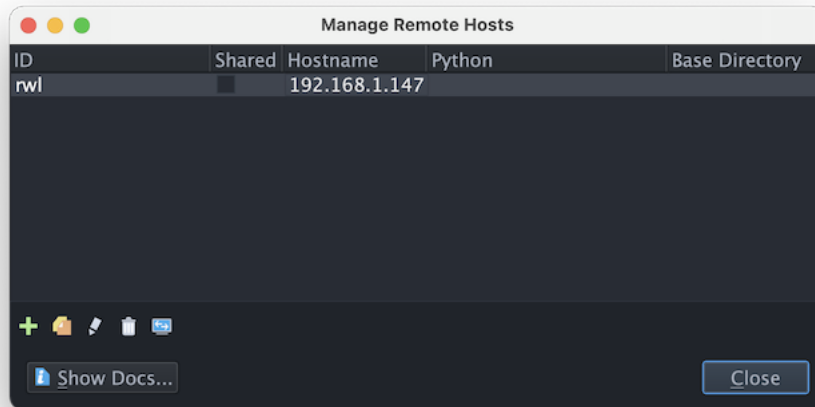
It is possible to set up multiple remote host configurations for one project, but the project's **Python Executable** in **Project Properties** (from the **Project** menu) can only point to one of the remote hosts, and that is where the **Python Shell** and debug processes are run. Changing the **Python Executable** is what determines whether a project points to local disk or some remote system. Here are the **Project Properties** that were set up automatically in the project we created above:
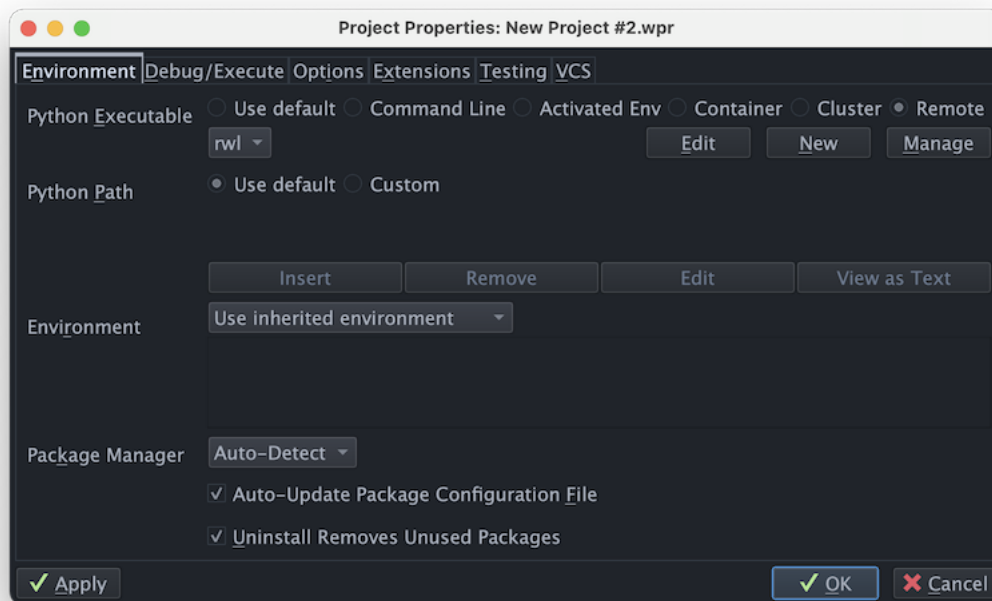
Remotely Stored Projects

In this example, we stored the project file on local disk, but project files can also be stored on the remote host. In that case, the remote host configuration needs to be checked as **Shared**. This stores the remote host configuration locally so that it can be used to access the remote project later with **Open Remote Project** from the **Project** menu.

You can also use this feature to remotely open a regular locally created Wing project.

Remote Display with X11

To work with code that displays a user interface, you can forward X11 display to occur on the machine where Wing is running. This is done by checking the **Forward X11** option in your remote host configuration, under the **Options** tab.

Unless Wing is running on Linux, you will also need to install and run an X11 server on the machine where Wing is runing, for example **XQuartz** on macOS or **MobaXTerm** on Windows.

Further Reading

For more information see:

- Remote Hosts for more detailed instructions and advanced configuration options.
- Remote Web Development describes how to set up remote development where the debug process is launched from outside of the IDE, for example by a web server or framework.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

Please don't hesitate to contact support@wingware.com if you need help getting remote development working.

How-Tos for Scientific and Engineering Tools

The following How-Tos explain how to get started using Wing with tools for scientific and engineering data analysis and visualization.

2.1. Using Wing with Matplotlib
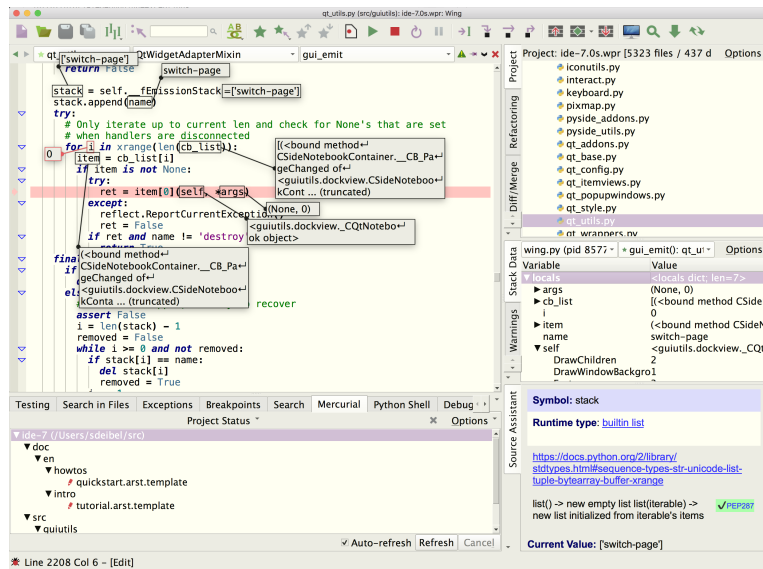
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Matplotlib, a powerful numerical and scientific plotting library.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Matplotlib. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Working Interactively

Wing supports interactive development and debugging of Python code designed for the Matplotlib numerical and scientific plotting library, so plots can be shown and updated from the command line. For example, two plots could be shown in succession by typing the following into Wing's integrated **Python Shell**, one line at a time:

```
from matplotlib.pyplot import plot, show
x = range(10)
plot(x)
show()
y = [2, 8, 3, 9, 4] * 2
plot(y)
```

Wing sets up the environment so that **show()** runs to completion and immediately returns you to the prompt, rather than waiting until the plot is closed. In addition, Wing calls Matplotlib's main loop to keep plots windows interactive and updating while you are at the prompt. This allows plots to be added or changed without restarting a process or interrupting your work flow.

Evaluating Files and Selections

Code from the editor can be executed in the **Python Shell** with the **Evaluate ... in Python Shell** items in the **Source** menu and in the editor's right-click context menu. By default the **Python Shell** restarts automatically before evaluating a whole file, but this can be disabled in its **Options** menu.

Active Ranges

Wing also allows you to set a selected range of lines in the editor as the "active range" for the **Python Shell** by clicking the ⊞+ icon in the top right of the **Python Shell** tool. Wing highlights and maintains the active range as you edit it in the editor, and it can be re-evaluated easily with the ✿ icon that appears in the top right of the **Python Shell** once an active range has been set into it. Use the ⊟– icon to clear the active range from the editor and shell.

Supported Backends

Interactive development is supported for the **TkAgg**, **GTKAgg**, **GtkCairo**, **WXAgg** (for wxPython 2.5+), **Qt5Agg**, **Qt4Agg**, **MacOSX**, and **WebAgg** backends. It will not work with other backends.

Debugging

Code can be debugged either by launching a file with ▶ in the toolbar (or **Start/Continue** the **Debug** menu) or by enabling debug in the integrated **Python Shell** and working from there. In either case, Wing can be used to reach breakpoints or exceptions, step through code, and view the program's data. For general information on using Wing's debugger see the Debugger Quick Start.

When executing code that includes **show()** in the debugger, Wing will block within the **show()** call just as Python would if launched on the same file. This is by design, since the debugger seeks to replicate as closely as possible how Python normally runs.

However, interactive development from a breakpoint or exception is still possible, as described below. This capability can be used to load setup code before interacting with Matplotlib, or to try out a fix when an exception has been reached.

Interactive Debugging from the Debug Console (Wing Pro only)

Whenever the debugger is stopped at a breakpoint or exception, Wing Pro's **Debug Console** provides a command prompt that may be used to inspect and interact with the paused debug process. Commands entered here run in the context of the currently selected debug stack frame.

The tool implements the same support for interactive development provided by the **Python Shell**, so plots may be shown and modified interactively whenever Wing's debugger is paused. Once the debug process is continued, Wing switches off interactive mode and returns to behaving in the same way that Python would when running the code outside of the debugger.

> ### *Note*
>
> Interactive development from the **Debug Console** requires that you have already imported **matplotlib** in the code that you are debugging or in a previous command entered in the console. Otherwise **show()** may block and plots won't be updated.

Interactive Debugging from the Python Shell

Another way to combine the debugger with interactive development is to turn on both **Enable Debugging** and **Enable Recursive Prompt** in the **Python Shell**'s **Options** menu. This causes Wing to add a breakpoint margin to the **Python Shell** and to stop in the debugger if an exception or breakpoint is reached, either in code in the editor or code that was entered into the **Python Shell**.

The option **Enable Recursive Prompt** causes Wing to show a new recursive prompt in the **Python Shell** whenever the debugger is paused, rather than waiting for completion of the original command before showing another prompt. Showing or updating plots from recursive prompts works interactively in the same way as described earlier.

If another exception or breakpoint is reached, Wing stops at those as well, recursively to any depth. Continuing the debug process from a recursive prompt completes the innermost invocation and returns to the previous recursive prompt, unless another exception or breakpoint is reached first.

Trouble-shooting

If **show()** blocks when typed into the **Python Shell**, if plots fail to update, or if you run into other event loop problems while working with Matplotlib, then the following may help solve the problem:

(1) When working in the **Debug Console**, evaluate the imports that set up Matplotlib first, so that Wing can initialize its event loop support before **show()** is called. Evaluating a whole file at once in the **Debug Console** (but not the **Python Shell**) will cause **show()** to block if Matplotlib was not previously imported.

(2) In case there is a problem with the specific Matplotlib backend that you are using, try the following as a way to switch to another backend before issuing any other commands:

```
import matplotlib
matplotlib.use('TkAgg')
```

Instead of **TkAgg** you may also try other supported backends, including **Qt5Agg** (which requires that Qt5 is installed) or **WebAgg** (which uses a web browser for plot display).

Please email support@wingware.com if you run into problems that you cannot resolve.

Related Documents

For more information see:

- The Matplotlib website
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

2.2. Using Wing with Jupyter Notebooks
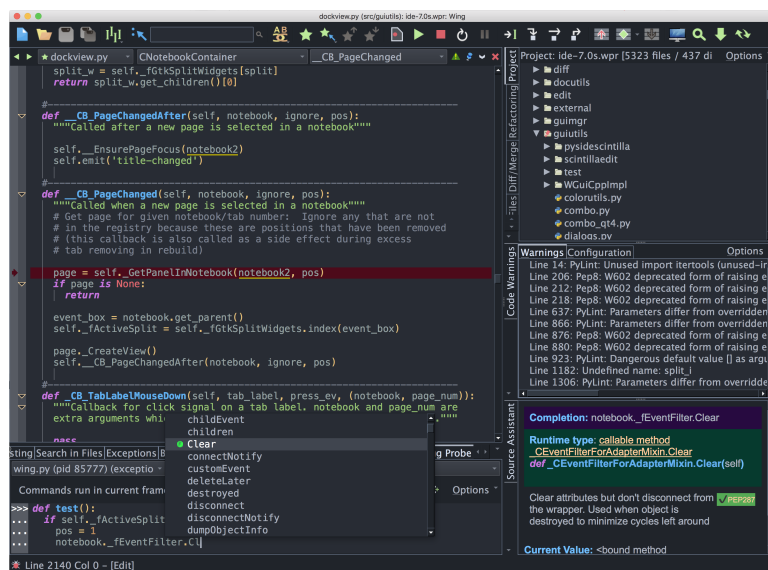
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Jupyter, an open source scientific notebook system.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Jupyter. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Setting up Debug

Since Jupyter is started outside of Wing, you will need to initiate debug from your code or from the Jupyter notebook. There are a few configuration options that need to be set correctly for this to work properly.

Limitation: Jupyter does not provide a usable filename for code that resides directly in a notebook **.ipynb** file (it is simply set to names like **<ipython-input-1>**). As a result you cannot stop in or step through code in the notebook itself. Instead, you need to place your code in a Python file that is imported into the notebook, and then set breakpoints and step through code in the Python file.

Configure wingdbstub.py

To initiate debug, you will need to copy **wingdbstub.py** out of your Wing installation (on macOS it is located in **Contents/Resources** within the **.app** bundle) and place it in the same directory as your **.ipynb** file.

You may need to set **WINGHOME** inside of **wingdbstub.py** to the installation location of Wing. This is set automatically during installation of Wing except on macOS, on Windows if you use the zip installer, and on Linux if you use the tar installer. An alternative to editing **wingdbstub.py** is just to set the environment variable **WINGHOME** before you run **jupyter notebook**.

Listen for Debug Connections

Next, tell Wing to listen for externally initiated debug connections by clicking on the bug icon in the lower left of Wing's window and checking on **Accept Debug Connections**.

Starting Debug

Now add code like the following to the top of your Jupyter notebook:

```
import wingdbstub
wingdbstub.Ensure()
```

When you run that cell, Wing will start debugging Jupyter. You should see Wing's toolbar change and the **Stack Data** tool should show one running process:
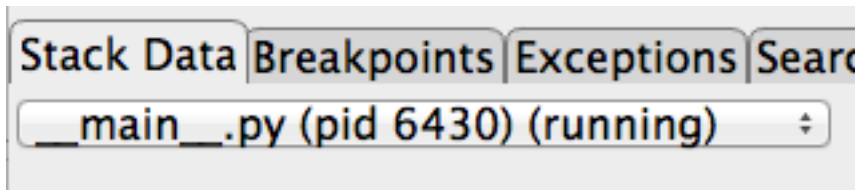
Working with the Debugger

To try out debugging, save a file named **testdebug.py** in the same directory as your **.ipynb** file with the following contents:

```
def run():
    print("Hello world")
    x = 1
    print("Done")
```

Open this in Wing and place a breakpoint on the first line by of the body of **run()** by clicking on the breakpoint margin to the left, as follows:
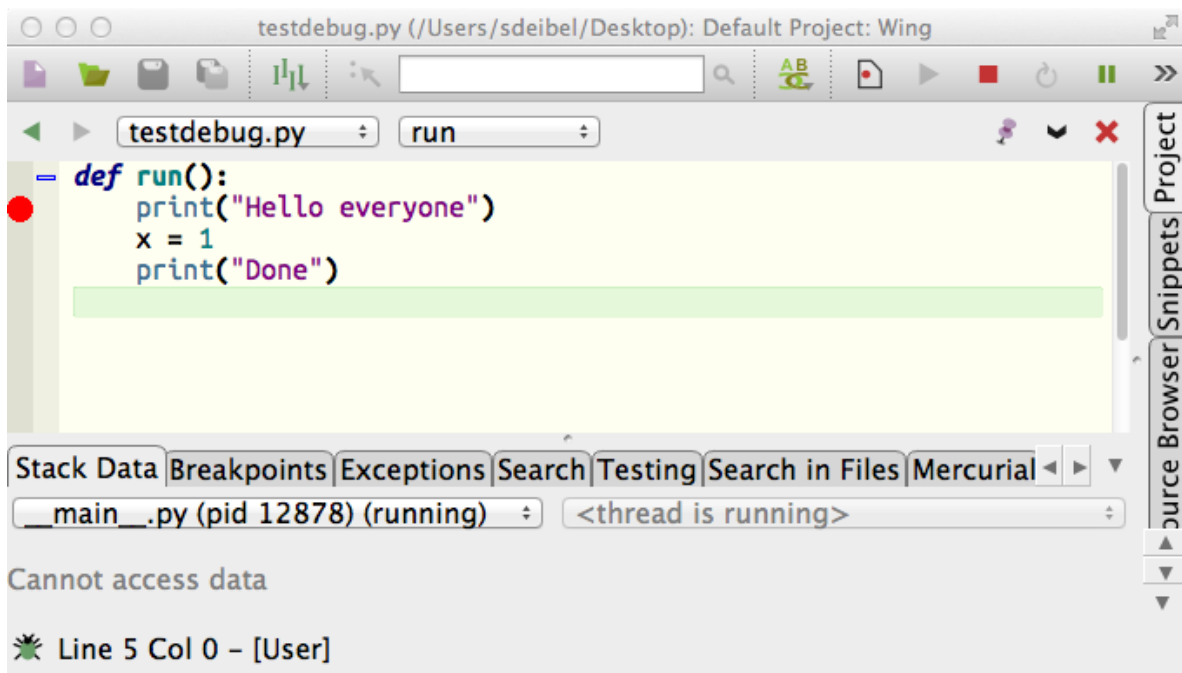
Now add the following cell to your Jupyter notebook:

```
import testdebug
testdebug.run()
```

When you execute that cell, Wing should stop on the breakpoint in **testdebug.py**:
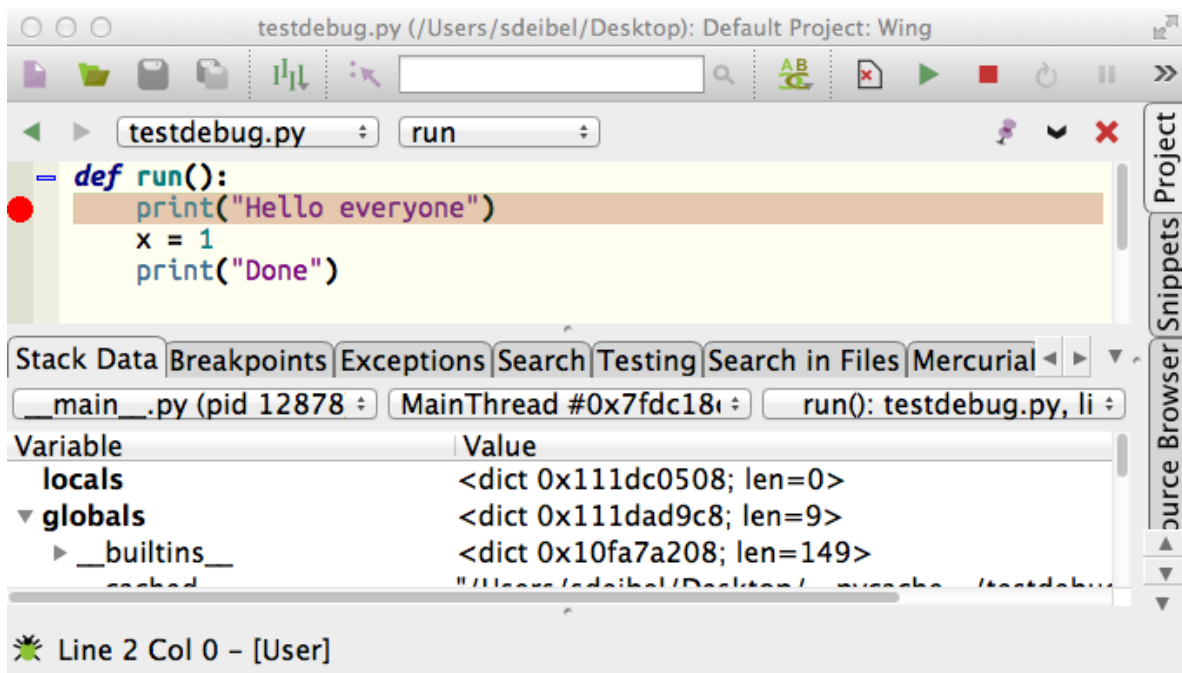
Now you can use the toolbar icons to step through code, view data in the **Stack Data** tool in Wing, interact in the context of the current debug stack frame with the **Debug Console** (Wing Pro only), and use all of Wing's other debugging features on your code. See the **Tutorial** in Wing's **Help** menu for more detailed information on Wing's debugging capabilities.

To complete execution of your cell, press the green continue arrow ▶ in the toolbar. Now if you execute the cell again, you should reach your breakpoint a second time. Then continue again to complete execution of the cell.

Editing Code

Now try editing code in **testdebug.py** to change **Hello world** to **Hello everyone** and save the file. If you execute your cell again in Jupyter you'll notice the text being output has not changed. This is because the module has already been imported by Python and Jupyter is not automatically reloading it. To load your changes you'll need to restart the kernel from Jupyter's toolbar or its **Kernel** menu. In many cases **Restart and Run All** in the **Kernel** menu will be the most efficient way to reload your code and get back to your breakpoint.

Try selecting the **Source Assistant** from Wing's **Tools** menu and then adding some other code in **testdebug.py**, for example add **z = yy** for your code reads as follows:

```
def run():
    print("Hello everyone")
    z = yy
    print("Done")
```

Notice that Wing offers auto-completion and updates the **Source Assistant** with call tips, documentation, and other information about what you are typing, or what you have selected in the auto-completer. If a debug process is active and the code you are typing is on the stack, Wing includes also symbols found through inspection of the live runtime state in the auto-completer. In some code, but not the above example, this can include information Wing was not able to find through static analysis of the Python code.

Working in live code like this is a great way to write new code in the **Debug Console**, where you can try it out immediately.

Or, you can work in the editor and try out selected lines of code by pressing the ☷+ icon in top right of the **Debug Console** to make an active range. Once that is done, you can execute those lines repeatedly by pressing the ⚙ icon in the **Debug Console**:
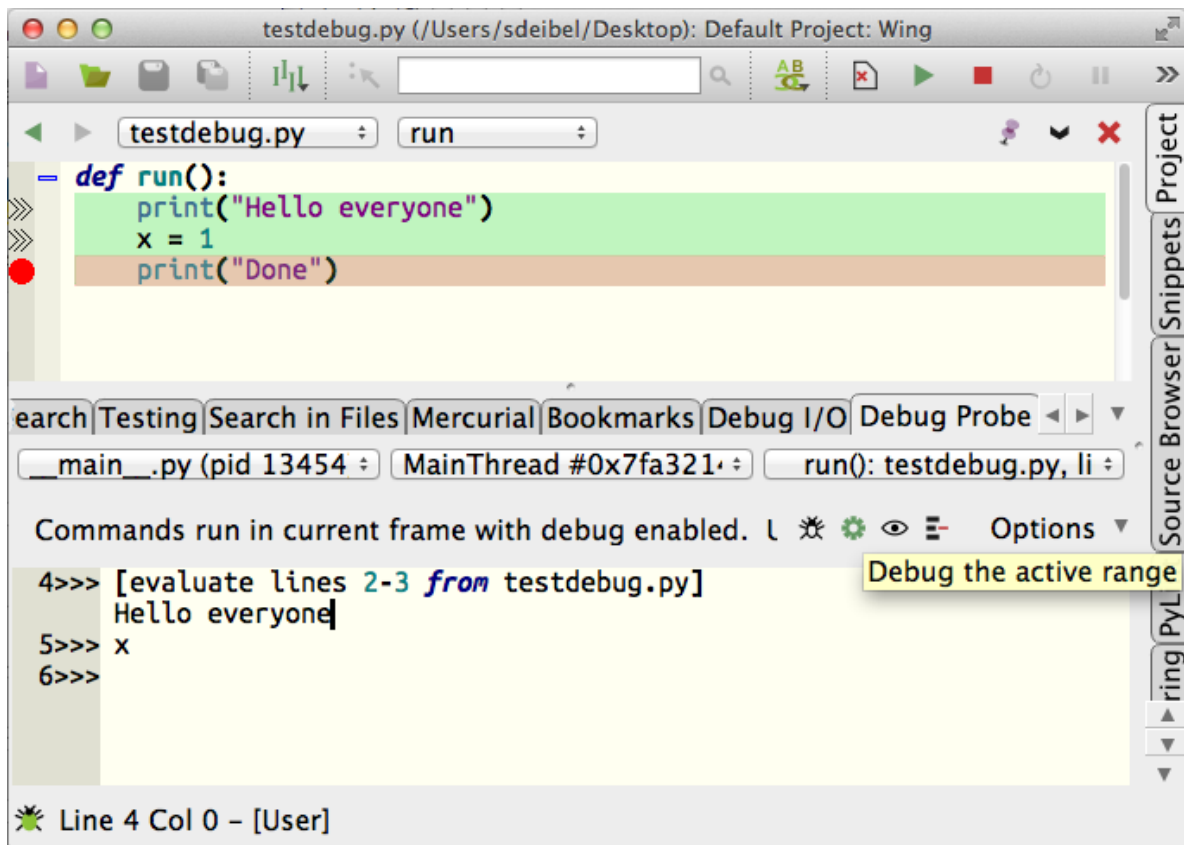
Stopping on Exceptions

Since Jupyter handles all exceptions that occur while executing a cell, Wing will not stop on most exceptions in your code. Instead, you will get the usual report in the notebook output area.

Try this by now by restarting the Jupyter kernel and executing your edited copy of **testdebug.py**, which should read as follows:

```
def run():
    print("Hello everyone")
    z = yy
    print("Done")
```

Jupyter will report the exception in the notebook (undefined symbol **yy**), but Wing will not stop on it.

It is possible to get Wing to stop on exceptions, although currently the only way to do that is to edit code in IPython's **interactiveshell.py**. You can easily find that by setting a breakpoint in **run()** as before and going up the stack in Wing using the **Stack Data** or **Call Stack** tool. Then add the following code to the final **except:** clause in **InteractiveShell.runcode**. This will log the exception, which Wing takes as a clue that it should report the exception even though it is being handled:

```
if 'WINGDB_ACTIVE' in os.environ:
    import logging
    logging.exception(sys.exc_info()[1])
```

You will need to restart the Jupyter kernel after making this change. Then try executing your cell again and you will see Wing now reports the exception:
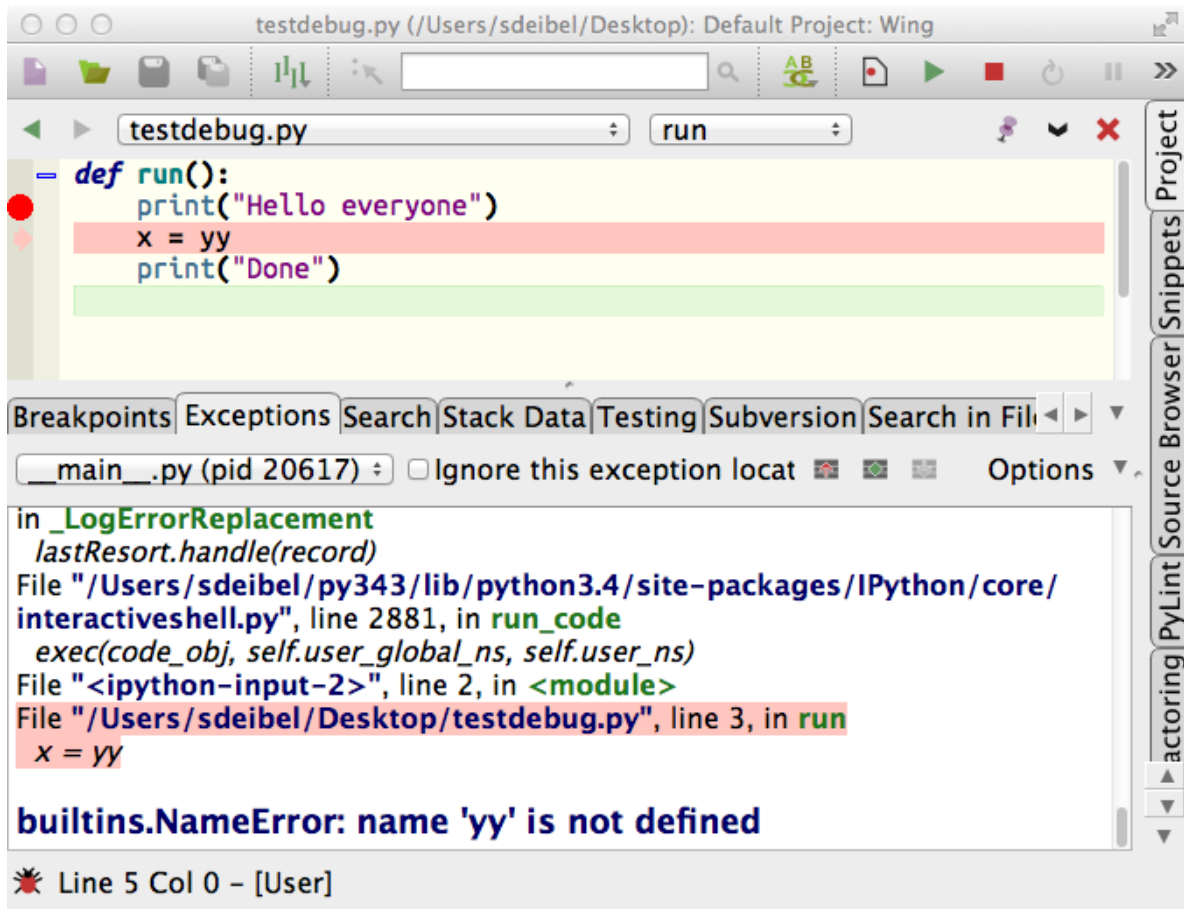
You can continue as usual from the exception and it will also be reported in the Jupyter notebook.

Fixing Failure to Debug

If you accidentally disconnect Wing's debugger from Jupyter, for example by pressing the red stop icon ■ in Wing's toolbar, you can reestablish the debug connection at any time by re-executing the first cell we set up above, or by placing the following code into any other code that gets executed:

```
import wingdbstub
wingdbstub.Ensure()
```

Note that if you plan to restart the Jupyter kernel every time you start debug then you don't need the **wingdbstub.Ensure** line. This makes sure that debug is active and connected to the IDE, so it is only needed if the debug connection has been dropped since the first time **wingdbstub** was imported.

If debugging stops working entirely and this does not solve it, you will need to restart the Jupyter kernel from its toolbar or **Kernel** menu and then re-execute the above code to start debugging again.

Reloading Changed Modules

The instructions above rely on restarting of the kernel as the way to reload changed code into Jupyter. Module reloading is also an option, making it possible to reload code without restarting the kernel.

Simple module reloading can be done using Python's builtin function **reload()** (or in Python 3.x instead **imp.reload()** after **import imp**). For details see instructions for reloading in IPython.

Or, for more complex cases, the autoreload extension for IPython may help.

In general module reload can be problematic if old program state is not cleared correctly, and the complexity of this depends on the modules being used and their implementations. Simply restarting the kernel is always the safest option.

Related Documents

Wing provides many other options and tools. For more information:

- Jupyter website
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

2.3. Using Wing with PyXLL

> **Note**
>
> **"Out of all the Python IDEs available I found Wing to have the fastest and easiest to use debugger by far. Using it to debug Python code running in Excel with PyXLL is a joy!"** *-- Tony Roberts*
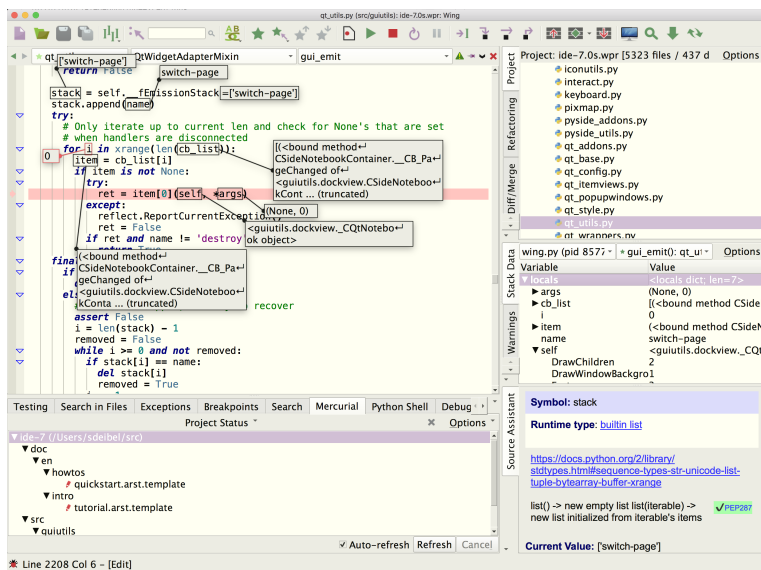
[Wing Pro](#) is a Python IDE that can be used to develop, test and debug Microsoft Excel add-ins written in Python with [PyXLL](#).

This document focuses on configuring Wing to debug Python code running in Excel. To learn more about Wing in general, please refer to the **Tutorial** in Wing's **Help** menu or read the [Quickstart Guide](#).

Introduction

[PyXLL](#) is a commercial product that embeds Python into Microsoft Excel on Windows. It allows you to expose Python code to Excel as worksheet functions, macros, menus, and ribbon toolbars.

PyXLL add-ins can be developed, tested, and debugged using Wing. Wing's [remote debugger](#) is used to connect to Excel in order to debug the Python code.

Installation and Configuration

Take the following steps to set up and configure Wing for use with PyXLL:

- Install PyXLL as described in the [PyXLL user guide](#). Be sure the follow this guide to the end and install the optional PyXLL wheel using pip.

- Install Wing if you don't already have it.
- Launch Wing from the **Start** menu on Windows.
- Create a new project in Wing with **New Project** in the **Project** menu. Select **Create Blank Project** as the project type and press **Create Project**. Then use **Project Properties** in the project creation confirmation dialog or **Project** menu to set **Python Executable** to **Command Line** and then enter the full path to the Python you are using with PyXLL. This is the same value used for **executable** in the PyXLL config file.
- Locate the folder where you have installed PyXLL and in Wing select **Add Existing Directory** from the **Project** menu to add it to your project. Also add any other directories that store the source code you are working on.
- Save your project to disk with **Save Project As** in the **Project** menu.

Debugging Python Code in Excel

This section describes how to debug Python code running in the Excel process through the PyXLL add-in.

- Copy **wingdbstub.py** from the **Install Directory**, listed in Wing's **About** box, accessed from the **Help** menu, into a directory listed on the **pythonpath** in your PyXLL config file. If you are just starting with PyXLL, this could be the **examples** folder in your PyXLL folder.
- Open your copy of **wingdbstub.py** and make the following changes:

    1. Make sure **WINGHOME** is set to the full path of the Wing installation from which you copied **wingdbstub.py**. This may already be done, since it is usually set automatically during installation.
    2. Change the value of **kEmbedded** to **1**. This tells Wing's debugger that you are working with an embedded copy of Python, which affects some aspects of how code is debugged.

- Add **wingdbstub** to the modules list in your **pyxll.cfg** file:

```
[PYXLL]
modules =
   wingdbstub
   ...
```

- Make sure the **Debugger > Listening > Accept Debug Connections** preference is enabled on in Wing, to allow debug connections from the Excel process. This can also be enabled by clicking on the bug icon in the lower left of Wing's window.

    Now hovering your mouse over the bug icon should show that Wing is listening for externally initiated debug connections on the local host.

    If Wing is not listening, it may be that it has not been allowed to do so by Windows. In that case, try restarting Wing so that Windows will prompt you to allow network connections.

- Set any required breakpoints in your Python source code by clicking on the leftmost margin next to the code in Wing's editor, or with the breakpoint items in the **Debug** menu.

- Restart Excel or reload the PyXLL add-in so that the **wingdbstub** module is imported. You should see the status indicator in the lower left of Wing's window change to yellow, red, or green, as described in Debugger Status.
- Call a Python function from Excel that will reach a breakpoint.

When a breakpoint is reached, Wing will come to the front and show the file where the debugger has stopped. If no breakpoint or exception is reached, the program will run to completion, or you can use the **Pause** command in the **Debug** menu.

Trouble-shooting

If this doesn't work at first, try using **wingdbstub.Ensure()** to force **wingdbstub** to make the connection to the debugger. The following code creates an Excel worksheet function that, when called, ensures the debugger is connected:

```
from pyxll import xl_func
import wingdbstub

@xl_func
def debug_test():
    wingdbstub.Ensure()
    return "Connected Ok!"
```

If this code can't connect then check that the Wing application is allowed to make network connections in your Windows Firewall settings. To do this, go to the Windows **Start** menu and type "Allow an app through Windows firewall", select "Change Settings" and then "Allow another app...". Navigate to the Wing installation folder and select the Wing executable from the **bin** folder. Restart Wing and Excel and now the two should be able to connect.

If you still have problems making this work, try setting the **kLogFile** variable in **wingdbstub.py** to log additional diagnostic information. This diagnostic output can be emailed to support@wingware.com for help.

Related Documents

Wing provides many other options and tools. For more information:

- PyXLL website.
- Debugging Externally Launched Code.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

How-Tos for Web Development

The following How-Tos explain how to get started using Wing with a number of popular web development frameworks.

3.1. Remote Web Development
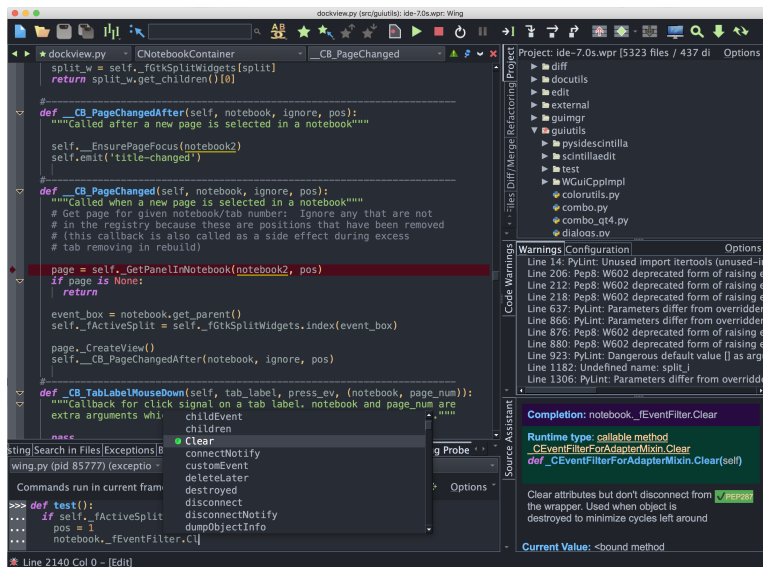
Wing Pro is a Python IDE that can be used to develop, test, and debug websites running on a remote server, VM, or other system where an IDE cannot be installed. Debugging takes place in the context of the web server or web framework, as code is invoked by browser page load. Wing provides auto-completion, call tips, find uses, and many other features that help you work with Python code.

If you do not already have Wing Pro installed, download it now.

This document focuses on how to configure Wing for remote web development. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Setting up SSH Access

Wing Pro's remote development support assumes that you have already set access to the remote host with SSH. This can be tested outside of Wing using **ssh** or PuTTY, or you can use Wing's built-in SSH implementation. See Setting up SSH for Remote Development for a detailed description of the available configuration options. However, in most cases all you will need to know is the user name and the ip address or host name of the remote system.

Installing the Remote Agent

The next step is to set up a remote host configuration from **Remote Hosts** in the **Project** menu. Press the **+** icon to add a new remote host configuration.

You will need to choose a short identifier that Wing will use to refer to the host and enter the hostname, which may be a name or an IP address and can be in **username@hostname** form if the remote user name does not match the local one. You will only rarely need to specify any of the other values in a remote host configuration. For now, leave them blank. For example:
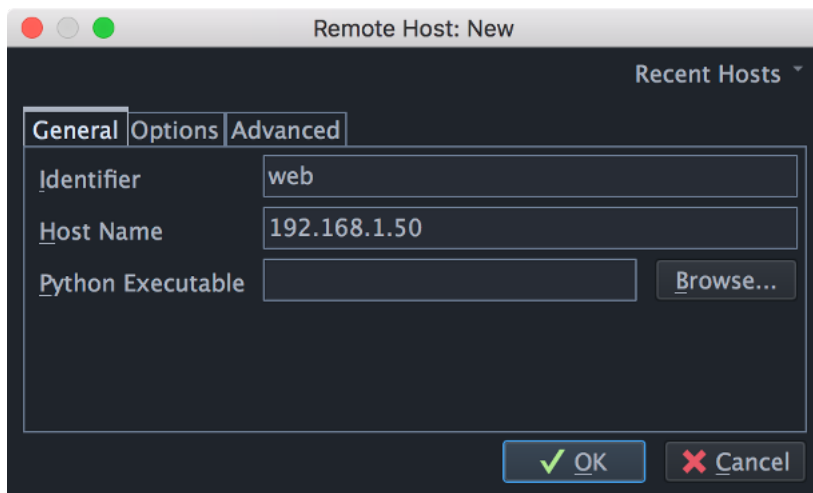
Once you submit the dialog for creating the configuration, Wing will try to probe the remote host and then install the appropriate remote agent. When the process completes it should confirm that the remote agent is installed and working as follows:
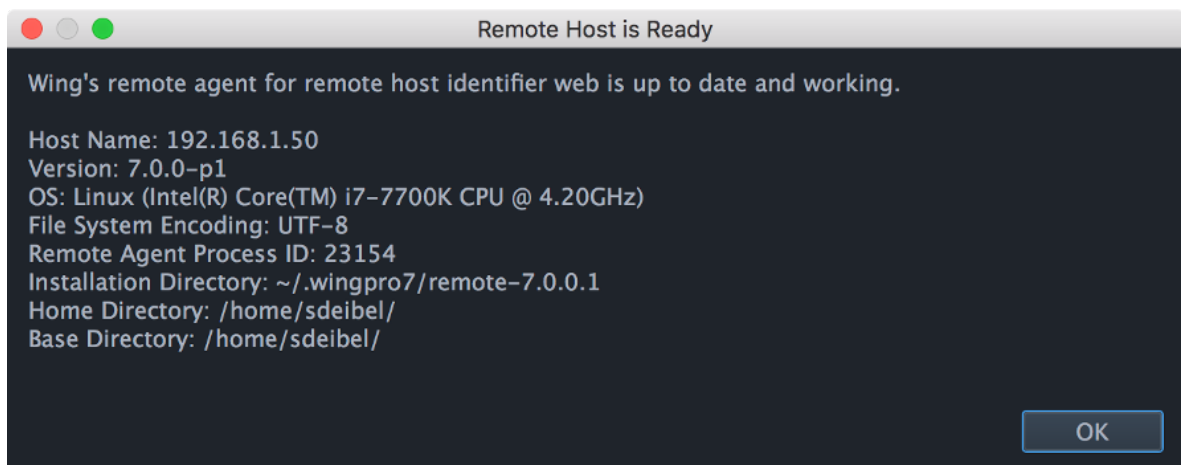
Next return to the **Remote Hosts** dialog and specify that the remote host configuration you've just created should be shared, so that it isn't just stored in the currently open project:

73
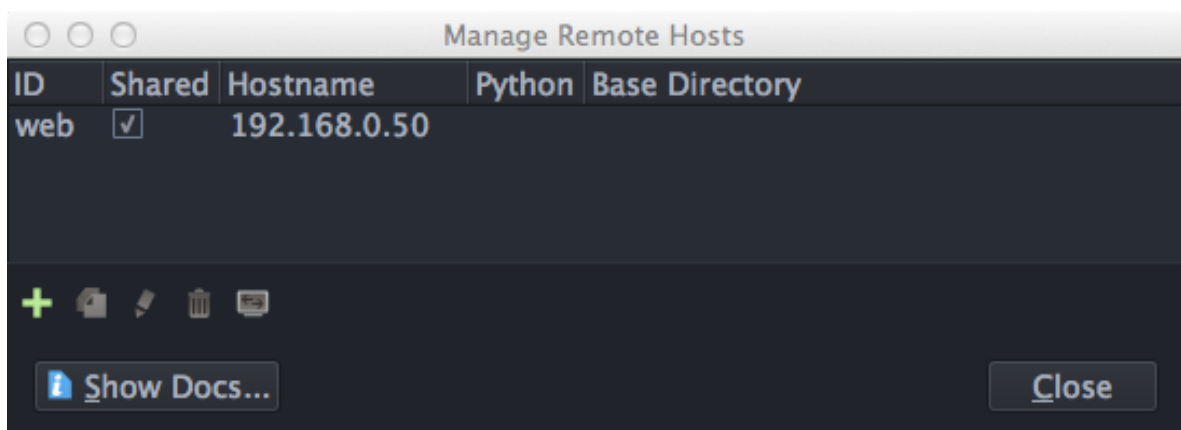
If something goes wrong during the remote agent installation process, Wing will instead show a dialog similar to the following:
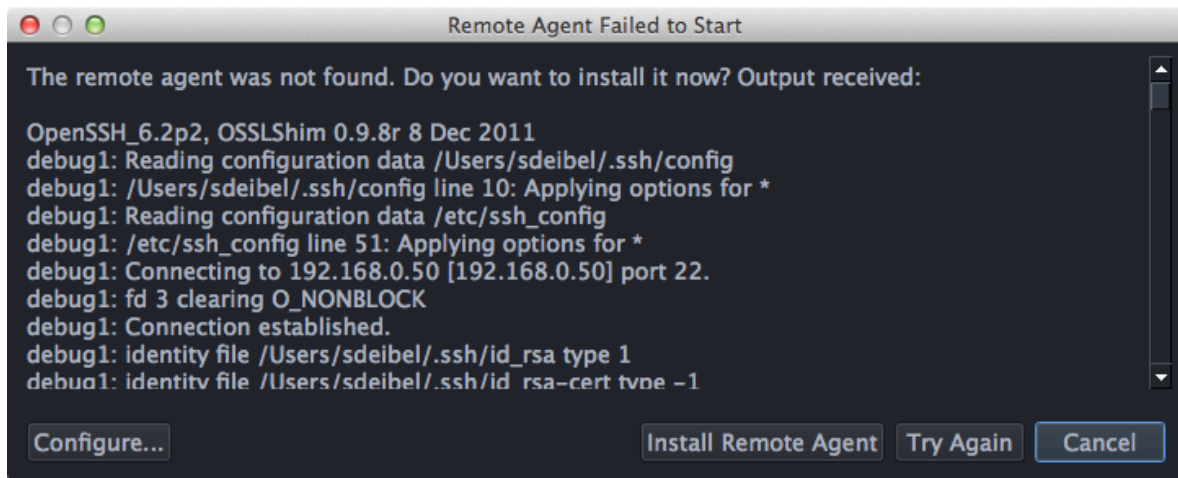
In this case, the output provided may help to diagnose and fix the problem. Or, for help contact support@wingware.com.

Setting up a Project

Now it's time to set up a project that accesses your files on the remote host. To do this, select **New Project** in the **Project** menu, use the **Create Blank Project** option, and then press **Create Project**.

You can also set up a remote host from the **New Project** dialog, but let's use the blank project type now so that you will see where the relevant configuration takes place.

Click **OK** to create the project, and then go into **Project Properties** from the **Project** menu. Set the **Python Executable** to **Remote** and select the remote host you've just configured:
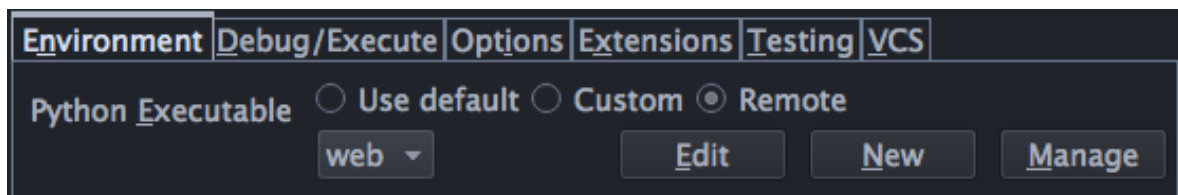
Next add your files to the project with the **Add Existing Directory** item in the **Project** menu. Press the **Browse** button next to the **Directory** field to display a dialog that browses the file system on the remote host. Go into the directory you want to add and press **Select Directory**.

Wing will spend some time scanning the disk and analyzing files but you should already be able to open and edit source files stored on the remote host from the **Project** tool.

Initiating Debug

This How-To assumes you're going to be launching your web server or web framework from outside of Wing and want to debug-enable code that is invoked as you browse your development website. The way Wing does this is by providing a module **wingdbstub.py** that you can import to initiate debug.

When you installed the remote agent above, Wing wrote a preconfigured copy of **wingdbstub.py** to the remote agent installation directory. By default this is **~/.wingpro8/remote-8.3.3.0/wingdbstub.py** where **~** indicates the remote user's home directory. This will vary if you change the **Installation Directory** under the **Advanced** tab of your remote host configuration.

Copy that file to the same directory as your code and add the following to your code before it reaches anything you'll want to debug:

```
import wingdbstub
```

Next tell Wing to listen for connections from an externally launched debug process. This is done by clicking on the bug icon in the lower left of Wing's window and checking on **Accept Debug Connections**:
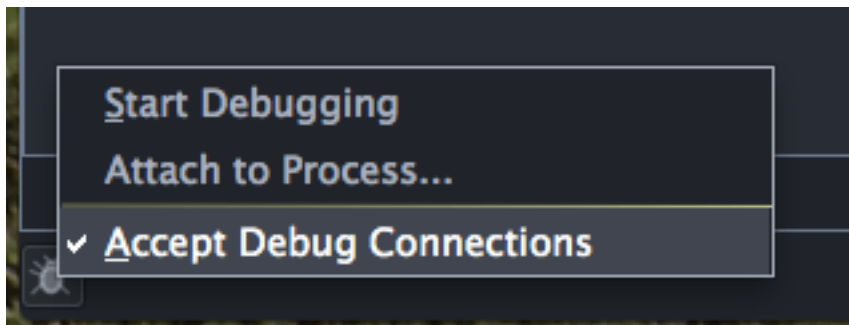
If you hover over the bug icon, Wing shows that it is listening for connections, both on the local host and on the configured remote host:
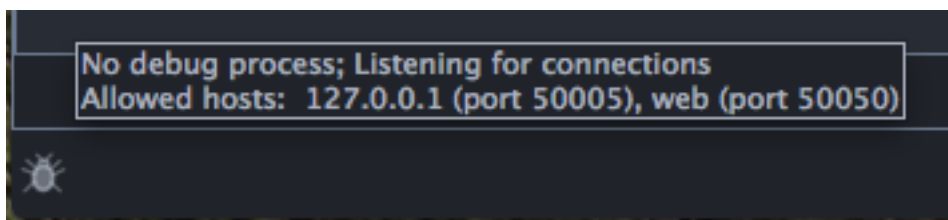
Note: If you are using Apache or otherwise run your code as a user that is different from the one used in your remote host configuration, you'll need to adjust permissions on some files as described in the section **Managing Permissions** below.
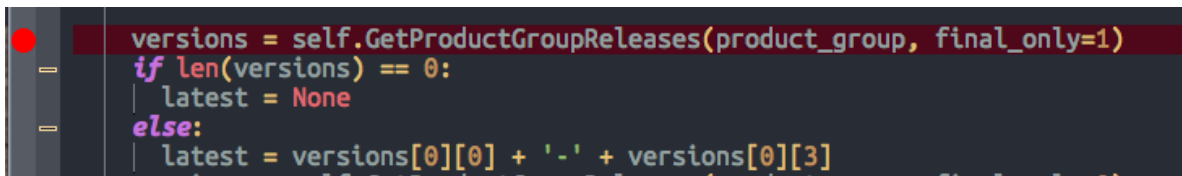
Debugging Code

Now you can set some breakpoints by clicking on the breakpoint margin to the left of your code. For example:
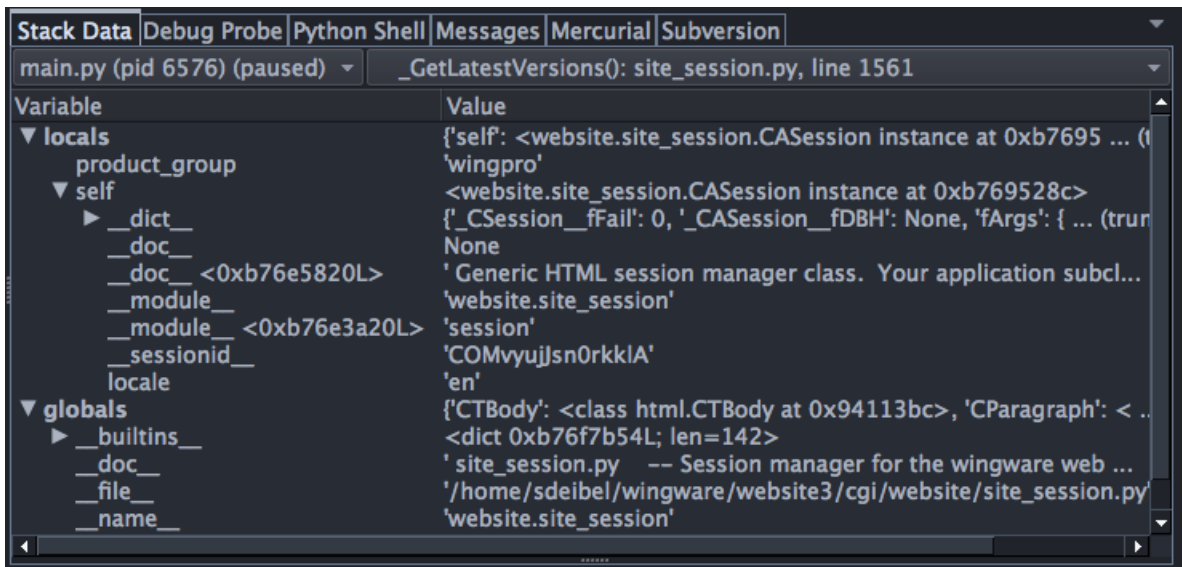
Once this is done you should be able to point your browser at a web page that executes code with a breakpoint, and Wing should stop on it:
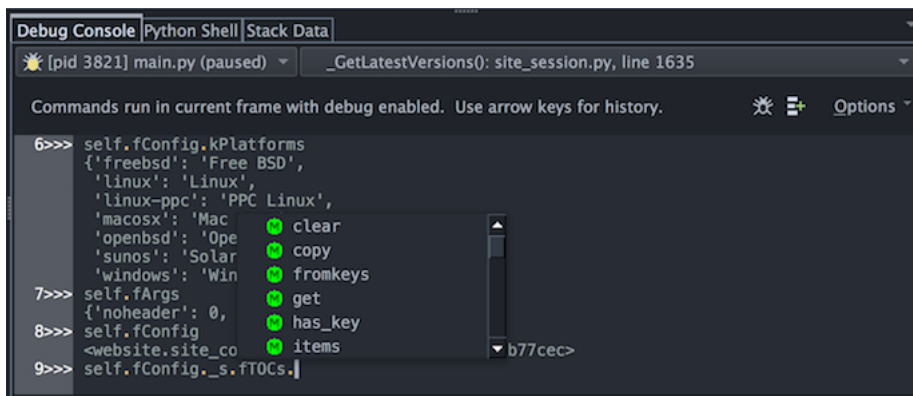
Use **Stack Data** to view debug data:

Or just interact on the command line within the current stack frame in the **Debug Console** tool:

Both of these tools are accessible from Wing's **Tools** menu.

This technique should work with any web development framework. See Web Development How-Tos for details on using Wing with specific frameworks.

Managing Permissions

If your code is running as a different user than the one specified in your remote host configuration, as may be the case if running under Apache or another web server, then you will need to make some additional changes to make remote debugging work. For example, your remote host configuration may set **Host Name** to **devel@192.168.0.50** so the user that installs the remote agent is **devel** while the code is actually run by the user **apache**.

In this case you must change the disk permissions on the **Install Dir** from which you copied **wingdbstub.py** so it can be read by the user that runs your debug process. The best way to do this is to create a group that includes both users and use that group for the directory, for example with **chgrp -R groupname dirname**.

Then change your copy of **wingdbstub.py** by replacing **~** with the full path to the home directory of the user in the remote host configuration. This is needed because **~** will expand to a different directory if the code is run as a different user.

You may also want to change the permissions on the debugger security token file **wingdebugpw** so that both users can read it, for example with **chmod 640 wingdebugpw**. The default for this file is to allow only the owner to read it. If this isn't done, Wing will generate a different debugger security token on the remote host and will initially reject your debug connection and prompt for you to accept the new security token. Once that is done, future debug connections will be accepted.

Resources

- Web Development How-Tos contains instructions for using Wing with specific web development frameworks, such as Django, Flask, Pyramid, web2py, and others.
- Remote Hosts documentation provides details for configuring Wing Pro to work with remote hosts.
- Quick Start provides an introduction to Wing's features.
- Tutorial takes you through Wing's features step by step.

- Wing Reference Manual documents Wing in detail.

3.2. Using Wing with Django

Note

"Wing is really the standard by which I judge other IDEs. It opens, it works, and does everything it can do to stay out of my way so I can be productive. And its remote debugging, which I use when I'm debugging Django uWSGI processes, makes it a rock star!" *-- Andrew M*
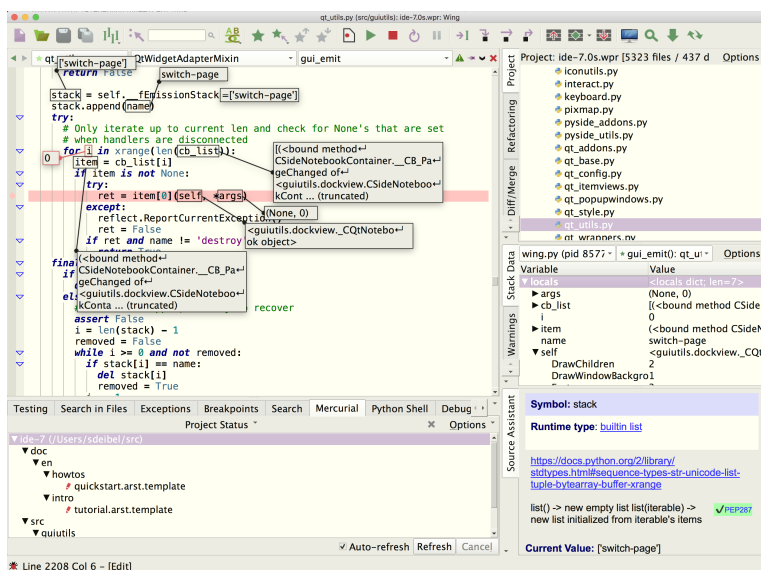
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for the Django web development framework. The debugger works with Django's auto-reload feature and can step through and debug Python code and Django templates. Wing Pro also automates some aspects of the creation and management of Django projects and applications.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Django. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Creating a Project

You can configure a new Wing project to use an existing Django project, or you can create a new Django project at the same time.

Existing Django Project

To set up a Wing Pro project for an *existing* Django project, use **New Project** in the **Project** menu. First select the host you with to work on, then choose your existing source directory, and set **Project Type** to **Django**. If Wing auto-detects a Python environment in your selected source directory, you can immediately create the project. Otherwise, press **Next** to select your Python environment as described below.

New Django Project

If you are starting a *new* Django project at the same time as you are setting up your Wing project, use **New Project** in the **Project** menu. First select the host you wish to work on, then choose a new project directory, and set **Project Type** to **Django**. Then press **Next** to configure your Python environment as described in the previous section.

Selecting the Python Environment

The Python Environment that you use with Django can be any existing Python installation or environment, or you can create a new virtualenv, pipenv, conda env, or Docker container along with your project. See Creating Python Environments for details.

If you select an existing Python environment, be sure that it has Django installed into it before creating your project. If you don't know where your Python is located, run it outside of Wing and type the following:

```
import sys
print(sys.executable)
```

The resulting full path can be used with **Command Line** under the **Existing Python Executable** option in the **New Project** dialog.

Once the project is created, this will display a dialog that confirms the configuration, with a detailed list of the settings that were made.

Now you should be able to start Django in Wing's debugger, set breakpoints in Python code and Django templates, and reach those breakpoints in response to a browser page load.

Usage Tips

Automated Django Tasks

The **Django** menu, shown in Wing when the current project is configured for Django, contains items for common tasks such as creating a new app, generating SQL for a selected app, migrating an app or database, running validation checks or unit tests, and restarting the integrated **Python Shell** with the Django environment.

Wing's Django extensions are open source and can be found in **scripts/django.py** in the install directory listed in Wing's **About** box. For detailed information on writing extensions for Wing, see Scripting and Extending Wing.

Debugging Exceptions

Django contains a catch-all handler that displays exception information to the browser. When debugging with Wing, it is useful to also propagate these exceptions to the IDE. This can be done with a monkey patch as follows (for example, in **local_settings.py** on your development system):

```python
import os
import sys

import django.views.debug

def wing_debug_hook(*args, **kwargs):
    if __debug__ and 'WINGDB_ACTIVE' in os.environ:
        exc_type, exc_value, traceback = sys.exc_info()
        sys.excepthook(exc_type, exc_value, traceback)
    return old_technical_500_response(*args, **kwargs)

old_technical_500_response = django.views.debug.technical_500_response
django.views.debug.technical_500_response = wing_debug_hook
```

The monkey patch only activates if Wing's debugger is active and assumes that the **Debugger > Exceptions > Report Exceptions** preference is left set to its default value **When Printed**.

Template Debugging

Wing Pro allows you to set breakpoints in any file that contains **{%%}** or **{{}}** tags, and the debugger will stop at them.

Note that stepping is tag by tag and not line by line, but breakpoints are limited to being set for a particular line and thus match all tags on that line.

When template debugging is enabled, you won't be able to step into Django internals during a template invocaton. To work around that, temporarily uncheck **Enable Django Template Debugging** under the **Extension** tab of Project Properties in Wing, and then restart your debug process.

Better Auto-Completion

Wing provides auto-completion on Python code and Django templates. The completion information is based on static analysis of the files and runtime introspection if the debugger is active and paused. It is often more informative to work with the debugger paused or at a breakpoint, particularly in Django templates where static analysis is not as effective as it is in Python code.

Running Unit Tests

Wing Pro includes a unit testing integration capable of running and debugging Django unit tests. For Django projects, the **Default Testing Framework** under the **Testing** tab of **Project Properties** is set to **Django Tests** so that the **Testing** tool runs **manage.py test** and displays the results. Individual tests can be run or debugged by selecting them and pressing **Run Tests** or **Debug Tests** in the **Testing** tool.

If unit tests need to be run with different settings, the environment variable **WING_TEST_DJANGO_SETTINGS_MODULE** can be set to replace **DJANGO_SETTINGS_MODULE** when unit tests are run.

Related Documents

For more information see:

- Django home page provides downloads and documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.
- Remote Web Development describes how to set up development to a remote host, VM, or container.

3.3. Using Wing with Flask

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for the Flask web development framework.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Flask. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Project Configuration

To create a new project, use **New Project** in Wing's **Project** menu. Choose **Use Existing Directory** or **Create New Directory** to select or create your source directory and then choose the **Project Type** of **Flask**. Press **Next** and then select or create the Python environment to use with your new project.

Using an Existing Python Installation or Environment: Select **Use Existing Python** and **Command Line** to use an existing Python installation or environment. This should be set to the value of **sys.executable** (after **import sys**) from the Python you wish to use.

Or, if you are using Flask in an existing virtualenv or Anaconda environment, select **Activated Env** instead and enter the command that activates the environment (for example, **c:\path\to\env\Scripts\activate.bat**, **/path/to/env/bin/activate**, or Anaconda's **activate env**). The drop down menu to the right of this field lists recently used and automatically found environments.

Note that using an activate command whose full path contains spaces will not work. In this case, use the **Command Line** option as described above.

When an existing Python installation or environment is used, Flask is assumed to already be installed into it. If this is not the case, you can install it from the **Packages** tool after your project has been set up.

Creating a New Environment: You can also create a new virtualenv, pipenv, conda, or Docker environment from here along with your project, as described in more detail in Creating Python Environments.

When you have finished selecting your Python environment, press **Create Project** to create your new project. If you are using an existing source directory, you will need to save the project to disk using **Save Project** in the **Project** menu. If you created a new source directory, the project will automatically be stored there.

Remote Hosts and VMs

Wing Pro can work with Flask code that is running on a remote host or VM. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, after first setting the **Host** in the **New Project** dialog. See Remote Hosts for more information on remote development with Wing Pro.

Containers

For containers, select **Local Host** on the first page of the **New Project** dialog, then **Create New Environment** on the second page, and set up a new **Docker** container. For details, see Using Wing Pro with Docker.

Port Forwarding

When a remote host or container is used with your project, Wing sets up port forwarding in the remote host or container configuration created with your project, so that you can connect to the Flask instance using a browser running on the same host as the IDE.

For example, if Flask is listening on port 8000 on the remote host or container, Wing forwards port 8000 on localhost to the remote system, and the url you will use to access Flask from the host where Wing is running is: **http://localhost:8000**.

You can view and alter the port forwarding configuration on the **Options** page of the remote host or container configuration.

Debugging Flask in Wing

If you used New Project to start a new Flask project, Wing will have written a starter **app.py** that is already properly configured for debugging. If you are starting to use Wing with an existing Flask source base, you will need to turn off Flask's built-in debugger, so that Wing's debugger can take over reporting exceptions.

To do this, you can set up your main entry point as in the following example:

```
from flask import Flask
app = Flask(__name__)

...

if __name__ == "__main__":
```

```
    import os
    if 'WINGDB_ACTIVE' in os.environ:
        app.debug = False
    app.run()
```

Notice that this turns off Flask's debugging support only if Wing's debugger is present.

Once this is done, use **Set Current as Main Entry Point** in the **Debug** menu to set this file as your main entry point. Then you can start debugging from the IDE, see Flask's output in the **Debug I/O** tool, and load pages from a browser to reach breakpoints or exceptions in your code.

Use **Restart Debugging** in the **Debug** menu or the restart icon in the toolbar to quickly restart Flask after making changes to your code. Or you can automate this as described in the next section.

Setting up Auto-Reload with Wing Pro

Auto-reload is set up automatically by Wing if **New Project** is used to start a new **Flask** project.

If you are working with an existing code base, you can configure it to auto-reload changed code adding **use_reloader=True** to your **app.run()** call, as follows:

```
app.run(use_reloader=True)
```

Then enable **Debug Child Processes** under the **Debug/Execute** tab in **Project Properties** from the **Project** menu. This tells Wing Pro to debug also child processes created by Flask, including the reloader process.

Now Flask will automatically restart on its own whenever you save an already-loaded source file to disk.

You can add additional files for Flask to watch as follows:

```
watch_files = ['/path/to/file1', '/path/to/file2']
app.run(use_reloader=True, extra_files=watch_files)
```

Whenever any of these additional files changes, Flask will also automatically restart.

Related Documents

For more information see:

- Flask home page provides links to documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

3.4. Using Wing with Pyramid
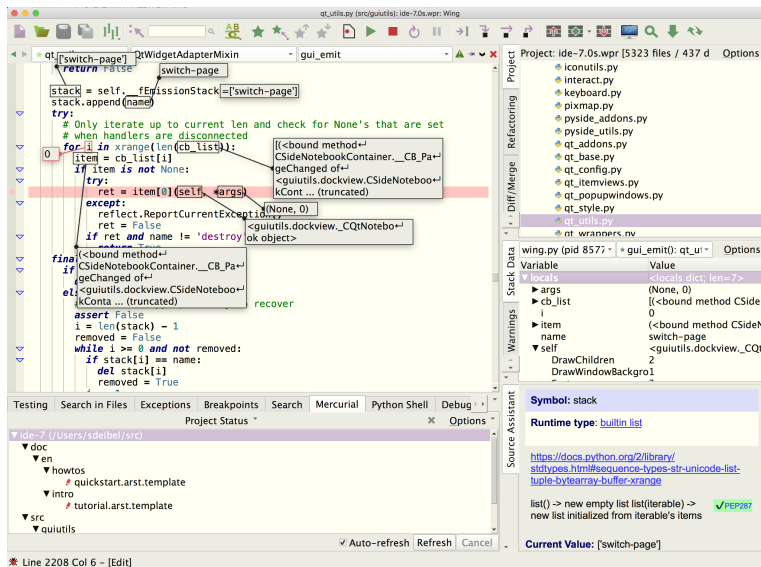
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for the Pyramid web development framework.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Pyramid. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Creating a Wing Project

To create a new project, use **New Project** in Wing's **Project** menu with **Project Type** set to **Pyramid**. You'll be able to select or create a source directory for your project and select or create a Python environment. See Creating a Project for details on creating projects in Wing.

Debugging

Launching from Wing

The easiest way to debug Pyramid is just to launch it from Wing. To do this, find and open **pserve** from Pyramid and select **Set Current as Main Entry Point** from the **Debug** menu.

Then right-click on **pserve** and under **Environment** enter your run arguments, for example:

```
development.ini
```

Then go into **Project Properties** in the **Project** menu and set **Initial Directory** under the **Debug/Execute** tab to the full path of the directory that contains your **.ini** files.

Now you can start debugging with **Start/Continue** in the **Debug** menu or from the toolbar. You can load http://localhost:6543/ or other page, or initiate an AJAX request, and Wing will stop on any

breakpoints or exceptions. This works with any Python code, including any View Callable, Pyramid internals, or any other library or package used by your code.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

Auto-reloading Changes

With the above configuration, you'll need to restart Pyramid every time you make a change. If you have Wing Pro you can cause Pyramid to auto-reload changes. To do this, add the **--reload** option to the run arguments you set for **pserve**, for example:

```
--reload development.ini
```

Then enable **Debug Child Processes** in **Project Properties**, from the **Project** menu, so that Pyramid's reloaded processes will also be debugged.

This option is only available in Wing Pro. In Wing Personal you'll need to use **wingdbstub** for reloading, as described below.

Launching Outside of Wing

Wing can also debug code that is launched from outside of the IDE, for example from the command line. To do this with Pyramid, copy **wingdbstub.py** from the **Install Directory** listed in Wing's **About** box into the directory that contains your Pyramid **.ini** files. You may need to set the value of **WINGHOME** inside your copy of this file to the full path of the install directory you copied it from, or on macOS to the full path of the **.app**.

Next place the following line into your source, on a line before the code you wish to debug:

```
import wingdbstub
```

Then click on the bug icon in the lower left of Wing's window and make sure that **Accept Debug Connections** is checked.

Now you can start your Pyramid server as you usually would, for example:

```
pserve --reload development.ini
```

Using **--reload** is not necessary but it is supported by Wing's debugger and makes testing of changes much easier.

Notes on Auto-Completion

Wing provides auto-completion on Python code and in other files, including the various templating languages that can be used with Pyramid.

The autocomplete information available to Wing is based on static analysis of your project files and any files Wing can find through imports, by searching on your Python Path.

When the debugger is active and paused, Wing also uses introspection of the live runtime for any template or Python code that is active on the stack. As a result, it is often more informative to work on your source files while Wing's debugger is active and paused at a breakpoint, exception, or anywhere in the source code reached by stepping.

Debugging Jinja2 Templates

The Jinja2 template engine sets up stack frames in a way that makes it possible to set breakpoints directly in **.jinja2** template files and step through them, viewing data in **Stack Data** and other tools in the same way as for Python code.

Debugging support in the Jinja engine is imperfect in that that not all tags are reached and some tags cause lines to be visited multiple times. However, this capability can still be useful to stop Wing's debugger when a particular template is being invoked.

Debugging Mako Templates

Another good choice of templating engine for Pyramid is Mako, because it allows the full syntax of Python in expression substitutions and control structures. However, Mako templates cannot be directly stepped through using the debugger. Instead, you can set breakpoints in the **.py** files produced by Mako for templates.

To debug Mako templates with Wing you will need to modify your **.ini** file to add the following line in the **[app:main]** section:

```
mako.module_directory=%(here)s/data/templates
```

You may need to change the path to match your project. Without this setting, mako templates are compiled in memory and not cached to disk, so you won't be able to debug them. With this setting, Mako will write **.mako.py** files for each template to the specified directory, whenever the template changes. You can set breakpoints within these generated files.

Your **.mako.py** files will not be in one-to-one line correspondence with their **.mako** source files, but mako inserts tracking comments indicating original source line numbering.

If you are starting Pyramid outside of Wing and need to use **wingdbstub** to initiate debugging, as described earlier, and want to do this from a Mako template, then you can add the following to the template:

```
<%! import wingdbstub %>
```

Remote Development

Wing Pro can work with Pyramid code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See Remote Hosts for more information on remote development with Wing Pro.

Related Documents

For more information see:

- Pyramid documentation
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

3.5. Using Wing with web2py
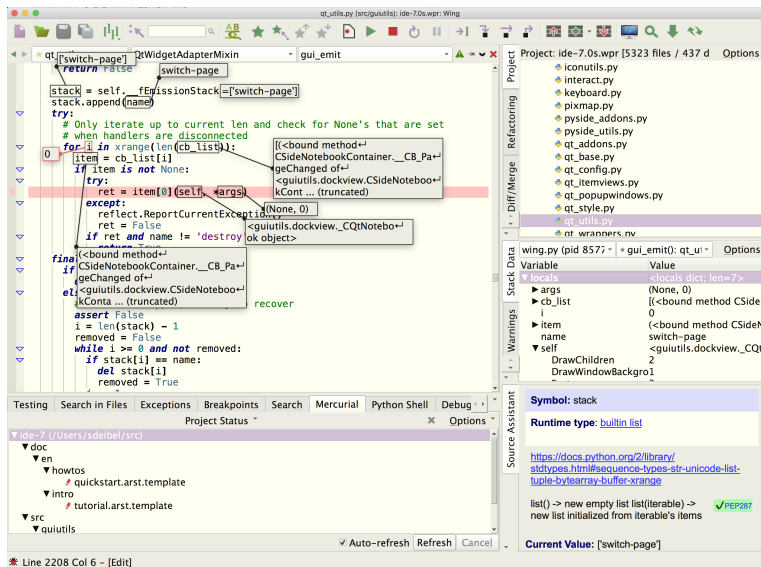
[Wing Pro](#) is a Python IDE that can be used to develop, test, and debug Python code and templates written for [web2py](#), a powerful open source web development framework.

If you do not already have Wing Pro installed, [download it now](#).

This document describes how to configure Wing for web2py. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

Introduction

Wing allows you to debug Python code and templates running under web2py as you interact with it from your web browser. Breakpoints set in your code from the IDE will be reached, allowing inspection of your running code's local and global variables with Wing's various debugging tools. In addition, in Wing Pro, the **Debug Console** allows you to interactively execute methods on objects and get values of variables that are available in the context of the running web app.

There is more than one way to do this, but in this document we focus on an "in process" method where the web2py server is run from within Wing, as opposed to [attaching to a remote process](#).

Setting up a Project

The best way to install web2py is to clone the [git repository](#). Be sure to follow the instructions in the readme so you clone all the dependencies recursively.

To create a new project, use **New Project** in Wing's **Project** menu with **Project Type** set to **web2py**. You'll be able to select or create a source directory for your project and select or create a Python environment. See [Creating a Project](#) for details on creating projects in Wing.

After the pressing **Create Project**, open the **Project** tool from the **Project** menu. From there, find and right click on the file **web2py.py** and select **Set As Main Entry Point**.

Remote Development

Wing Pro can work with web2py code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See Remote Hosts for more information on remote development with Wing Pro.

Debugging

You can now debug web2py by clicking on the green **Debug** icon in Wing's toolbar and waiting for the web2py console to appear. Enter a password and start the server as usual.

Once web2py is running, open a file in Wing that you know will be reached when you load a page of your web2py application in your web browser. Place a breakpoint in the code and load the page in your web browser. Wing should stop at the breakpoint. Use the **Stack Data** tool or **Debug Console** (in Wing Pro) to look around.

An example is to set a breakpoint in **applications/examples/views/default/index.html**, which is loaded when you go to the URL **http://127.0.0.1:8000/examples/default/index** (assuming local web2py install running on port 8000).

Notice that breakpoints work both in Python code and HTML template files.

Wing's **Debug Console** (in the **Tools** menu) is similar to running a shell from web2py (with **python web2py.py -S myApp -M**) but additionally includes your entire context and provides auto-completion. You can easily inspect or modify variables, manually make function calls, and continue debugging from your current context.

Usage Tips

Setting Run Arguments

When you start debugging, Wing will show the **File Properties** for **web2py.py**. This includes a **Run Arguments** field under the **Debug** tab where you can add any web2py option. For example, adding **-a '<recycle>'** will give you somewhat faster web2py startup since it avoids showing the **Tk** dialogs and automatically opening a browser window. This is handy once you already have a target page in your browser. Run **python web2py.py --help** for a list of all the available options.

To avoid seeing the **File Properties** dialog each time you debug, un-check the "Show this dialog before each run" check box. You can access it subsequently with **Debug Environment** in the **Debug** menu.

Hung Cron Processes

Web2py may spawn cron sub-processes that fail to terminate on some OSes when web2py is debugged from Wing. This can lead to unresponsiveness of the debug process until those sub-processes are killed. To avoid this, add the parameter **-N** to prevent the cron processes from being spawned.

Better Auto-completion

Because of the way web2py is designed, Wing's static analysis engine can fail to find the types of commonly used values like **db**. To work around this, run to a breakpoint in your code before editing it. This causes Wing to use runtime analysis as well as static analysis to drive auto-completion and other IDE features.

Related Documents

Wing provides many other options and tools. For more information:

- web2py website provides documentation and downloads.
- Remote Web Development describes how to set up development on a remote host, VM, or container.
- **Quickstart Guide which contains additional basic information**
   about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual, which describes Wing in detail.

3.6. Using Wing with mod_wsgi
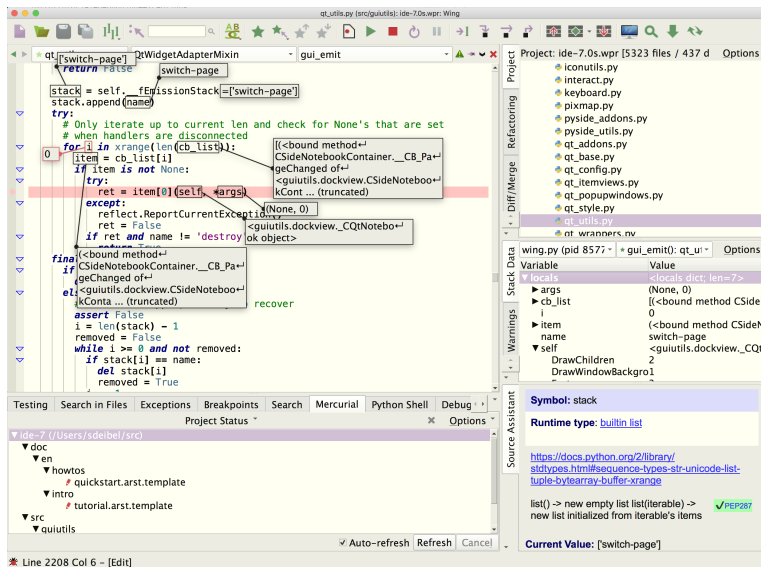
[Wing Pro](#) is a Python IDE that can be used to develop, test, and debug Python code that is running under [mod_wsgi](#).

If you do not already have Wing Pro installed, [download it now](#).

This document describes how to configure Wing for mod_wsgi. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

Debugging Setup

When debugging Python code running under **mod_wsgi**, the debug process is initiated from outside of Wing, and must connect to the IDE. This is done with **wingdbstub** according to the instructions in [Debugging Externally Launched Code](#).

Because of how **mod_wsqi** sets up the interpreter, be sure to set **kEmbedded=1** in your copy of **wingdbstub.py** and use the debugger API to reset the debugger and connection as follows:

```
import wingdbstub
wingdbstub.Ensure()
```

Then click on the bug icon in lower left of Wing's window and make sure that **Accept Debug Connections** is checked. After that, you should be able to reach breakpoints by loading pages in your browser.

Disabling stdin/stdout Restrictions

In order to debug, may also need to disable the WSGI restrictions on stdin/stdout with the following mod_wsgi configuration directives:

```
WSGIRestrictStdin Off
WSGIRestrictStdout Off
```

Remote Development

Wing Pro can work with mod_wsgi code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See Remote Hosts for more information on remote development with Wing Pro.

Related Documents

For more information see:

- mod_wsgi website for downloads and documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

How-Tos for GUI Development

The following How-Tos explain how to get started using Wing with a number of popular GUI development frameworks.

4.1. Using Wing with wxPython
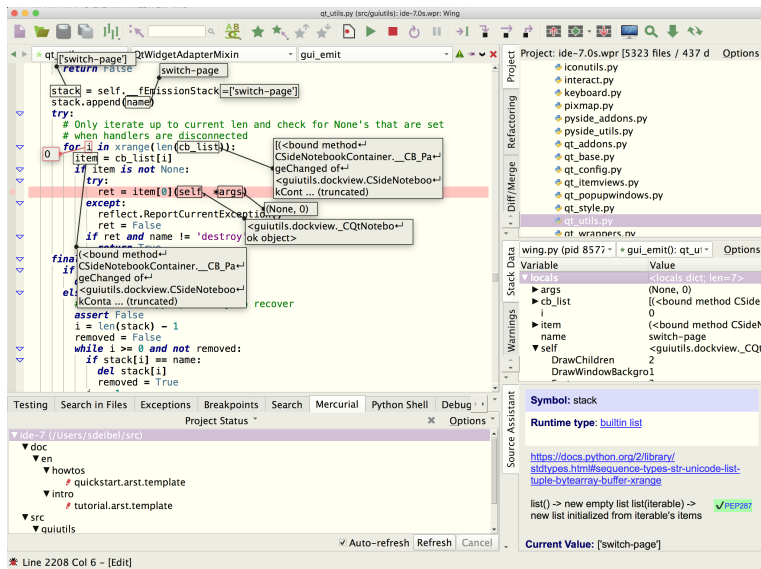
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for the wxPython cross-platform GUI development toolkit.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for wxPython. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Introduction

wxPython is a good choice for desktop application developers that want to use Python. It is available for Windows, Linux, and macOS and provides native look and feel on each of these platforms.

While Wing does not provide a GUI builder for wxPython, it does provide advanced editing, debugging, testing, and code inspection capabilities for Python, and it can be used with other available GUI builders, as described below.

Installation and Configuration

Take the following steps to set up and configure Wing for use with wxPython:

- Install wxPython as described on the wxPython Downloads page. Be sure to install also the wxPython demo and samples.
- Install Wing if you don't already have it.
- Start Wing from the Start menu on Windows, the Finder or macOS, or by typing **wing8** on the command line on Linux.
- Select **Show Python Environment** from the **Source** menu. If the Python version reported there doesn't match the one you're using with wxPython, then select **Project Properties** from the **Project** menu and set **Python Executable**.

- Locate and open wxPython's **demo.py** into Wing and then select **Add Current File** from the **Project** menu to add it to your project. If you can't find **demo.py** but have other wxPython code that works, you can just use that. However, you'll need to adapt the instructions in the rest accordingly.
- Set **demo.py** as main entry point for debugging using the **Set Current as Main Entry Point** item in the **Debug** menu.
- Save your project to disk. Use a name ending in **.wpr**.

Test Driving the Debugger

Now you're ready to try out the debugger:

Start debugging with the **Start / Continue** item in the **Debug** menu. Uncheck the **Show this dialog before each run** checkbox at the bottom of the dialog that appears and select **OK**.

The demo application will start up. If its main window doesn't come to front, bring it to front from your task bar or window manager. Try out the various demos from the tree on the left of the wxPython demo app.

Next open **ImageBrowser.py**, located in the same directory as **demo.py**. Set a breakpoint on the first line of **runTest()** by clicking on the dark grey left margin. Go into the running demo app and select More Dialogs / ImageBrowser. Wing will stop on your breakpoint.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

See the Wing Tutorial and Quick start for more information.

Using a GUI Builder

Wing doesn't include a GUI builder for wxPython but it can be used with other tools, such as wxGlade or wxFormBuilder. Wing will automatically reload files that are generated by the GUI builder.

Related Documents

Wing provides many other options and tools. For more information:

- wxPython website
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

4.2. Using Wing with PyQt

Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for the PyQt cross-platform GUI development toolkit.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for PyQt. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Introduction

PyQt is a commercial GUI development environment that runs with native look and feel on Windows, Linux, Mac OS, and mobile devices.

While Wing does not provide a GUI builder for PyQt, it does provide advanced editing, debugging, testing, and code inspection capabilities for Python, and it can be used with other available GUI builders, as described below.

These instructions should also work with PySide, which are roughly comparable non-commercial open source bindings for Qt.

Installation and Configuration

Take the following steps to set up and configure Wing for use with PyQt:

- Install PyQt as described in Installing PyQt5. Be sure to install also the **qtdemo**.
- Install Wing if you don't already have it.
- Start Wing from the Start menu on Windows, the Finder or macOS, or by typing **wing8** on the command line on Linux.

- Select **Show Python Environment** from the **Source** menu. If the Python version reported there doesn't match the one you're using with PyQt, then select **Project Properties** from the **Project** menu and set **Python Executable**.
- Locate and open **qtdemo.py** into Wing and then select **Add Current File** from the **Project** menu to add it to your project. If you can't find **qtdemo.py** but have other PyQt code that works, you can just use that. However, you'll need to adapt the instructions in the rest accordingly.
- Set **qtdemo.py** as main entry point for debugging with **Set Current as Main Entry Point** in the **Debug** menu.
- Save your project to disk. Use a name ending in **.wpr**.

Test Driving the Debugger

Now you're ready to try out the debugger:

Start debugging with the **Start / Continue** item in the **Debug** menu. Uncheck the **Show this dialog before each run** checkbox at the bottom of the dialog that appears and select **OK**.

The demo application will start up. If its main window doesn't come to front, bring it to front from your task bar or window manager.

Next locate and open **menumanager.py** in the **qtdemo** directory and set a breakpoint on the first line of the method **itemSelection**. Once set, this breakpoint should be reached whenever you click on a button in the **qtdemo** application.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

See the Wing Tutorial and Quick start for more information.

Using a GUI Builder

Wing doesn't include a GUI builder for PyQt but it can be used with an external GUI builder like Qt Designer. Wing will automatically reload files that are generated by the GUI builder.

Related Documents

For more information see:

- PyQt home page, which provides links to documentation and downloads.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

4.3. Using Wing with GTK and PyGObject
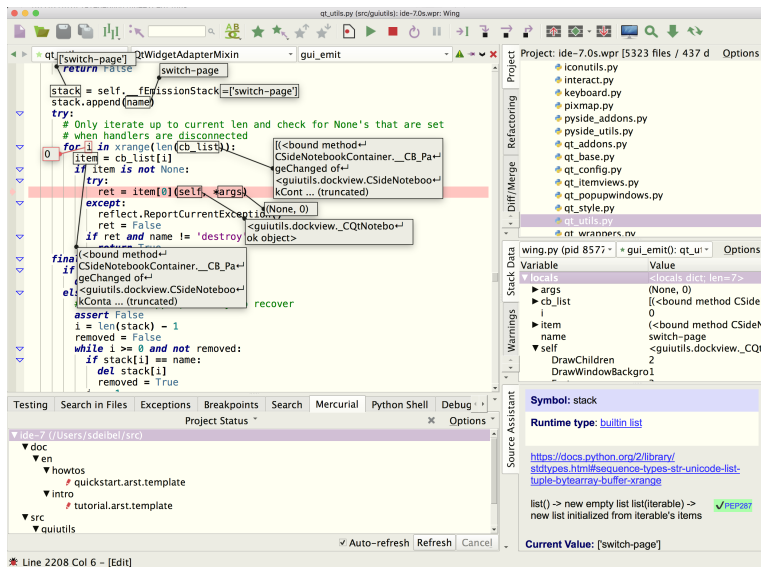
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for GTK using PyGObject.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for GTK and PyGObject. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Introduction

PyGObject implements Python bindings for GTK, an open source GUI development toolkit.

While Wing does not provide a GUI builder for GTK, it does provide advanced editing, debugging, testing, and code inspection capabilities for Python, and it can be used with other available GUI builders, as described below.

Installation and Configuration

Take the following steps to set up and configure Wing for use with PyGObject:

- Install PyGObject as described in the PyGObject documentation.
- Install Wing if you don't already have it.
- Start Wing from the Start menu on Windows, the Finder or macOS, or by typing **wing8** on the command line on Linux.
- Select **Show Python Environment** from the **Source** menu. If the Python version reported there doesn't match the one you're using with PyGObject, then select **Project Properties** from the **Project** menu and set **Python Executable**.
- Locate and open the Python main entry point for your PyGObject-based application and then select **Add Current File** from the **Project** menu to add it to your project.

- Set your Python main entry point for debugging with **Set Current as Main Entry Point** in the **Debug** menu.
- Save your project to disk. Use a name ending in **.wpr**.

Test Driving the Debugger

Now you're ready to try out the debugger:

Start debugging with the **Start / Continue** item in the **Debug** menu. Uncheck the **Show this dialog before each run** checkbox at the bottom of the dialog that appears and select **OK**.

Your application should start up. If its main window doesn't come to front, bring it to front from your task bar or window manager.

Next locate and open Python source code that you know will be reached when you use your application and set a breakpoint by clicking on the margin to the left of the code. Then trigger the breakpoint by performing an action in your application that results in execution of the code at that line.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

See the Wing Tutorial and Quick start for more information.

Improving Auto-Completion

PyGObject uses lazy (on-demand) loading of functionality to speed up startup of applications that are based on it. This prevents Wing's analysis engine from inspecting PyGObject-wrapped APIs and thus the IDE fails to offer auto-completion.

To work around this, use Fakegir, which is a tool to build a fake Python package of PyGObject modules that can be added to the **Source Analysis > Advanced > Interface File Path** preference. The parent directory of the generated gi package should be added; if the defaults are used, the directory to add is **~/.cache/fakegir**.

Fakegir's **README.md** provides usage details.

Don't add the Fakedir produced package to the **Python Path** defined in Wing's **Project Properties** because code will not work if the fake module is actually on **sys.path** when importing any PyGObject-provided modules.

Once this is done Wing should offer auto-completion for all PyGObject-provided modules and should be able to execute and debug your code without disruption.

Using a GUI Builder

Wing doesn't include a GUI builder for PyGObject but it can be used with an external GUI builder like Glade. Wing will automatically reload files that are generated by the GUI builder.

Related Documents

For more information see:

- GTK using PyGObject websites.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

How-Tos for Modeling, Rendering, and Compositing Systems

The following How-Tos explain how to get started using Wing with modeling, rendering, and compositing systems that use Python for game development and 2D and 3D animation.

5.1. Using Wing with Blender

[Wing Pro](#) is a Python IDE that can be used to develop, test, and debug Python code written for [Blender](#), an open source 3D content creation system.

If you do not already have Wing Pro installed, [download it now](#).

This document describes how to configure Wing for Blender. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

Working with Blender

Blender loads Python scripts in a way that makes them difficult to debug in a Python debugger. The following stub file can be used to work around these problems:

```python
import os
import sys

# MODIFY THESE:
winghome = r'c:\Program Files (x86)\Wing Pro 8'
scriptfile = r'c:\src\test\blender.py'

os.environ['WINGHOME'] = winghome
if winghome not in sys.path:
    sys.path.append(winghome)
#os.environ['WINGDB_LOGFILE'] = r'c:\src\blender-debug.log'
import wingdbstub
wingdbstub.Ensure()

def runfile(filename):
    if sys.version_info < (3, 0):
        execfile(filename)
    else:
        import runpy
```

```
        runpy.run_path(filename)

runfile(scriptfile)
```

To use this script:

1. Modify **winghome** & **scriptfile** definitions where indicated to the wing installation directory and the script you want to debug, respectively. When in doubt, the location to use for **winghome** is given as the **Install Directory** in Wing's About box (accessed from **Help** menu).
2. Run blender
3. Press **Shift-F11** to display the text editor
4. Press **Alt-O** to browse for a file and select this file to open

Once the above is done you can debug your script by executing this blender stub file in blender. This is done using the **Run Script** button on the bottom toolbar or by pressing **Alt-P**, although note that **Alt-P** is sensitive to how the focus is set.

Note that you will need to turn on listening for externally initiated debug connections in Wing, by clicking on the bug icon in the lower left of the main window and selecting **Accept Debug Connections** in the popup menu that appears.

Related Documents

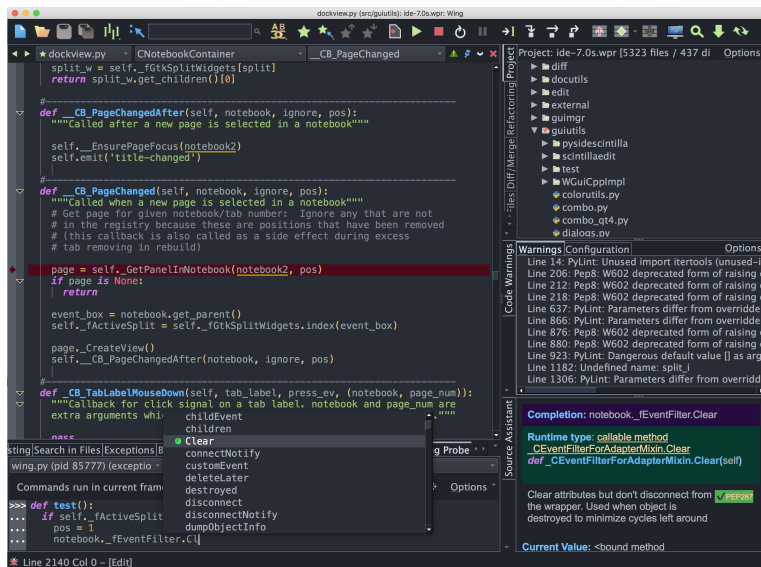For more information see:

- Blender website
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

5.2. Using Wing with Autodesk Maya
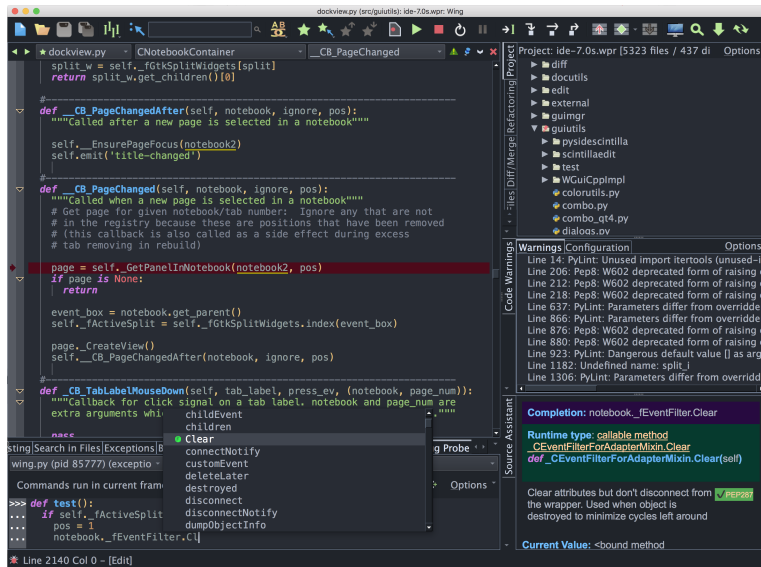
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Autodesk Maya, a commercial 3D modeling application.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Maya. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Debugging Setup

When debugging Python code running under **Maya**, the debug process is initiated from outside of Wing, and must connect to the IDE. This is done with **wingdbstub** according to the detailed instructions in the Debugging Externally Launched Code section of the manual. In summary, you will need to:

1. Copy **wingdbstub.py** from your Wing installation into a directory that will be on the **sys.path** when Python code is run by Maya. You may need to inspect that (after **import sys**) first from Maya, or you can add to the path with **sys.path.append()** before importing **wingdbstub**.
2. Because of how **Maya** sets up the Python interpreter, be sure to set **kEmbedded=1** in your copy of **wingdbstub.py**
3. In your code, where you initiate debugging, use the debugger API to ensure the debugger is connected to the IDE before any other code executes, as follows:

```
import wingdbstub
wingdbstub.Ensure()
```

4. In some cases you may need to edit **wingdbstub.py** to set **WINGHOME** to point to the directory where Wing is installed. This is usually set up automatically by Wing's installer, but

won't be if you are using the **.zip** installation of Wing. Note that if you edit **wingdbstub.py** after Maya has already imported it then you will need to restart Maya to get it to import the modified **wingdbstub**.

5. Then click on the bug icon in lower left of Wing's window and make sure that **Accept Debug Connections** is checked.

At this point, you should be able to reach breakpoints by causing the scripts to be invoked from Maya. In Maya 2018 at least, running a script does not set up the file name in the compiled Python code correctly, so breakpoints only work in modules that are imported into your top-level script. Breakpoints in the main script may work in older Maya versions.

Once debugging starts, when a breakpoint or exception is reached, Wing should come to the front and show the place where the debugger stopped. Although the code is running inside Maya, editing and debugging happens inside Wing.

Avoiding Crashing in Maya 2020

Maya 2020 may use an old copy of OpenSSL that leads to crashing on 10th (Ice Lake), 11th (Rocket Lake) or 12th (Alder Lake) generation Intel CPUs. The problem occurs when **import wingdbstub** is reached, crashing Maya completely.

There are two ways to work around this problem:

1) Uncheck the **Debugger > Network > Use Digests to Identify Files** preference in Wing. This turns off a part of Wing's debugger implementation that calls **hashlib**, and thus avoids the crash.

-or-

2) Set Windows system environment variable **OPENSSL_ia32cap** to the value **~0x20000000** before starting Maya.

Using Maya's Python in Wing

You can use the **mayapy** executable found in the **Maya** application directory to run Wing's **Python Shell** tool and to debug standalone Python scripts.

To do this, select **Command Line** for **Python Executable** in **Project Properties**, accessed from the **Project** menu, and then enter the full path of the **mayapy** file (**mayapy.exe** on Windows).

Better Static Auto-completion

Setting **Python Executable** in Wing's **Project Properties**, as described above, is also needed to obtain auto-completion for Maya's Python API.

At least in some versions of Maya, Wing cannot statically analyze the files in the Python API without some additional configuration. As a result, it will fail to offer auto-completion for the API. The solution to this depends on the version of Maya.

Maya 2020

In Maya 2020 it is necessary to download and install the Maya 2020 devkit from the devkit downloads listed on the Autodesk website. The **pi** interface files will be located in **devkitBase\devkit\other\pymel\extras\completion\pi** inside your Maya 2020 installation directory. This can be added to the Source Analysis > Advanced > **Interface File Path** preference in Wing.

Maya 2018

Maya 2018 ships with .pi files in the **devkit/pymel/extras/completion/pi** subdirectory of the Maya 2018 install directory. This can be added to the Source Analysis > Advanced > **Interface File Path** preference in Wing.

Maya 2016

Maya 2016 is missing necessary developer files so you will need to download and install the Maya 2016 devkit which should create **devkit\other\pymel\extras\completion\py\maya\api** in your Maya installation. This can then be used by making the following edits:

- In "OpenMaya.py" add **from _OpenMaya_py2 import ***
- In "OpenMayaAnim.py" add **from _OpenMayaAnim_py2 import ***
- In "OpenMayaRender.py" add **from _OpenMayaRender_py2 import ***
- In "OpenMayaUI.py" add **from _OpenMayaUI_py2 import ***

This method is based on this forum post.

Instead of editing files in the Maya installation, it is also be possible to add **.pi** files with the added source. For example, placing **OpenMaya.pi** with contents **from _OpenMaya_py2 import *** in the same directory as **OpenMaya.py** causes Wing to merge the analysis of the ***.pi** file with what is found in the ***.py** file.

Alternatively, place these files in another directory that is added to the **Source Analysis > Advanced > Interface File Path** preference in Wing.

You will also want to set the **Python Executable** in Wing's **Project Properties** to **Command Line** and then enter the full path to **mayapy.exe** so that the API is on the Python Path and you are using the correct version of Python.

Maya 2011+

Maya 2011+ before 2016 also shipped with .pi files that can be used as described for Maya 2018 above.

Older Versions

For older Maya versions, .pi files from the PyMEL distribution at http://code.google.com/p/pymel/ may be used. Just unpack the distribution and add **extras/completion/pi** to the **Source Analysis > Advanced > Interface File Path** preference in Wing.

Additional Information

Some additional information about using Wing with Maya can be found in the mel wiki under the **wing** tag.

See also the section **Using Wing with Maya** in Autodesk Maya Online Help: Tips and tricks for scripters new to Python.

Related Documents

For more information see:

- Autodesk Maya website
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

5.3. Using Wing with NUKE and NUKEX
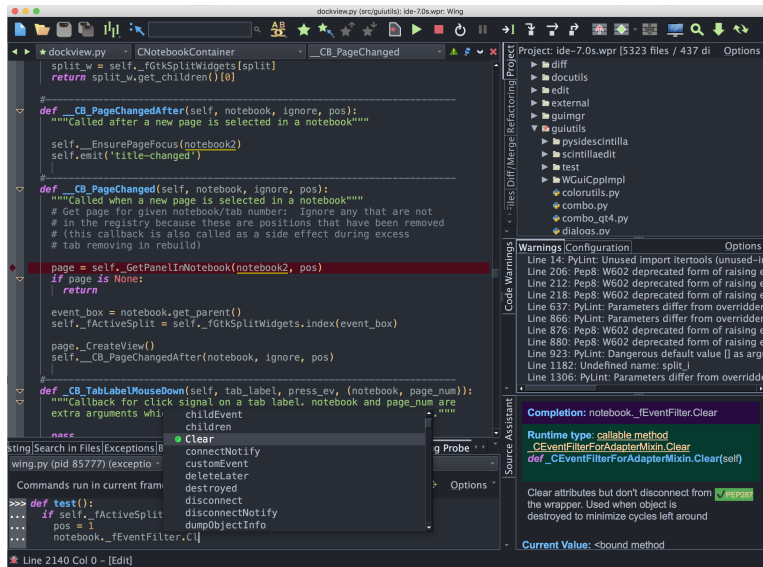
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for The Foundry's NUKE and NUKEX digital compositing tool.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for NUKE and NUKEX. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Project Configuration

First, launch Wing and create a new project from the **Project** menu. Although not required for working with NUKE, you may want to select your existing source directory or add it after project creation with **Add Existing Directory** in the **Project** menu. Doing this tells Wing's source analysis, search, and revision control features know which files are part of the project.

Configuring for Licensed NUKE/NUKEX

If you have NUKE or NUKEX licensed and are not using the Personal Learning Edition, then you can create a script to run NUKE's Python in terminal mode and use that as the **Python Executable** in Wing's Project Properties. For example on macOS create a script like this:

```
#!/bin/sh
/Applications/Nuke6.3v8/Nuke6.3v8.app/Nuke6.3v8 -t -i "$@"
```

Then perform **chmod +x** on this script to make it executable. On Windows, you can create a batch file like this:

```
@echo off
"c:\Program Files\Nuke7.0v9\Nuke7.0.exe" -t -i %*
```

Next, you will make the following changes in **Project Properties**, from the **Project** menu in Wing:

- Set **Python Executable** to **Command Line** and then enter the full path to this script
- Change **Python Options** under the **Debug** tab to **Custom** with a blank entry area (no options instead of **-u**)

Apply these changes and Wing will use NUKE's Python in its Python Shell (after restarting from its **Options** menu), for debugging, and for source analysis.

Configuring for Personal Learning Edition of NUKE

The above will not work in the Personal Learning Edition of NUKE because it does not support terminal mode. In that case, install a Python version that matches NUKE's Python and use that instead. You can determine the correct version to use by by looking at **sys.version** in NUKE's Script Editor.

Then set **Python Executable** in **Project Properties**, from the **Project** menu in Wing, to the full path to the Python interpreter. The correct value to use can be determined by running Python outside of Wing and executing the following:

```
import sys
print(sys.executable)
```

Using a matching Python version is a good idea to avoid confusion caused by differences in Python versions, but is not critical for Wing to function. However, Wing must be able to find *some* Python version or many of its features will be disabled.

Additional Project Configuration

When using Personal Learning Edition, and possibly in other cases, some additional configuration is needed to obtain auto-completion on the NUKE API also when the debugger is not connected or not paused.

The API is located inside the NUKE installation, in the **plugins** directory. The **plugins** directory (parent directory of the **nuke** package directory) should be added to the **Python Path** configured in Wing's **Project Properties** from the **Project** menu. On macOS this directory is within the NUKE application bundle, for example **/Applications/Nuke6.3v8/Nuke6.3v8.app/Contents/MacOS/plugins**.

Replacing the NUKE Script Editor with Wing Pro

Wing Pro can be used as a full-featured Python IDE to replace NUKE's Script Editor component. This is done by downloading and configuring NukeExternalControl.

First set up and test the client/server connection as described in the documentation for NukeExternalControl. Once this works, create a Python source file that contains the necessary client-side setup code and save this to disk.

Next, set a breakpoint in the code after the NUKE connection has been made, by clicking on the breakpoint margin on the left in Wing's editor or by clicking on the line and using **Add Breakpoint** in the **Debug** menu or the breakpoint icon in the toolbar.

Then debug the file in Wing Pro by pressing the green run icon in the toolbar or with **Start/Continue** in the **Debug** menu. After reaching the breakpoint, use the **Debug Console** in Wing to work interactively in that context.

You can also work on a source file in Wing's editor and evaluate selections within the file in the **Debug Console** with **Evaluate Selection in Debug Console** from the **Source** menu.

Both the **Debug Console** and Wing's editor should offer auto-completion on the NUKE API, at least while the debugger is active and paused in code that is being edited. The **Source Assistant** in Wing Pro provides additional information for symbols in the auto-completer, editor, and other tools in Wing.

This technique will not work in Wing Personal because it lacks the **Debug Console** feature. However, debugging is still possible using the alternate method described in the next section.

Debugging Python Running Under NUKE

Another way to work with Wing and NUKE is to connect Wing directly to the Python instance running under NUKE. In order to do this, you need to import a special module in your code, as follows:

```
import wingdbstub
```

You will need to copy **wingdbstub.py** out of the install directory listed in Wing's **About** box and may need to set **WINGHOME** inside **wingdbstub.py** to the location where Wing is installed if this value is not already set by the Wing installer. On macOS, **WINGHOME** should be set to the full path of Wing's **.app** folder.

Before debugging will work within NUKE, you must also set the **kEmbedded** flag inside **wingdbstub.py** to **1**.

Next click on the bug icon in the lower left of Wing's main window and make sure that **Accept Debug Connections** is checked.

Then execute the code that imports the debugger. For example, right click on one of NUKE's tool tabs and select **Script Editor**. Then in the bottom panel of the Script Editor enter **import wingstub** and press the **Run** button in NUKE's Script Editor tool area. You should see the bug icon in the lower left of Wing's window turn green, indicating that the debugger is connected.

If the import fails to find the module, you may need to add to the Python Path as follows:

```
import sys
sys.path.append("/path/to/wingdbstub")
import wingdbstub
```

After that, breakpoints set in Python modules should be reached and Wing's debugger can be used to inspect, step through code, and try out new code in the live runtime. Breakpoints set in the script

itself won't be hit, though, due to how Nuke loads the script, so code to be debugged should be put in modules that are imported.

For example, place the following code in a module named **testnuke.py** that is located in the same directory as **wingdbstub.py** or anywhere on the **sys.path** used by NUKE:

```
def wingtest():
   import nuke
   nuke.createNode('Blur')
```

Then set a breakpoint on the line **import nuke** by clicking in the breakpoint margin to the left, in Wing's editor.

Next enter the following and press the **Run** button in NUKE's Script Editor, just as you did when importing wingdbstub above:

```
import testnuke
testnuke.wingtest()
```

As soon as the second line is executed, Wing should reach the breakpoint. Then try looking around with the **Stack Data** and **Debug Console** (in Wing Pro only).

Debugger Configuration Detail

If the debugger import is placed into a script file, you may also want to call **Ensure** on the debugger, which will make sure that the debugger is active and connected:

```
import wingdbstub
wingdbstub.Ensure()
```

This way it will work even after the Stop icon has been pressed in Wing, or if Wing is restarted or the debugger connection is lost for any other reason.

For additional details on configuring the debugger see Debugging Externally Launched Code.

Limitations and Notes

When Wing's debugger is connected directly to NUKE and at a breakpoint or exception, NUKE's GUI will become unresponsive because NUKE scripts are run in a way that prevents the main GUI loop from continuing while the script is paused by the debugger. To regain access to the GUI, continue the paused script or disconnect from the debug process with the **Stop** icon in Wing's toolbar.

NUKE will also not update its UI to reflect changes made when stepping through a script or otherwise executing code line by line. For example, typing **import nuke; nuke.createNode('Blur')** in the **Debug Console** will cause creation of a node but NUKE's GUI will not update until the script is continued.

When the NUKE debug process is connected to the IDE but not paused, setting a breakpoint in Wing will display the breakpoint as a red line rather than a red dot during the time where it has not

113

yet been confirmed by the debugger. This can be any length of time, if NUKE is not executing any Python code. Once Python code is executed, the breakpoint should be confirmed and will be reached. This delay in confirming the breakpoint does not occur if the breakpoint is set while the debug process is already paused, or before the debug connection is made.

These problems should only occur when Wing's debugger is attached directly to NUKE, and can be avoided by working through **NukeExternalControl** instead, as described in the first part of this document.

Related Documents

For more information see:

- NUKE/NUKEX home page, which provides links to documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

5.4. Using Wing with Unreal Engine
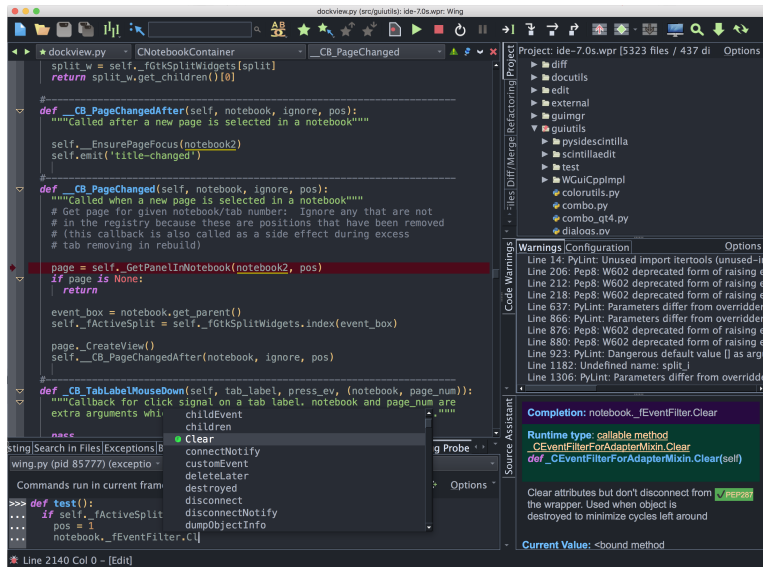
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Unreal Engine, a 3D world creation tool from Epic Games.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Unreal. To learn more about using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Creating a Project

To get started developing and debugging Python code written for Unreal Engine, you first need to create your Unreal project from Unreal Editor. This creates a directory that contains various starter files and directories and also your ***.uproject** file.

Once this is done, select **New Project** from Wing Pro's **Project** menu, choose **Local Host** and **Use Existing Directory** and then enter the full path of the directory that contains your ***.uproject** file.

Next select **Unreal Engine** under **Project Type** and press the **Next** button.

On the second **New Project** screen, select **Use Existing Python** and **Command Line**. Then enter the full path to the **python.exe** or **python** inside your Unreal Engine installation. For example:

```
c:\Program Files\Epic Games\UE_4.27\Engine\Binaries\ThirdParty\Python3\Win64\python.exe
```

> ### *Note*
>
> On macOS and possibly some other systems the Python command line included with Unreal Engine crashes on startup. In that case, you cannot use it for the **Command Line** setting in Wing. Instead, select **Use default** or point Wing to some other Python installation.
>
> This does mean that on systems where the Unreal Engine Python command line is broken, you will need to install Python separately if you don't already have it. Ideally this would match the version of Python that Unreal Engine uses (as of early 2022 this was Python 3.7) so that auto-completion on builtins and the standard library are accurate.

Now you can complete creation of your Wing project by pressing the **Create Project** button.

Wing will show the result of this operation, including all steps taken. It may warn about needing to restart Unreal Editor and/or turning on Developer Mode in Unreal Editor, which is done from **Preferences > Plugins > Python**. Developer Mode is needed to generate the stubs file that Wing uses to provide auto-completion for Unreal Engine's API. Even if that is already enabled, you may need to restart Unreal Editor before debugging works, since it does not rescan the disk for new Python Path directories created while it is running.

That's all there is to it!

Working with Wing

You can now type **import unreal** and should receive auto-completion and other support after typing **unreal.** in the editor.

You can also debug Python code in Unreal Editor. To start the debugger, run the following Python code in Unreal Editor:

```
import wingdbstub
```

You should see Wing's toolbar change to indicate that a debug process is connected.

Now you can open your Python script(s) in Wing and place breakpoints by clicking on the leftmost editor margin. Wing will stop on breakpoints and also any unhandled exceptions that are reach in your code.

If you drop the debug connection and want to reestablish it without restarting Unreal Editor, you can do this by executing the following code:

```
wingdbstub.Ensure()
```

You may need to **import wingdbstub** again before doing that, if you are executing this code from a different context than your original import of that module.

Debugging Notes

(1) While stopped at a breakpoint in Wing, Unreal Editor will be unresponsive, because the debugger has taken over control. You will need to continue in the debugger or disconnect Wing's debugger by pressing the red Stop item or using the **Debug > Stop Debugging** menu item before you can return to using Unreal Editor's UI.

(2) Because Unreal Engine does not normally call into any Python code on a regular basis, Wing's debugger cannot always immediately process requests from the IDE, such as those that add or remove breakpoints or request the debug process to Pause.

As a result, breakpoints set may initially appear as a small dash and only convert into a round circle once the debugger confirms setting the breakpoint. This may not be until you next execute a Python script, but should usually occur before the breakpoint is actually reached. However, sometimes old breakpoints removed in the IDE will not yet be removed inside Unreal Engine, causing Wing to stop there before the breakpoint updates are completed.

This should only occur during times when no Python code is being executed by Unreal Engine. For example, if you have a timer written in Python or other Python script activity occuring on a regular basis, then you should not see this issue.

(3) In rare cases, it may to be possible to get into a state where breakpoints are no longer reached at all. In this case, restarting Unreal Engine solves the problem.

Using Live Runtime Analysis

A way to get even better auto-completion in Wing is to run to a breakpoint in the debugger and then work in the editor or the **Debug Console** accessed from the Wing's **Tools** menu. If you're working in the current Python stack frame, Wing inspects the live runtime to populate the auto-completer and other code intelligence tools. This also has the advantage that you can immediately try out the code that you are writing in the **Debug Console**.

For more information on Wing's capabilities, see the **Tutorial** in the **Help** menu or the Quick Start Guide.

How it Works

Wing's auto-completion for Unreal Editor depends on its creation of the **unreal.py** stubs file, which is located in **Intermediate/PythonStub** inside your Unreal Editor project directory (where the ***.uproject** is located). This is created for any project opened by Unreal Editor after you've turned on **Developer Mode** in Unreal Editor from **Preferences > Plugins > Python**.

If you follow the above instructions for creating a Wing project for Unreal Editor, Wing will automatically add the **Intermediate/PythonStub** directory to the **Python Path** that is configured in **Project Properties** under the **Environment** tab. You can also add this manually to an existing Wing project.

Notes on sys.path

Because of how Unreal Engine packages Python, Wing cannot obtain the Python path from the command line executable, as it usually does.

This means that you need to manually add any other directories from which you import code to your **Python Path** in Wing's **Project Properties**. Otherwise, Wing will not be able to find and offer auto-completion and other code intelligence for code in those directories.

One way to review all the directories that are on the Python path inside Unreal Editor is to run the following code there:

```
import sys, os
print(os.pathsep.join(sys.path))
```

The output of this can be pasted into the **Python Path** field in Wing's **Project Properties** after selecting **View as Text**.

Note that many of the entries Unreal places on **sys.path** do not exist, and thus are not actually needed. However, you can still add them to the **Python Path** in Wing's **Project Properties**. Wing will warn about them but allows them in the configuration and will update its knowledge of your code if those directories are later created and populated.

Debug Configuration Details

When you created your Wing project for Unreal Engine, Wing wrote a pre-configured copy of its **wingdbstub.py** module into **Content/Python** inside your Unreal Engine project directory (where the ***.uproject** is located). The values that Wing sets in this module are **WINGHOME**, to tell the debugger where to find the debugger implementation, and **kEmbedded**, to tell Wing that it is debugging code running in an embedded instance of Python. The latter alters how it treats an exit from the outermost stack frame; without it, the debug connection would drop and need to be reestablished with **wingdbstub.Ensure()** after each invocation of Python code.

Wing also turns on *Accept Debug Connections*` in the menu for the bug icon in the lower left of Wing's window, so that debugging initiated from outside of Wing can connect to the IDE.

Related Documents

For more information see:

- Unreal Engine website
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

5.5. Using Wing with Source Filmmaker
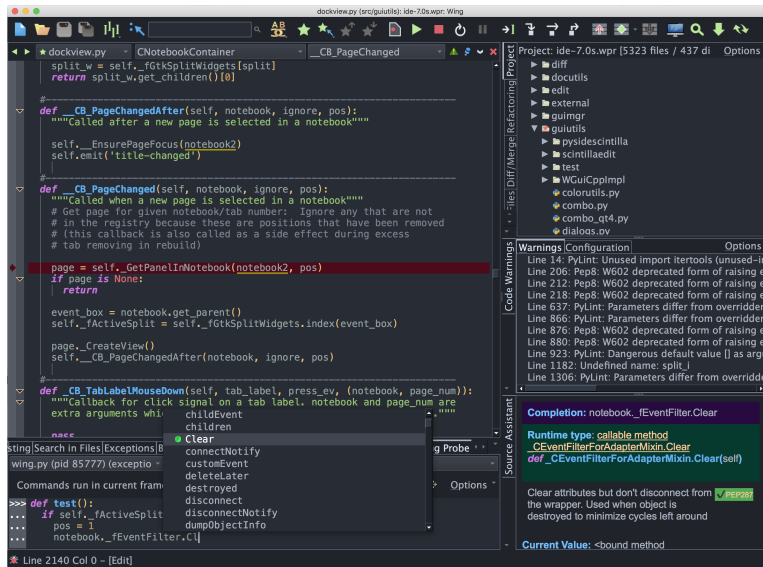
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Source Filmmaker (SFM), a movie-making tool built by Valve using the Source game engine.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Source Filmmaker. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Debugging Setup

Wing can debug Python code that's saved in a file, but not code entered in the Script Editor window. As of version 0.9.8.5 (released May 2014), this includes scripts run from the main menu. In all versions, code in imported modules may be debugged.

When debugging Python code running under **SFM**, the debug process is initiated from outside of Wing, and must connect to the IDE. This is done with **wingdbstub**, as described in in the Debugging Externally Launched Code section of the manual. Because of how **SFM** sets up the interpreter, you must set **kEmbedded=1** in your copy of **wingdbstub.py**.

Some versions of **SFM** comes with **wingdbstub.py** in the site-packages directory in its Python installation. However, this file must match the version of Wing you are using so you may need to copy **wingdbstub.py** from your Wing install directory to the site-packages directory. The default location of the site-packages directory is:

```
<STEAM>\steamapps\common\SourceFilmmaker\game\sdktools\python\2.7\win32\Lib\site-packages
```

Before debugging, click on the bug icon in lower left of Wing's window and make sure that **Accept Debug Connections** is checked. After that, you should be able to reach breakpoints by causing the scripts to be invoked from **SFM**.

To start debugging and ensure there's a connection from the **SFM** script being debugged to Wing, execute the following before any other code executes:

```
import wingdbstub
wingdbstub.Ensure()
```

To use the **python** executable found in the **SFM** application directory to run Wing's **Python Shell** tool and to debug standalone Python scripts, enter the full path of the **python.exe** file under **Command Line** in the **Python Executable** field of the **Project Properties```dialog**.

Related Documents

For more information see:

- Source Filmmaker website
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

5.6. Using Wing with pygame
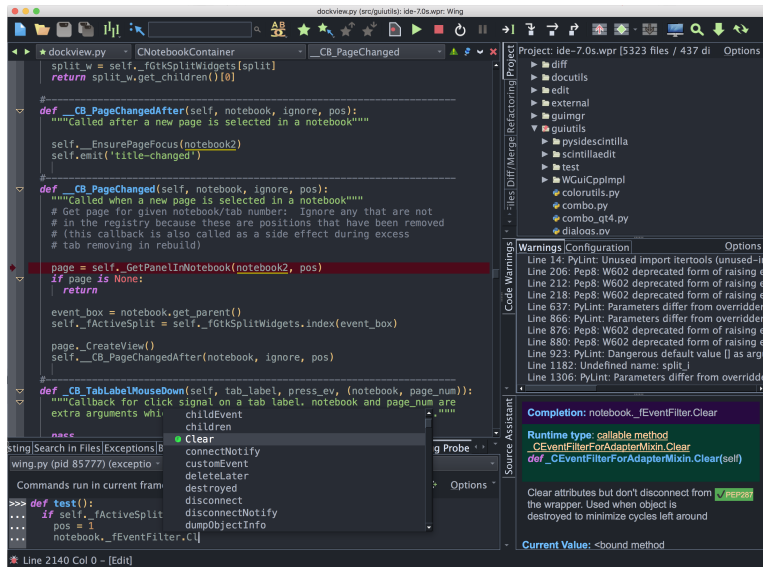
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for pygame, an open source framework for game development with Python.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for pygame. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Project Configuration

Pygame works just fine with Wing without any special configuration. You'll need to first install pygame according to the instructions on the pygame website.

To create a new project, use **New Project** in Wing's **Project** menu with **Project Type** set to **Pygame**. You'll be able to select or create a source directory for your project and select or create a Python environment. See Creating a Project for details on creating projects in Wing.

After you press **Create Project** in the **New Project** dialog, find your main entry point, open it into Wing, and select **Set Current as Main Entry Point** in the **Debug** menu.

Debugging

Now you can launch your game from Wing with **Start/Continue** in the **Debug** menu. Wing will stop on any exceptions or breakpoints reached while running your game, and you can use the debugger to step through code, inspect the value of variables, and try out new code interactively.
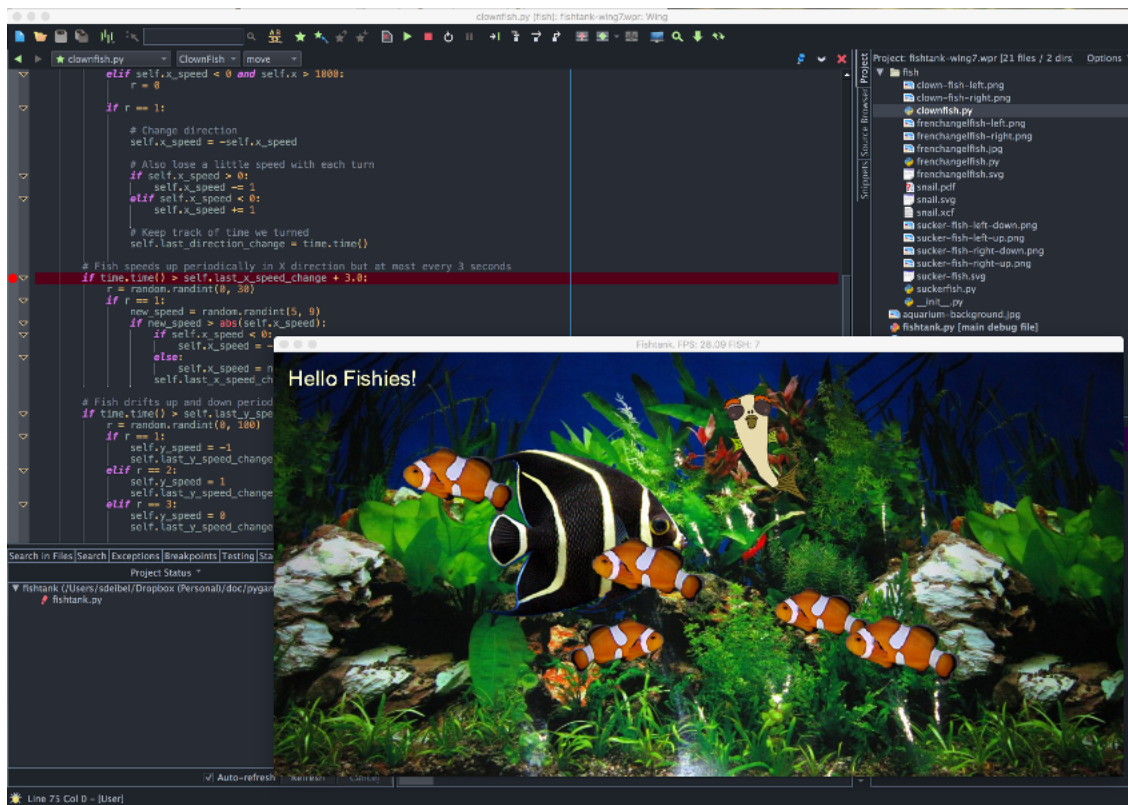
To learn more about Wing's features, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Related Documents

Wing provides many other options and tools. For more information:

- pygame home page provides downloads and documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

Unmaintained How-Tos

This section contains unmaintained How-Tos for using Wing with older and less commonly used frameworks, tools, and alternate Python implementations.

6.1. Using Wing with Twisted
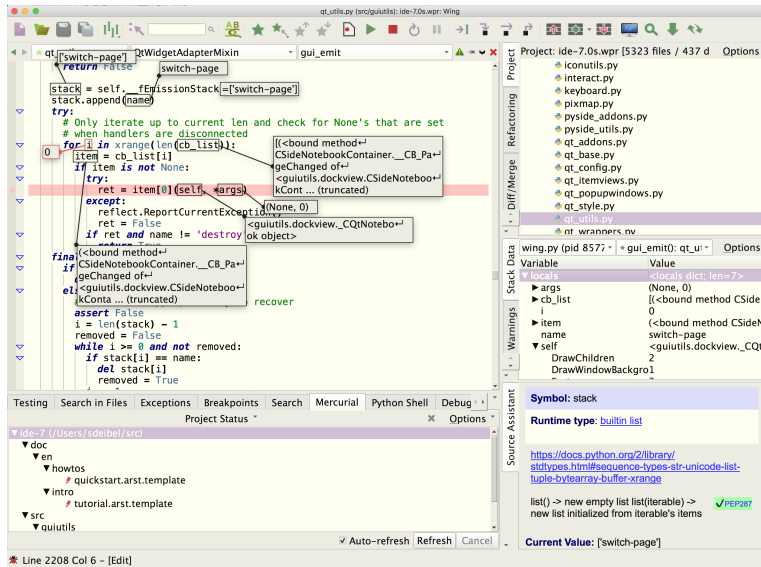
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Twisted.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Twisted. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Note: This document is not being maintained and was last tested with Twisted version 8.

Project Configuration

To create a new project, use **New Project** in Wing's **Project menu. Select the project type ``Twisted** and under **Python Executable** select **Custom** and then enter the full path of the Python you plan to use with Twisted. You can determine the correct value to use by executing the following commands interactively in Python. If you are using virtualenv, this will be the virtualenv's Python executable:

```
import sys
sys.executable
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

Remote Development

Wing Pro can work with Twisted code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See Remote Hosts for more information on remote development with Wing Pro.

Debug Configuration

To debug Twisted code launched from within Wing, create a file with the following contents and set it as your main entry point by adding it to your project and then using the **Set Main Entry Point** item in the **Debug** menu:

```
from twisted.scripts.twistd import run
import os
try:
    os.unlink('twistd.pid')
except OSError:
    pass
run()
```

Then go into the **File Properties** for this file (by right clicking on it) and set **Run Arguments** as follows:

```
-n -y filename.tac
```

The **-n** option tells Twisted not to daemonize, which would cause the debugger to fail because sub-processes are not automatically debugged. The **-y** option serves to point Twisted at your **.tac** file. Replace **filename.tac** in the above example with the correct name of your file.

Wing Pro may be able to debug Twisted without the **-n** option, if you enable **Debug Child Processes** under the **Debug/Execute** tab of **Project Properties**, from the **Project** menu.

You can also launch Twisted code from outside of Wing as described in Debugging Externally Launched Code in the manual.

Related Documents

For more information see:

- Twisted home page, which provides links to documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

6.2. Using Wing with Plone

> **Note**
>
> **"The best solution for debugging Zope and Plone"** *-- Joel Burton, Member, Plone Team*
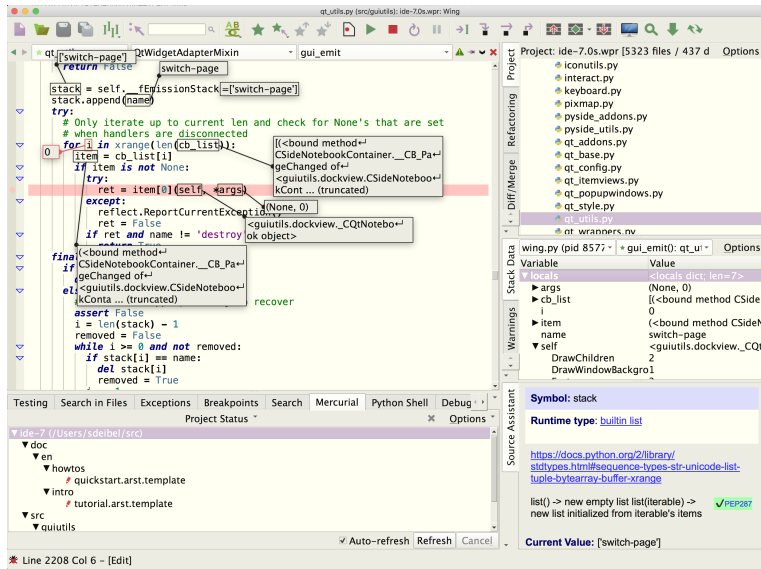
[Wing Pro](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [Plone](#) content management system.

If you do not already have Wing Pro installed, [download it now](#).

This document describes how to configure Wing for Plone. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

Note: This document is not being maintained and was last tested with Plone 4.

Introduction

These instructions are for the Plone 4+ unified installer. Wing no longer supports old style Zope Product name space merging so it cannot be used with older versions of Plone.

Configuring your Project

To create a new project, use **New Project** in Wing's **Project** menu. Select the project type **Plone** and under **Python Executable** select **Custom** and then enter the full path of the Python you plan to use with Plone. The full path can be found by looking at the top of many of the scripts in **zinstance/bin** or **zeocluster/bin**. You can also determine the correct value to use by executing the following commands interactively in Python. If you are using virtualenv, this will be the virtualenv's Python executable:

```
import sys
sys.executable
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

Next find and open the file **zinstance/bin/instance** and select **Set Current as Main Entry Point** in **Project** menu. If you have a ZEO cluster, instead use **zeocluster/bin/client1** or whatever name is given in the **.cfg** file. Wing reads the **sys.path** updates from that file so that it can find your Plone modules.

Debugging

If you have followed the instructions above, you should be able to start debug from the toolbar or **Debug** menu. The debugger will stop on breakpoints and any exceptions that are printed, and debug data can be viewed in the **Stack Data** tool, by hovering over values, and in Wing Pro by pressing **Shift-Space** or with the interactive **Debug Console**.

Related Documents

Wing provides many other options and tools. For more information:

- Plone home page, which provides links to documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

6.3. Using Wing with Turbogears
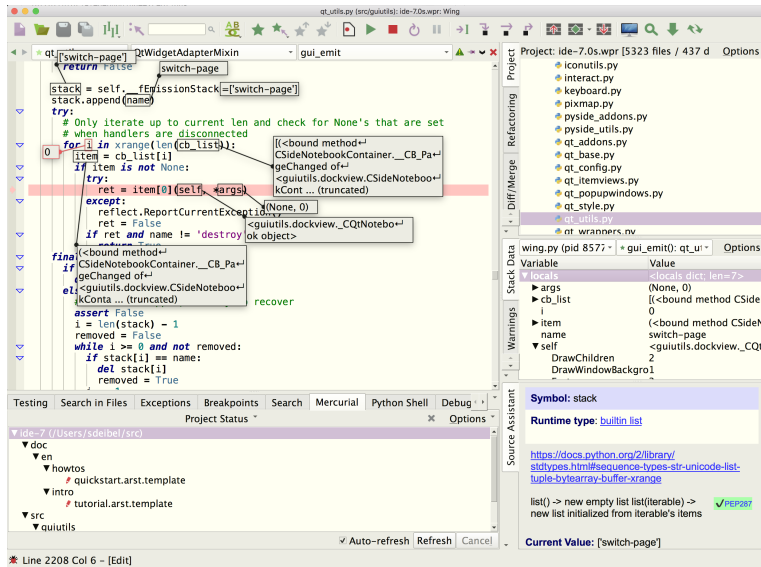
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for the Turbogears, web development framework.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for Turbogears. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Note: This document is not being maintained and was last tested with Turbogears 2.

Project Configuration

This section assumes your Turbogears 2 project is called **wingtest**. If not, substitute your project name in the following instructions.

- Go into the Turbogears instance directory **wingtest** and run Wing
- Add your instance directory to the project and save it as **wingtest.wpr** There is no need to add all of Turbogears to the project; just the instance should suffice.
- Add also the **paster** to your project. Then open it and and set it as main entry point from the **Debug** menu
- Open up the Python Shell tool and type **import sys** followed by **sys.executable** to verify whether Wing is using the Python that will be running Turbogears. If not, open **Project Properties** and set the **Python Executable** to the correct one.
- Next right click on **paster** and select **File Properties**. Under the **Debug** tab, set **Run Arguments** to **serve development.ini** (do not include the often-used --reload argument, as this will interfere with debugging). Then also set **Initial Directory** to the full path of **wingtest**.

128

Debugging

To debug Turbogears 2 from Wing:

- Set a breakpoint on the **return** line of **RootController.index()** in your **root.py** or somewhere else you know will be reached on a page load
- Start debugging in Wing from the toolbar or or **Debug** menu. If Wing shows a warning about **sys.settrace** being called in **DecoratorTools** select **Ignore this Exception Location** in the **Exceptions** tool in Wing and restart debugging. In general, **sys.settrace** will break *any* Python debugger but Wing and the code in DecoratorTools both take steps to keep debugging working in this case.
- Bring up the **Debug I/O** tool in Wing and wait until the server output shows that it has started
- Load **http://localhost:8080/** or the page you want to debug in a browser

Wing should stop on your breakpoint. From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

Remote Development

Wing Pro can work with Pyramid code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See Remote Hosts for more information on remote development with Wing Pro.

Related Documents

For more information see:

- Turbogears home page for downloads and documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

6.4. Using Wing with Google App Engine SDK for Python
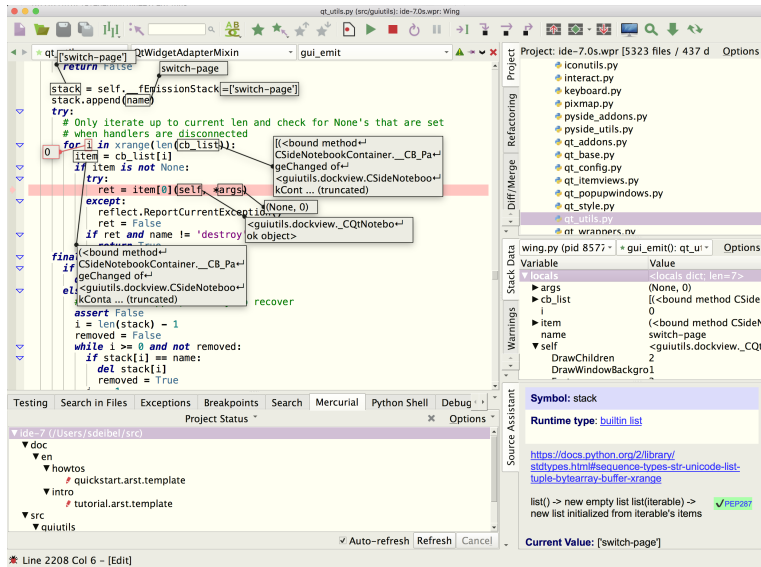
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Google App Engine SDK for Python. Wing Pro provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing Pro for Google App Engine. To get started using Wing Pro as your Python IDE, please refer to the tutorial in Wing Pro's **Help** menu or read the Quickstart Guide.

Note: This document is not being maintained and was last tested with Google App Engine version 1.9.

Creating a Project

Before trying to configure a project in Wing Pro, first install and set up Google App Engine SDK for Python and verify that it is working by starting it outside of Wing and testing it with a web browser.

Next, create a project in Wing Pro with **New Project** in the **Project** menu and selecting **Google App Engine** as the project type. Then use **Add Directory** in the **Project** menu to add your source directories to the project. You should also add at least **dev_appserver.py**, which is located in the top level of the Google SDK directory.

Next open up **dev_appserver.py** in Wing's editor and select **Set Current as Main Entry Point** in the **Debug** menu. This tells Wing to use this file as the main entry point, which is then highlighted in the **Project** tool.

Next you need to go into **Project Properties** and set **Debug/Execute > Debug Child Processes** to **Always Debug Child Processes**. This is needed because App Engine creates more than one process.

Finally, save your project with **Save Project** in the **Project** menu. Store the project at or near the top level of your source tree.

Configuring the Debugger

Before trying to debug make sure you stop Google App Engine if it is already running outside of Wing.

You can debug code running under Google App Engine SDK for Python by selecting **Start / Continue** from the **Debug** menu or using the green run icon in the toolbar. This displays a dialog that contains a **Run Arguments** field that must be altered to specify the application to run. For example, to run the guestbook demo that comes with the SDK, the run arguments would be **"${GOOGLE_APPENGINE_DIR}/demos/guestbook"** where **${GOOGLE_APPENGINE_DIR}** is replaced by the full pathname of the directory the SDK is installed in. The quotation marks are needed if the pathname contains a space. In other apps, this is the path to where the **app.yaml** file is located. If this path name is incorrect, you will get an error when you start debugging.

You can also leave the environment reference **${GOOGLE_APPENGINE_DIR}** in the path and define an environment variable under the **Environment** tab of the **Debug** dialog. Or use **${WING:PROJECT_DIR}** instead to base the path on the directory where Wing's project file was saved.

For most projects, you'll need to add at least **--max_module_instances=1** to the run arguments, and you may also want to add **--threadsafe_override=false**. These command line arguments disable some of GAE's threading and concurrency features that can prevent debugging from working properly.

Add a **--port=8082** style argument if you wish to change the port number that Google App Engine is using when run from Wing's debugger. Otherwise the default of **8080** will be used.

Using a partial path for the application may also be possible if the **Initial Directory** is also set in under the **Debug** tab.

Next, click the **OK** button to save your settings and start debugging. Once the debugger is started, the **Debug I/O** tool (accessed from the **Tools** menu) should display output from App Engine, and this should include a message indicating the hostname and port at which App Engine is taking requests.

If Google App Engine asks to check for updates at startup, it will do so in the **Debug I/O** tool and you can press "y" or "n" and then **Enter** as you would on the command line. Or send the **--skip_sdk_update_check** argument on the command line to **dev_appserver.py** to disable this.

Using the Debugger

After you have configured the debugger, set a break point in any Python code that is executed by a request and load the page in the browser. For example, to break when the main page of the guestbook demo is generated, set a breakpoint in the method **Mainpage.get** in **guestbook.py**. When you reach the breakpoint, the browser will stop and wait while Wing debugs the code.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. The **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu. You can also see data values by hovering the mouse over symbols in the editor or **Debug Console** and you can press **F4** to go to the point of definition of any symbol.

Continue running with the green run button in the toolbar. Unless another breakpoint or exception is reached, this shoud complete the page load in the browser.

You may edit the Python code for an application while the App Engine is running, and then reload in your browser to see the result of any changes made. In most cases, there is no need to restart the debug process after edits are made. However, if you try the browser reload too quickly, while App Engine is still restarting, then it may not respond or breakpoints may be missed.

To learn more about the debugger, try the **Tutorial** in Wing Pro's **Help** menu.

Improving Auto-Completion and Goto-Definition

Wing can't parse the **sys.path** hackery used by Google App Engine SDK for Python so it may fail to find some modules for auto-completion, goto-definition and other features. To work around this, set a breakpoint in **_run_file** in **dev_appserver.py** and start debugging. Then, after **script_name** has been set, in the **Debug Console** tool type the following:

```
os.pathsep.join(_PATHS.script_paths(script_name))
```

Copy this to the clipboard and open up the file properties for **dev_appserver.py** by right-clicking on the file. Then, in **Project Properties** under the **Environment** tab select **Custom** for the **Python Path**, click on the **View as Text** button and paste in the extra path.

You will need to redo this if you move the app engine installation, or you can use **${WING:PROJECT_DIR}** in the paths to based them on the location of the project file.

Debugging Multiple Applications

To set up multiple entry points without needing to change the file properties for **dev_appserver.py**, use **Named Entry Points** in the **Debug** menu. Each Named Entry Point can contain a different commands line and environment for **dev_appserver.py**.

In this case, configuration of **Python Path** and other values mentioned above is done in the Launch Configuration used in each Named Entry Point.

Notes

App Engine runs code in a restricted environment that prevents access to some system information, including process ID. This causes some of the sub-processes created by App Engine to be shown with process id -1. In this case they are not listed as children of the parent process and you will need to kill both processes, one at a time, from the toolbar or **Debug** menu.

Windows users may need to set the **TZ** environment variable to **UTC** via the environment field in Project Properties to work around problems with setting **os.environ['TZ']** while a process is running

(this is a Windows runtime bug). One possible symptom of this is repeated 302 redirects that prevent logging in or other use of the site.

Related Documents

For more information see:

- Google App Engine SDK for Python for downloads and documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

6.5. Using Wing with mod_python
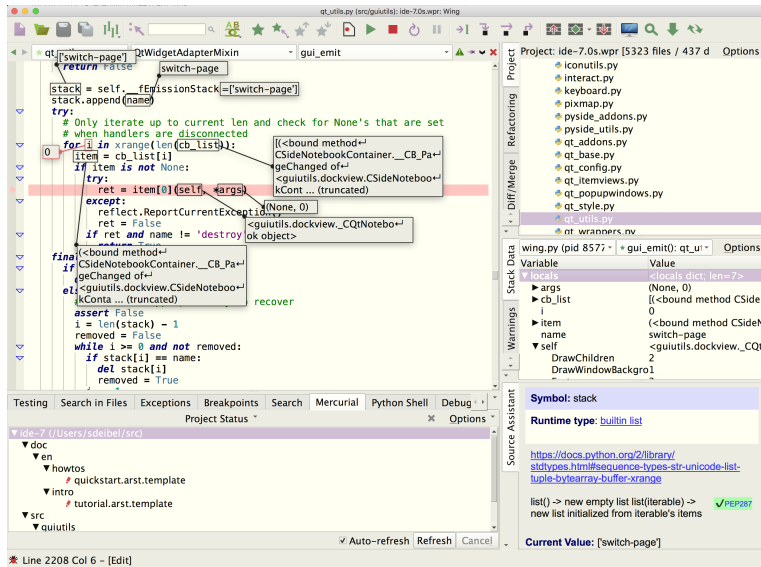
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code that is run by the mod_python module for the Apache web server.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for mod_python. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Note: This document is not being maintained and was last tested in ancient times.

Introduction

This document assumes mod_python is installed and Apache is configured to use it; please see the installation chapter of the mod_python manual for information on how to install it.

Since Wing's debugger takes control of all threads in a process, only one http request can be debugged at a time. In the technique described below, a new debugging session is created for each request and the session is ended when the request processing ends. If a second request is made while one is being debugged, it will block until the first request completes. This is true of requests processed by a single Python module and it is true of requests processed by multiple Python modules in the same Apache process and its child processes. As a result, it is recommended that only one person debug mod_python based modules per Apache instance and production servers should not be debugged.

Quick Start

- Copy **wingdbstub.py** (from the install directory listed in Wing's **About** box) into either the directory the module is in or another directory in the Python path used by the module.

- Edit **wingdbstub.py** if needed so the settings match the settings in your preferences. Typically, nothing needs to be set unless Wing's debug preferences have been modified. If you do want to alter these settings, see the Manually Configured Remote Debugging section of the Wing reference manual for more information.
- Copy **wingdebugpw** from your Settings Directory into the directory that contains the module you plan to debug. This step can be skipped if the module to be debugged is going to run on the same machine and under the same user as Wing. The **wingdebugpw** file must contain exactly one line.
- Insert **import wingdbstub** at the top of the module imported by the mod_python core.
- Insert **if wingdbstub.debugger != None: wingdbstub.debugger.StartDebug()** at the top of each function that is called by the mod_python core.
- Allow debug connections to Wing by setting the **Debugger > Listening > Accept Debug Connections** preference to true.
- Restart Apache and load a URL to trigger the module's execution.

Example

To debug the **hello.py** example from the Publisher chapter of the mod_python tutorial, modify the **hello.py** file so it contains the following code:

```
import wingdbstub

def say(req, what="NOTHING"):
  wingdbstub.Ensure()
  return "I am saying %s" % what
```

And set up the mod_python configuration directives for the directory that **hello.py** is in as follows:

```
AddHandler python-program .py
PythonHandler mod_python.publisher
```

Then set a breakpoint on the **return "I am saying %s" % what** line, make sure Wing is listening for a debug connection, and load **http://[server]/[path]/hello.py** in a web browser (substitute appropriate values for [server] and [path]). Wing should then stop at the breakpoint.

Remote Development

Wing Pro can work with mod_python code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See Remote Hosts for more information on remote development with Wing Pro.
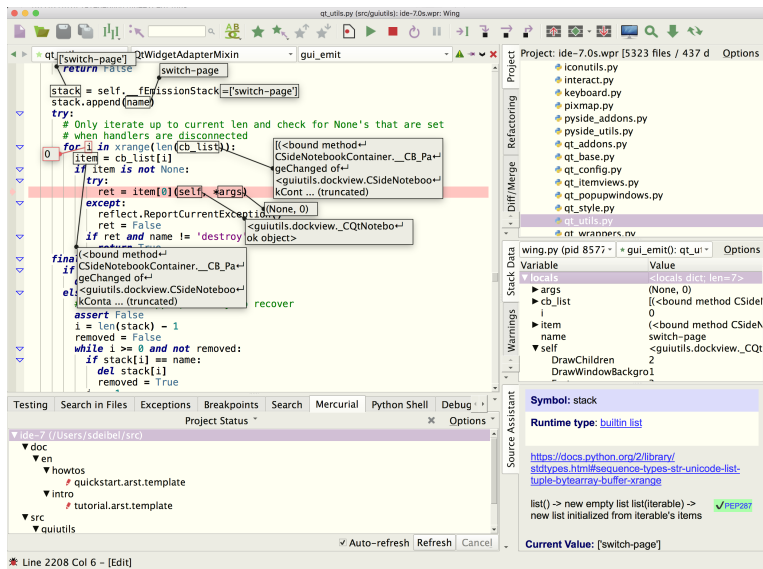
Related Documents

For more information see:

- Mod_python website for downloads and documentation.

- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

6.6. Debugging Code Running Under Py2exe

[Wing Pro](#) is a Python IDE that can be used to debug Python code running in an application packaged by [py2exe](#). This is useful to solve a problem seen only when the code is running from the package, or so that users of the packaged application can debug Python scripts that they write for the app.

If you do not already have Wing Pro installed, [download it now](#).

This document just describes how to configure Wing for debugging Python code running under **py2exe**. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

Note: This document is not maintained and was last tested in 2007.

Configuring the Debugger

To debug code running under **py2exe** you will need to use **wingdbstub** to initiate debug from outside of Wing, as described in [Debugging Externally Launched Code](#), along with some additional configuration described below.

There are two important ways in which the environment differs when code runs under **py2exe**:

1. When **py2exe** produces the ***.exe**, it strips out all but the modules it thinks will be needed by the application. This will remove modules needed by Wing's debugger.
2. **py2exe** runs in a slightly modified environment and it ignores the **PYTHONPATH** environment.

As a result, some custom code is needed so the debugger can find and load the modules that it needs:

```
# Add extra environment needed by Wing's debugger
import sys
```

```
import os
extra = os.environ.get('EXTRA_PYTHONPATH')
if extra:
    sys.path.extend(extra.split(os.pathsep))
print(sys.path)

# Start debugging
import wingdbstub

# Just some test code
print("Hello from py2exe")
print("frozen", repr(getattr(sys, "frozen", None)))
print("sys.path", sys.path)
print("sys.executable", sys.executable)
print("sys.prefix", sys.prefix)
print("sys.argv", sys.argv)
```

You will need to set the following environment variables before launching the packaged application:

```
EXTRA_PYTHONPATH=\Python25\Lib\site-packages\py2exe\samples\simple\dist;\Python25\lib;\Python25\dlls
WINGDB_EXITONFAILURE=1
```

In this example, **\Python25\Lib\site-packages\py2exe\samples\simple\dist** contains the source for the packaged application and also the copy of **wingdbstub.py** used to initiate debug.

The other added path entries point at a Python installation that matches the one being used by **py2exe**. This is how the debugger will load missing standard library modules from outside of the **py2exe** package.

Setting **WINGDB_EXITONFAILURE** causes the debugger to print an exception and exit if it fails to load. Without this it will fail silently and continue to run without debug.

The above was tested with Python 2.5 using **py2exe** run with **-q** and **-b2** options.

Related Documents

For more information see:

- py2exe home page provides downloads and documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

6.7. Using Wing with IDA Python
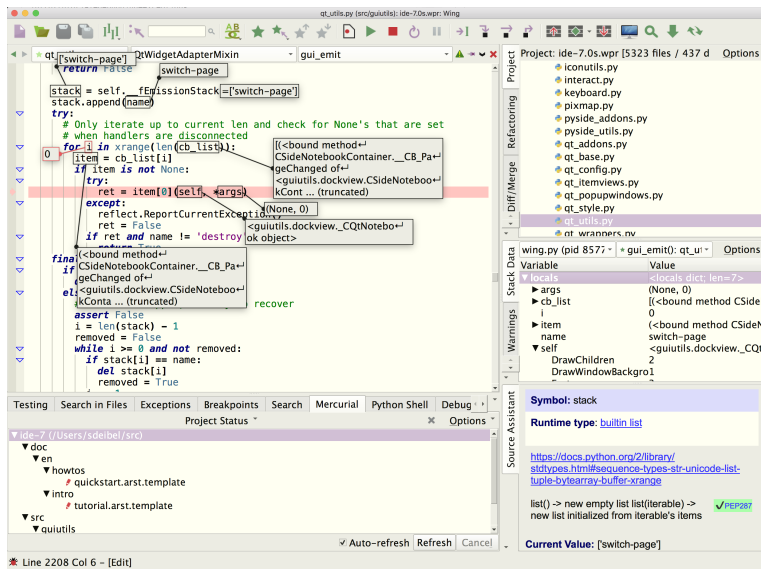
Wing Pro is a Python IDE that can be used to develop, test, and debug Python code written for Hex-Rays IDA multi-processor disassembler and debugger.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for IDA. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Note: This document is not maintained and was last tested in 2012.

Debugging IDA Python in Wing

IDA embeds a Python interpreter that can be used to write scripts for the system. In order to debug Python code that is run within IDA, you need to import a special module in your code, as follows:

```
import wingdbstub
wingdbstub.Ensure()
```

You will need to copy **wingdbstub.py** out of your Wing installation and may need to set **WINGHOME** inside **wingdbstub.py** to the location where Wing is installed. On macOS, this is the full path of Wing's **.app** folder.

Even though this is an embedded instance of Python, leave the **kEmbedded** flag set to **0**.

Next click on the bug icon in the lower left of Wing's main window and make sure that **Accept Debug Connections** is checked. Then restart IDA and the debug connection will be made as soon as the above code is executed.

At that point, any breakpoints set in Python code will be reached and Wing can be used to inspect the runtime state, step through code, and try out new code interactively.

For more information on this configuration, see Debugging Externally Launched Code.

Related Documents

For more information see:

- Hex-Rays IDA home page provides links to documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.

6.8. Using Wing with IronPython
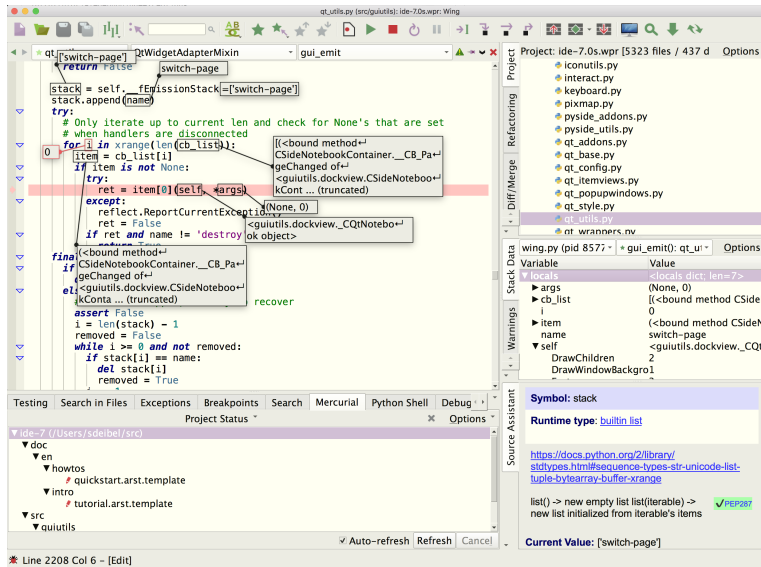
Wing Pro is a Python IDE that can be used to develop and test Python code written for IronPython.

If you do not already have Wing Pro installed, download it now.

This document describes how to configure Wing for IronPython. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the Quickstart Guide.

Note: This document is not maintained and was last reviewed in 2011.

Project Configuration

For instructions on setting up Wing with IronPython, see IronPython and Wing: Using Wing Python IDE with IronPython. This article provides a script to help with setting up auto-completion for the .NET framework, and some information on how to get Wing to execute your code in IronPython. It was written by Michael Foord, co-author of the book IronPython in Action.

The script the article refers to is now shipped with Wing, in **src\wingutils\generate_pi.py** inside the Wing install directory, which is listed in Wing's **About** box.

Related Documents

For more information see:

- IronPython home page provides downloads and documentation.
- Quickstart Guide contains additional basic information about getting started with Wing.
- Tutorial provides a gentler introduction to Wing's features.
- Wing Reference Manual documents Wing in detail.