# AutoTest Studio Getting started tutorials 1: Get started with AutoTest Studio

**Introduction to AutoTest Studio**

AutoTest Studio is an excellent software automation testing platform, using python language as the use case writing language, integrating use case development, debugging, management, task scheduling, automatic generation of test logs and test reports and other functions, meeting the automation implementation needs of most projects , And it is free. Compared with other similar automated testing frameworks, such as Robot Framework and TestNg, it is more flexible and simple, and does not require manual installation of numerous dependent packages and complex environment setup.

Official website：<https://www.autoteststudio.com/>

Download URL：<https://www.autoteststudio.com/?page_id=1259>

The latest version is 2.0.3.

Main features：

**Logs and Reports**

The test log records the details of the test case run, and the test report can count the results of the test case execution in each test task and present it as a chart.

[**Data Dictionary**](https://www.autoteststudio.com/?page_id=1258&c=t_008)

AutoTest Studio maintains an independent data configuration file for each test project. Through the data dictionary management tool, you can configure up to millions of data records, and access these datas through API interfaces in test cases.

**Package Manager**

The Python package manager allows you to install,delete,upgrade,import,and export the Python packages through a graphical interface.

**Code Editor**

AutoTest Studio has a professional code editor, which can perform operations, such as keyword highlighting, code folding, expansion, code auto-completion, and code formatting.

**Debugger**

AutoTest Studio integrates a powerful Python Debugger,which makes it easier for you to find exceptions and error messages in program execution.

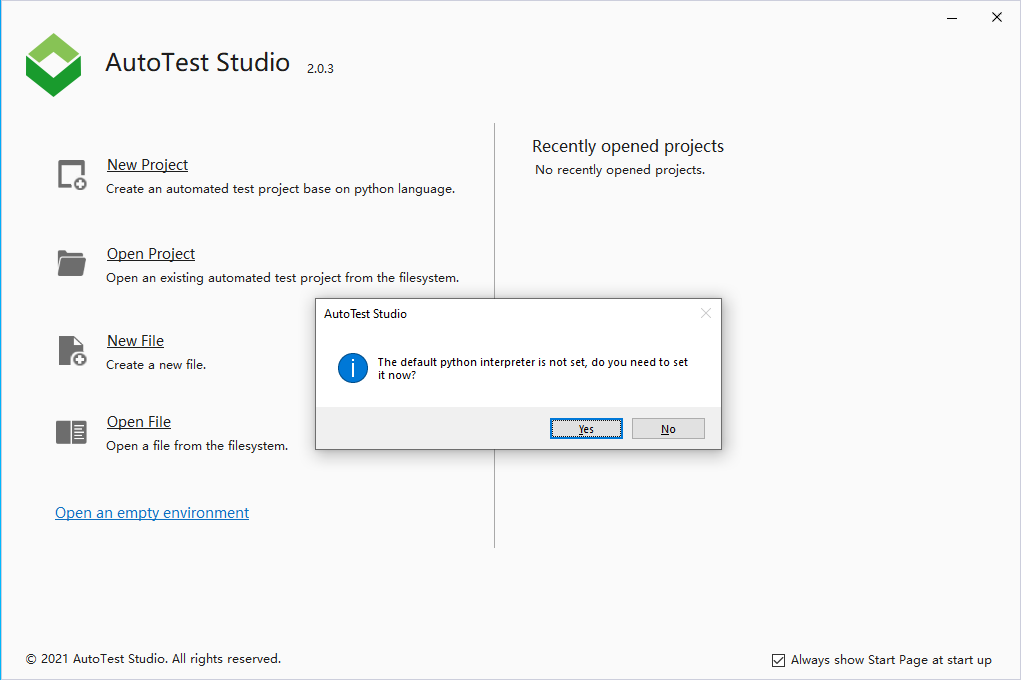
**Test case Runner**

AutoTest Studio has an advanced task engine that can provide a variety of task execution scheduling methods, such as timed tasks, cyclic execution, and execution of test cases in specified states.

**AutoTest Studio installation and configuration**

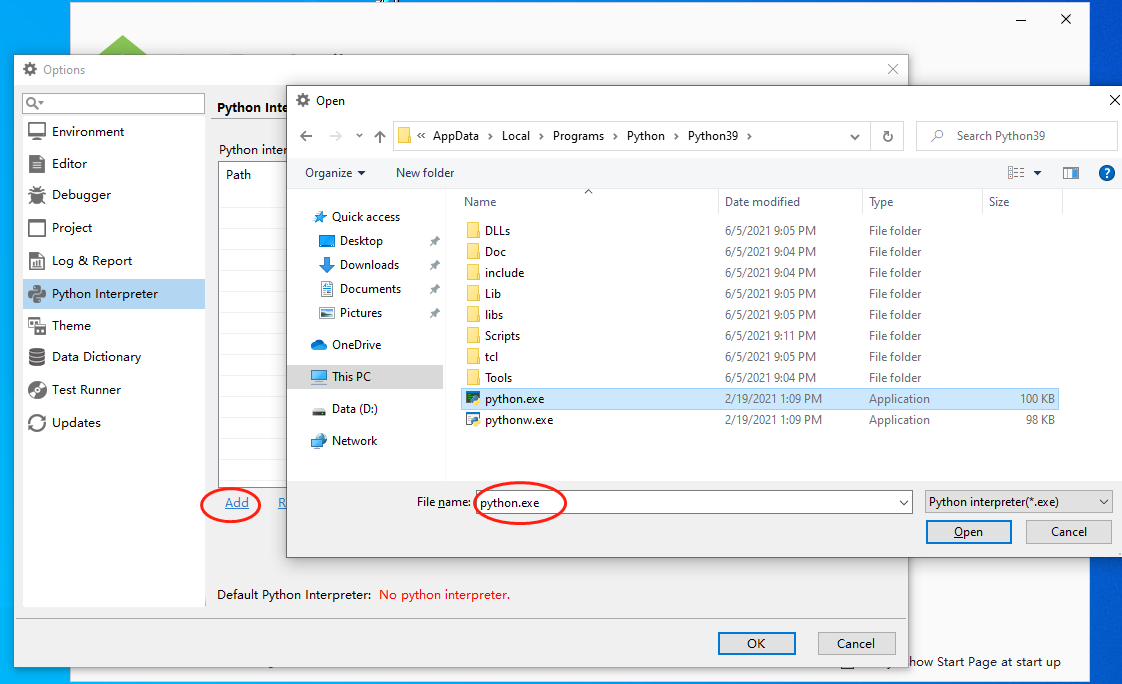
After downloading the installation package from the official, click "Next" to easily complete the installation.

After completing the installation, when you run the program for the first time, you will be prompted to configure the python interpreter path. I need to remind that this python interpreter is required for some plug-ins in the background of the program, not the python environment where the project runs. In the second section, I will introduce how to configure The python environment where the project is running.

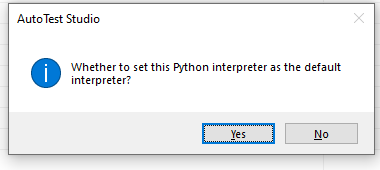


Of course, you can skip it directly and configure it later. Here we follow the prompts to configure the python interpreter.

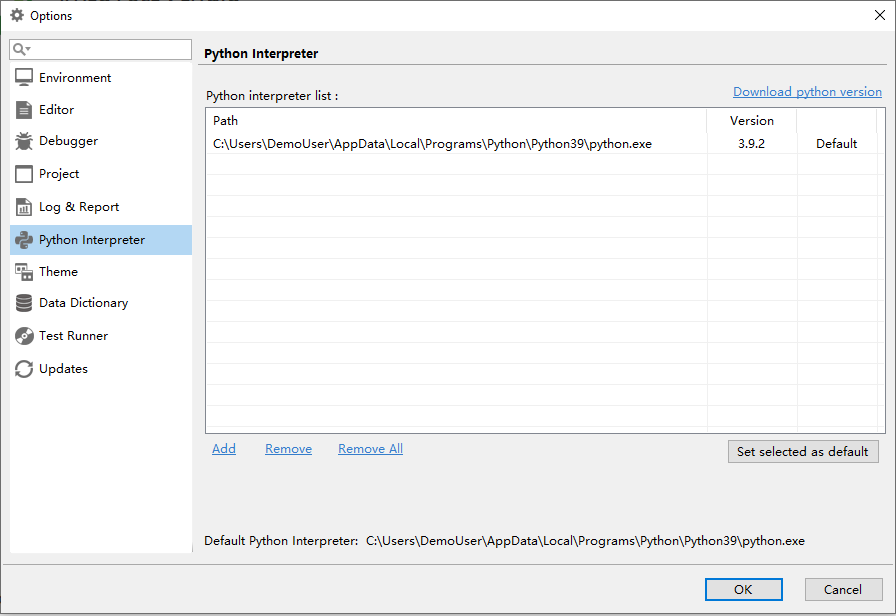
Click "Add" to add the python interpreter path.



You can add multiple python interpreters here. You can select the python interpreter used by the program by setting the default python interpreter. If you do not set the default python interpreter, you will be prompted whether to set the current python interpreter when adding It is the default python interpreter.



After setting the python interpreter as the default interpreter, the program will automatically check and install the python packages required by the program.



After setting, click OK.

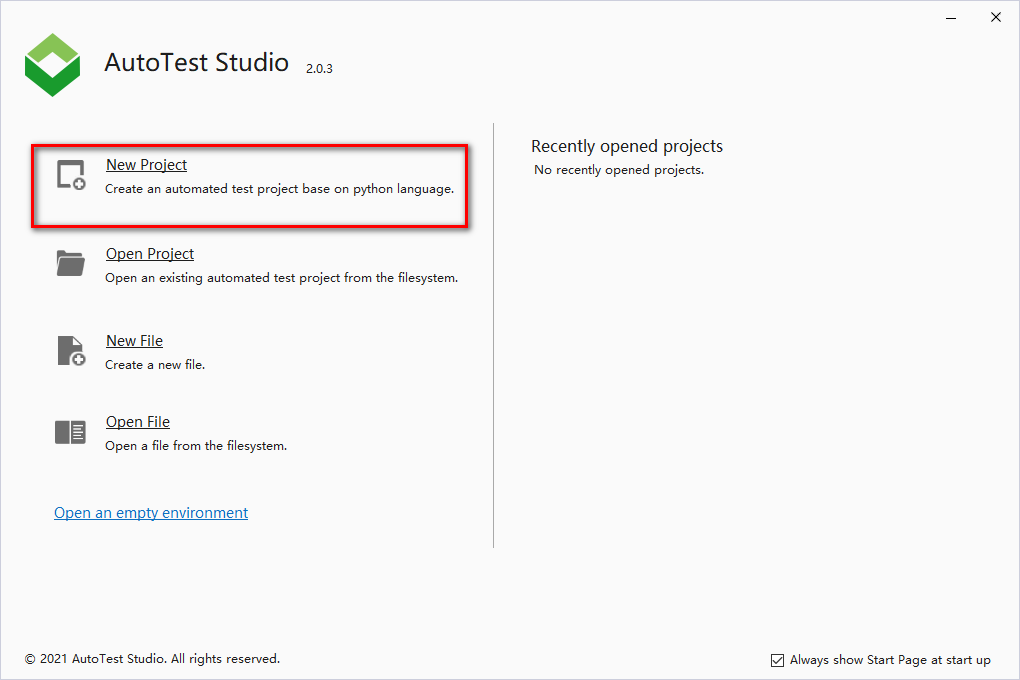
At this point, the installation and basic configuration of AutoTest Studio have been completed.

# AutoTest Studio Getting started tutorials 2: Start with a project

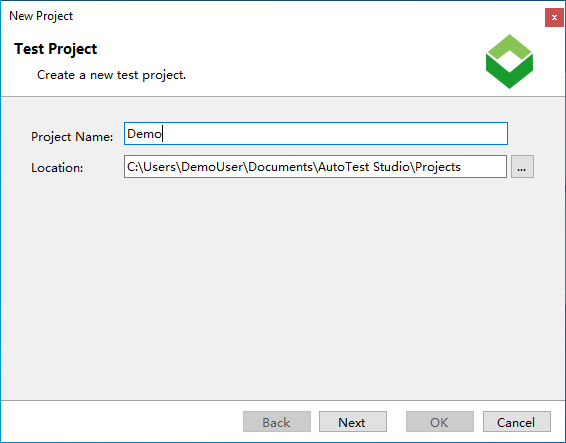
In this section, I will take a simple demo as an example to introduce how to start a test project. The scenario of the demo is to test a web service interface that queries ip address location information, and verify the interface by judging the status code after the interface is called. Is it normal? Interface prototype: http://ip-api.com/json/XXXX (XXXX is the ip address to be queried).

**New test project**

Open AutoTest Studio and select "**New Project**" on the start page to open the new project wizard.



Enter the project name and project save location in the New Project Wizard dialog box. This article takes "Demo" as an example, and the save location is the default path.



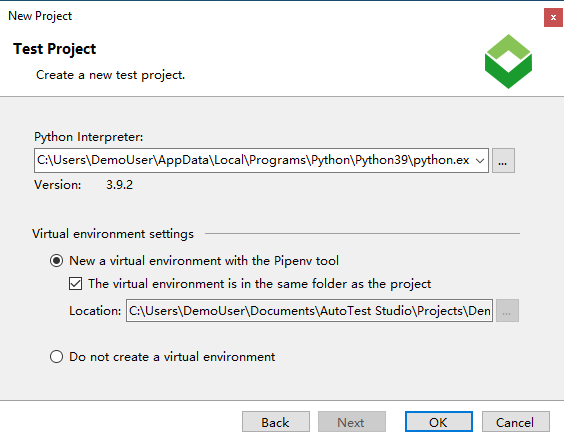
Configure the python environment of the project, "**Python Interpreter**" can select one from the pre-configured python list, or you can select a new python environment through the "**...**" button at the back.

AutoTest Studio supports running in a virtual environment. When you select "**New a virtual environment with the Pipenv Tool**", AutoTest Studio will create a virtual environment based on the selected "**Python Interpreter**". When there are multiple test projects on the same PC At the time, the virtual environment can isolate the python environment in which the project runs, so that each project has an independent python operating environment.

"**The virtual environment is in the same folder as the project**", save the configuration information of the virtual environment and the project in the same directory.

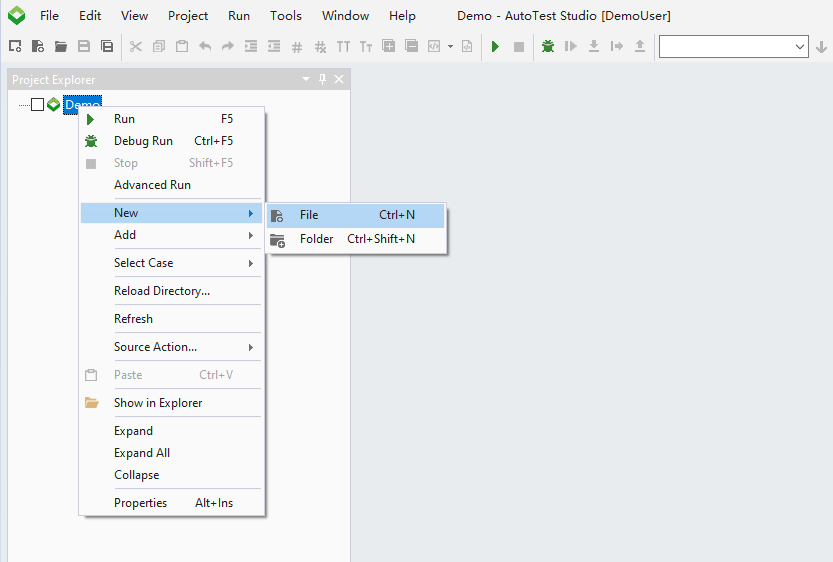
**Note:** After the Python environment configuration setting is completed, if you need to modify it later, you can also modify it in the project properties.

Click the "**OK**" button, AutoTest Studio will automatically create the project, including the initialization of the python environment.



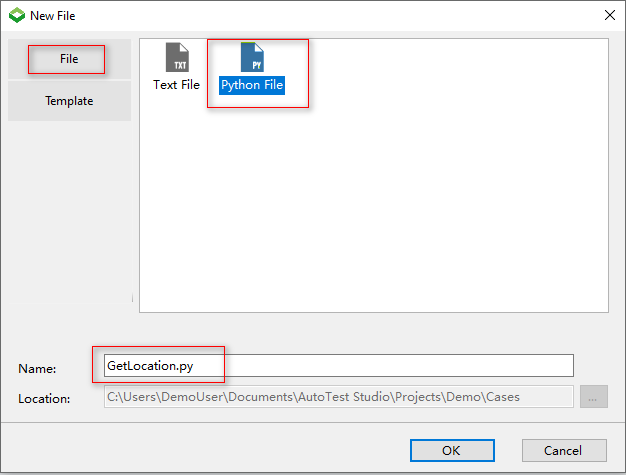
**New test case**

In the main interface of AutoTest Studio, in the right-click menu, click "**New**"-> "**File**" to pop up the test case creation wizard. It should be noted here that only test cases or files created through the right-click menu of the project directory can be associated with the current test project. And appear in the project's directory structure.



Through the creation wizard, select "**File**"->"**Python File**" to create a new test case file, fill in the name of the test case, and the file storage location cannot be changed.

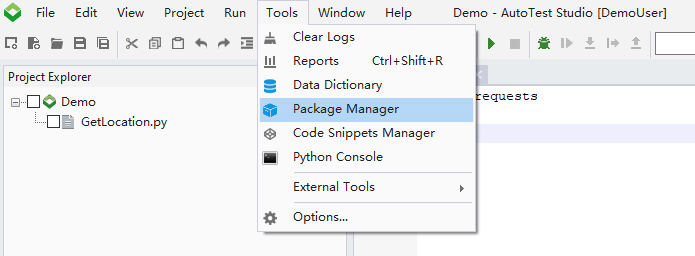
The creation of test cases can also be created through templates. AutoTest Stduio allows users to create different template files according to their needs and add them to the program.



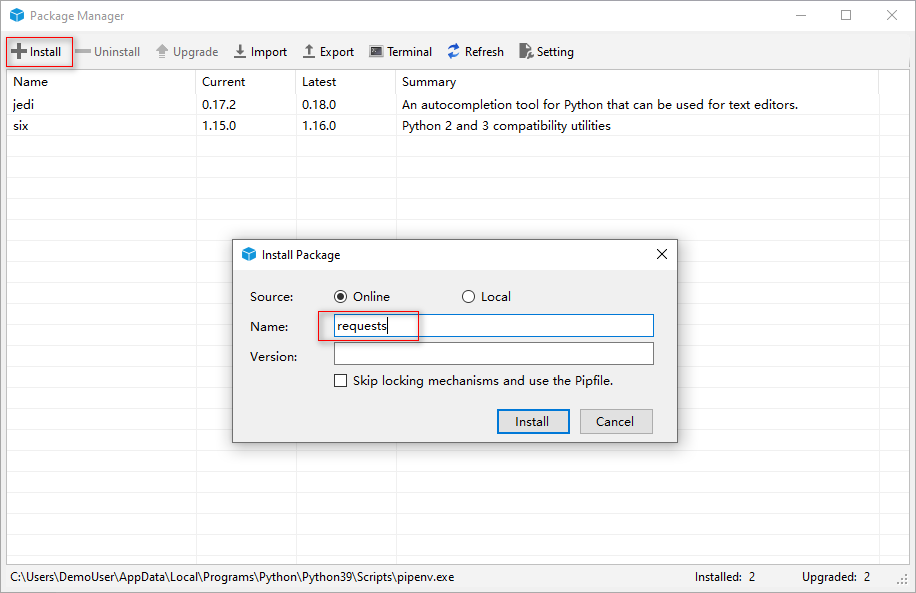
**Install the "requests" package**

Because the test cases in the demo will use the requests package of python, it is necessary to add the requests package through the python package manager tool. In the subsequent chapters, I will introduce the use of Package Manager in detail.

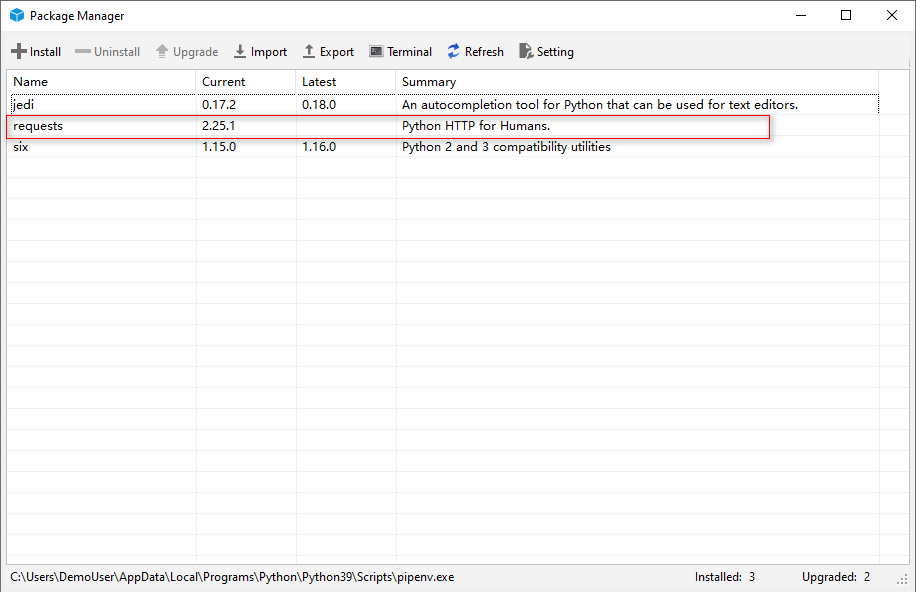
Click "**Tools**" -> "**Package Manager**" in the main menu of AutoTest Studio to open the python package manager.



In the "Package Manager", click "**Install**" in the pop-up dialog box, enter "requests", click "**Install**", "Package Manager" will automatically follow the requests package and related dependencies.



After the installation is successful, you can see the installed requests package in the list.



**Write use case code**

Enter the following code in the test case. The test code is relatively simple. You can query the ip address information of 8.8.8.8 through the interface and judge the result through AssertEqual.

import requests

from autotest import \*

SetCase("TEST-1","Get location by ip","1")

ip="8.8.8.8"

ipQueryUrl="http://ip-api.com/json/{0}".format(ip)

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

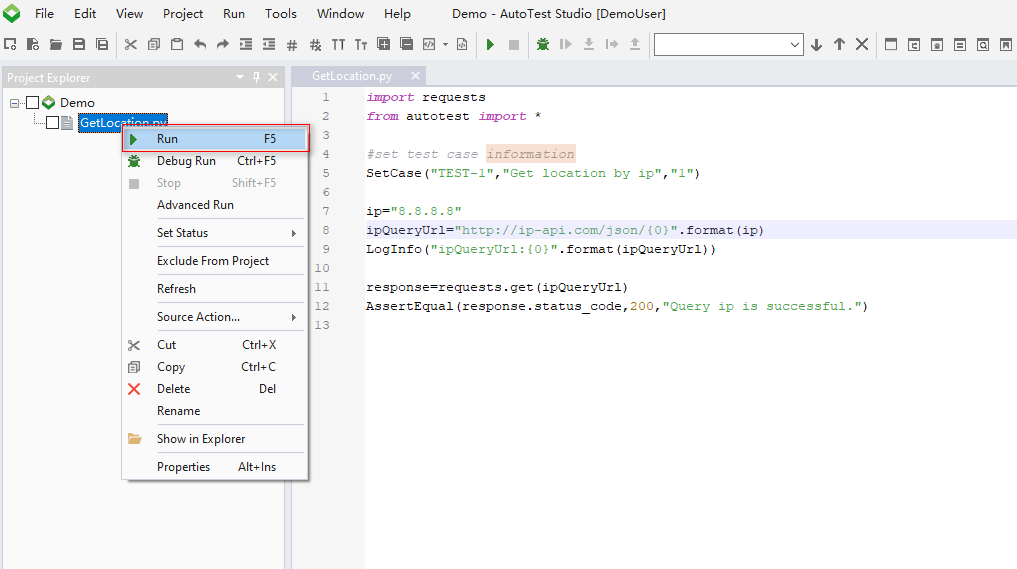
response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.")

**Run test cases**

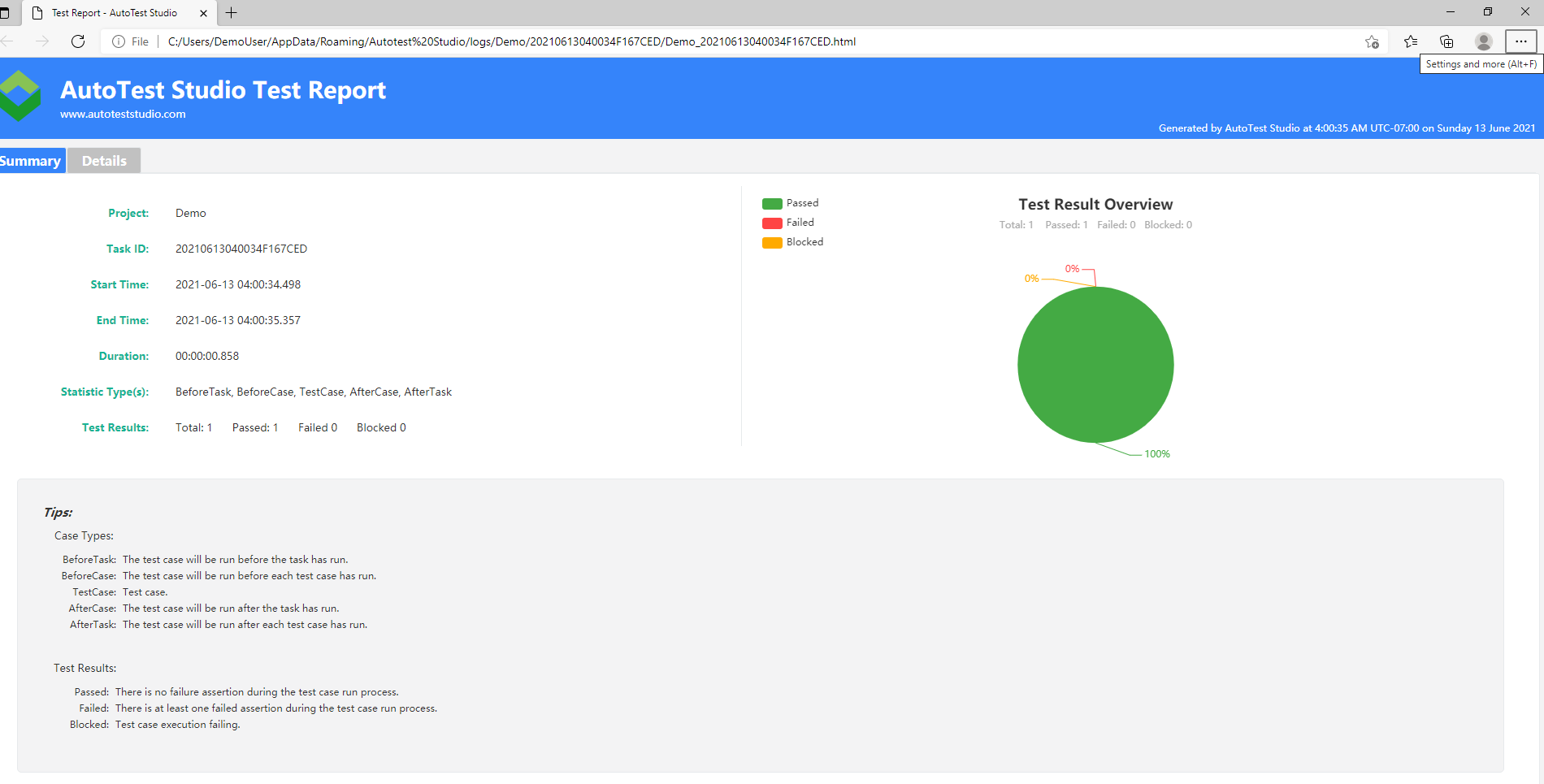
Select the use case in the project's directory structure, click "**Run**" in the right-click menu, and run the test case. After the run is completed, AutoTest Studio will automatically open the test report.

Note: To run a single test case, directly select the case and click "**Run**" in the right-click menu. If you are running multiple test cases, you need to select the test case you want to run, and click "Run" in the right-click menu of the parent directory .

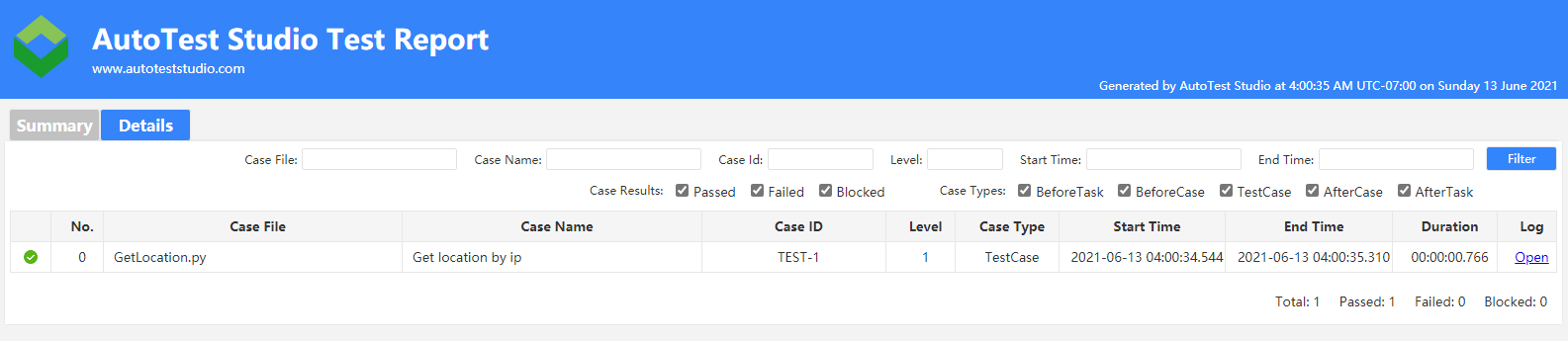


**View test reports and logs**

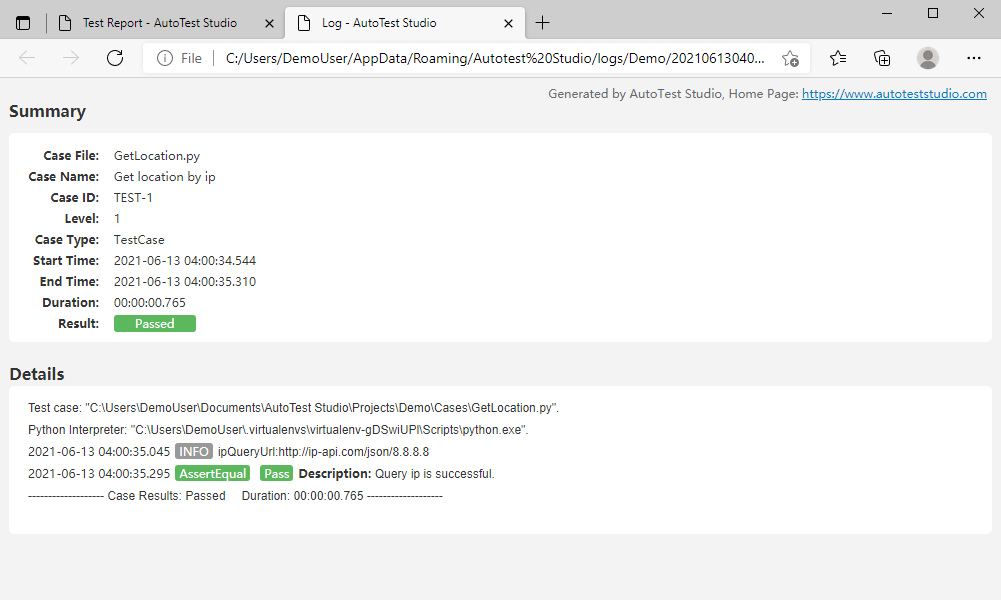
The test report includes the basic information and statistical information of the current task running.



Click "**Details**" of the test report to view the running information of the test case.



On the right side of the test case list item, click "Open" to open the log file of the test case operation.



Since then, it has been introduced how to create test tasks and test cases through AutoTest Studio, and run the test tasks.

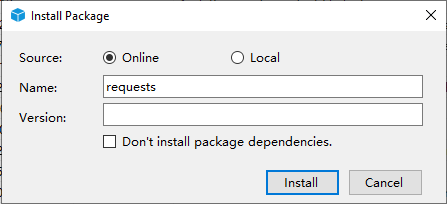
# AutoTest Studio Getting started tutorials 3: Package Manager

The python environment operated by Package Manager is the python environment of the current project. If a virtual environment is used, it refers to the python package of the virtual environment. For users, there is no need to care about whether the python environment is a virtual environment. Installation, uninstallation, and viewing operations are all identical.

As an example, the requests package used in the previous section of this article describes how to install, uninstall, view, upgrade, and configure the installation source.

**Installation Online**

Online installation, that is, install the python package directly through the network, this method is recommended.



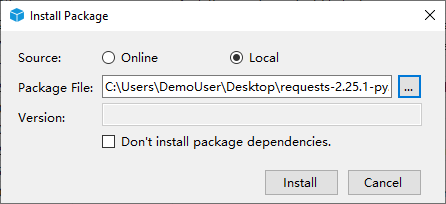
**Name**: python package name;

**Version**: The version of the package. When it is empty, the most detailed version is installed by default;

**Don’t install package dependencies**: Don’t install related dependencies, only install the specified packages. It is recommended not to check.

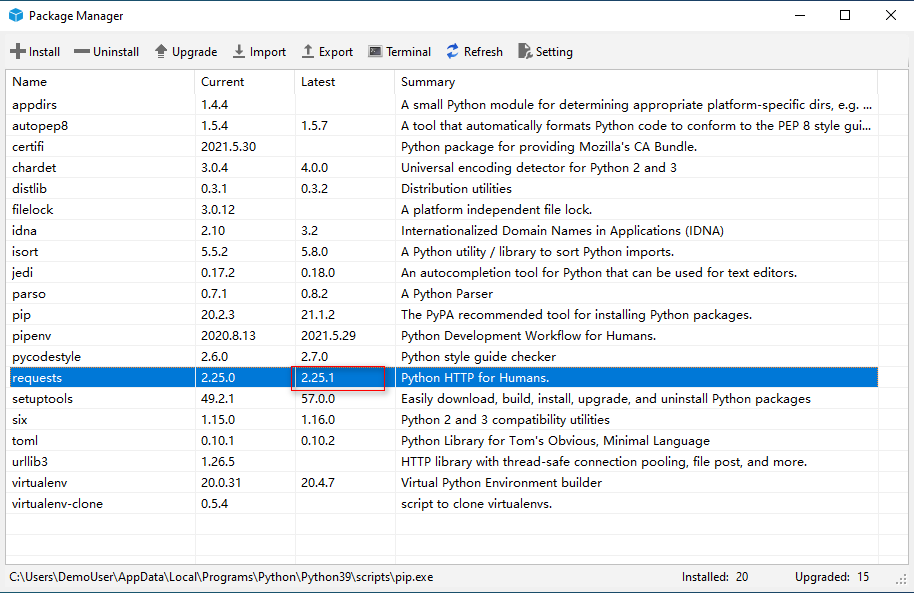
**Installation Offline**

If the package already exists locally, you can use the local installation, select "**Local**" and click “**...**"to select the path of the Python package.

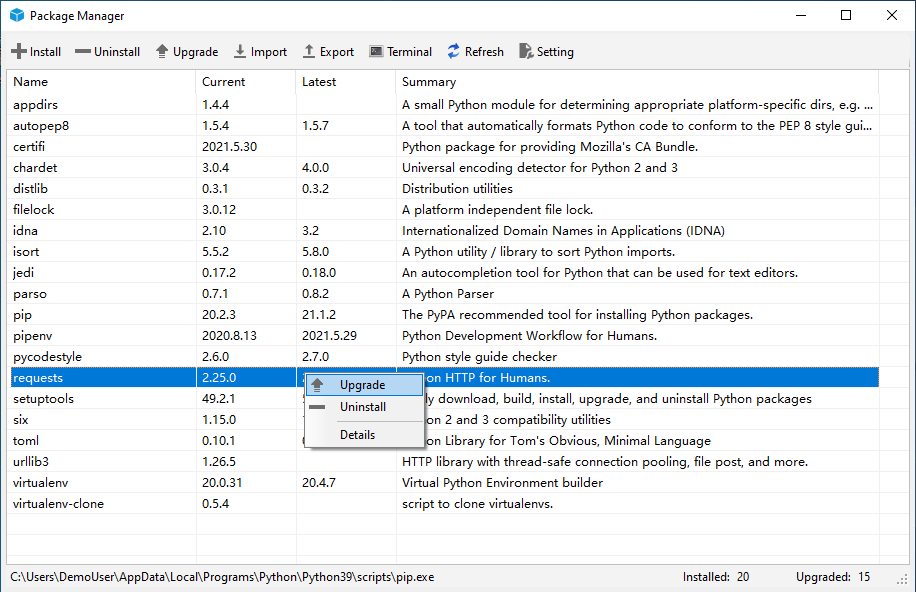


**Upgrade**

Package Manager automatically checks for the latest version and lists the thinnest versions in Latest, and to demonstrate the upgrade, try installing version 2.25.0 of requests first, as follows.



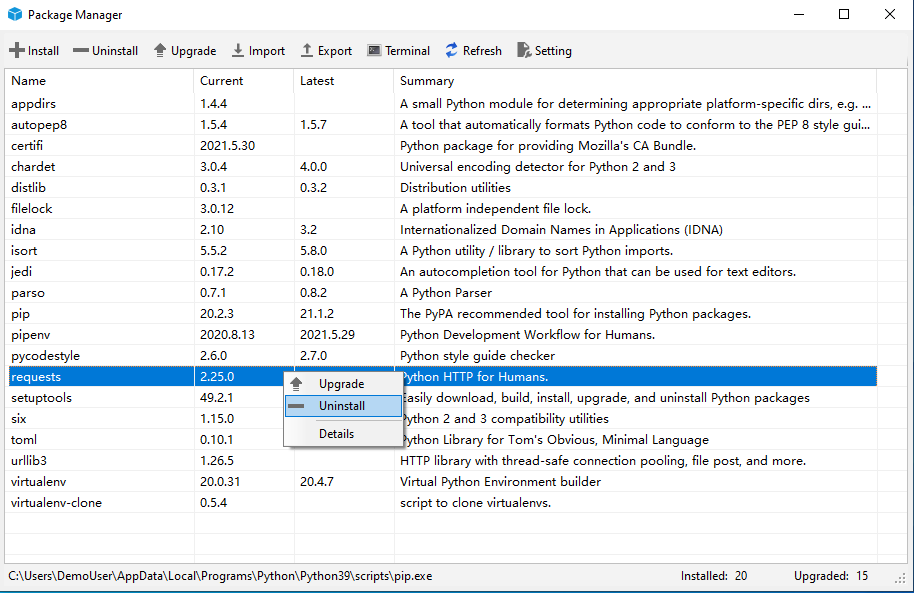
Select request in the list, click "Upgrade" and Package Manager will automatically upgrade to the latest version.



**Uninstall**

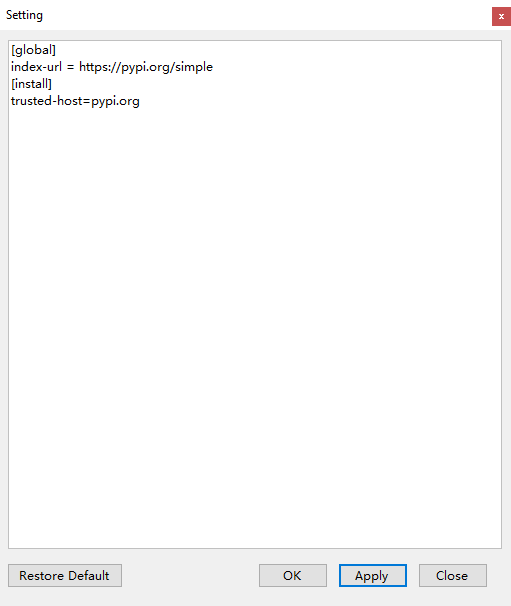
Select the package to be uninstalled and click "**Uninstall**" to uninstall the package.

**Note**: When uninstalling a package, dependent packages will not be uninstalled.



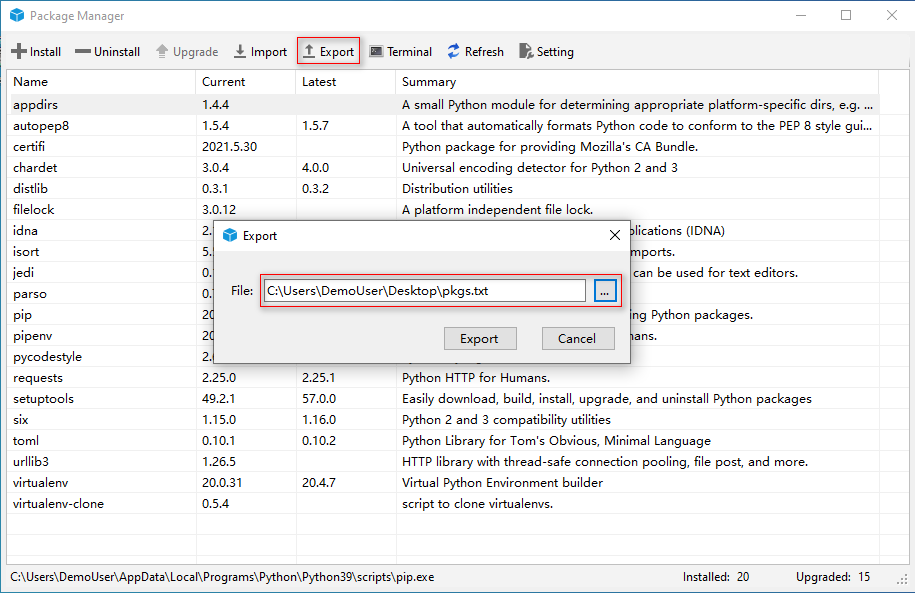
**Configure the installation source**

Package Manager allows users to configure the installation source, which is actually to configure the pip.ini file. Click "**Setting**" on the toolbar. After editing the installation source information in the pop-up dialog box, click "**OK**" to save. If you need to restore to the default installation source, you can click the "**Restore Defualt**" button at the bottom. The default installation source is the official python installation source.



**Export package list**

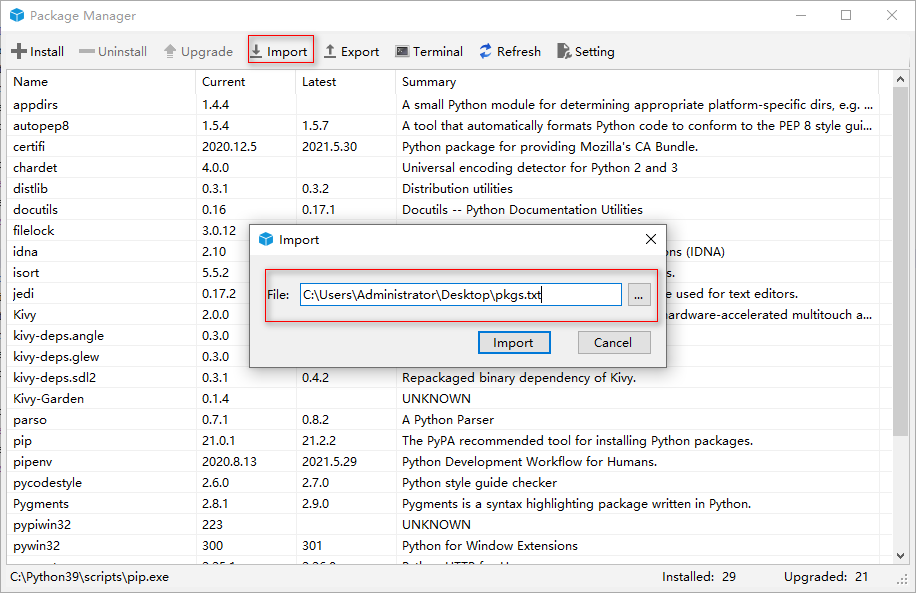
Package Manager can export the list of installed packages in the current environment.



The exported list is actually a text file containing the python package name and version.

**Import package list**

As shown in the figure below, the package in the text file can be imported into the current environment, and the import and export functions are mainly used to migrate the current environment package information to another python environment. It should be noted that when the package is imported, it is actually installed online, so the environment needs to be connected to the Internet.



Package Manager avoids the cumbersome management of packages through the command line. If you are familiar with pip and pipenv commands, you can also manage python packages directly through the command line.

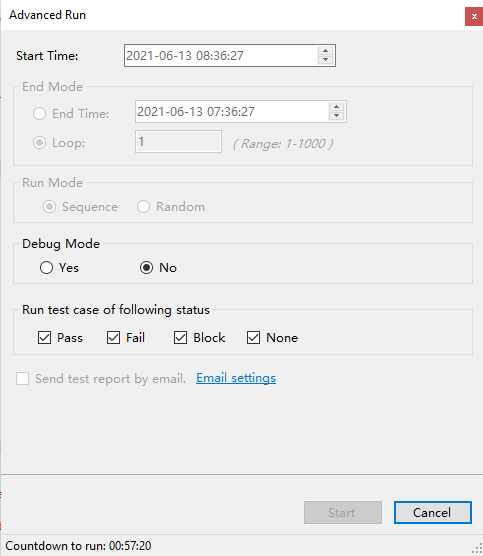
# AutoTest Studio Getting started tutorials 4: Task Scheduling

AutoTest Studio has a powerful task scheduling engine that allows users to flexibly run test tasks, such as running tasks regularly, running tasks quantitatively, and running test cases in specified states.

For demonstration purposes, we will copy 5 copies of the test cases in the second section, and modify the caseID to: TEST-1, TEST-2, TEST-3, TEST-4, TEST-5.

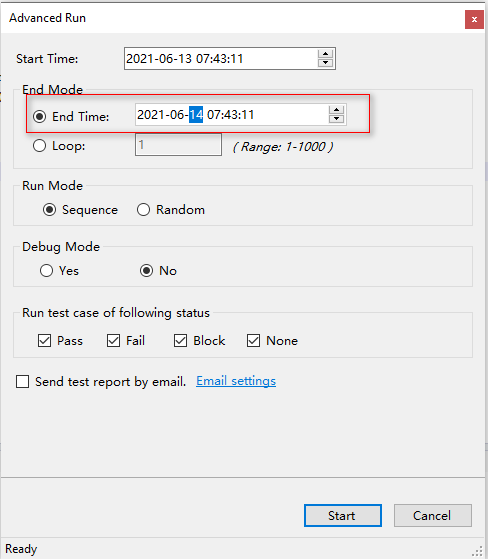
**Timed Run**

When the set Start Time is greater than the current time, AutoTest Stduio will enter a countdown and will not run until the specified time is reached. If the start time is less than the current time, the task will be run immediately.



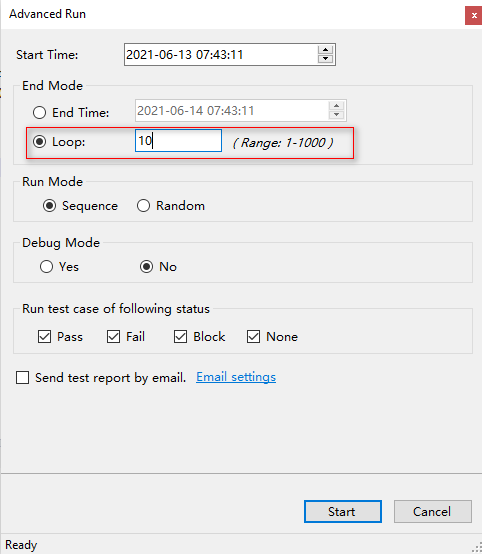
**Timing ends**

AutoTest Studio allows you to set the end time. When the end time is reached, the program immediately terminates the task. Here you don’t have to worry that AutoTest Studio will end the task violently. In fact, after the end time is reached, AutoTest Studio will run the current test case in Terminate the task.



**Specify the number of runs**

In addition to providing a specified time to end the task, AutoTest Studio also provides a specified number of runs. As shown in the figure below, AutoTest Studio will terminate the task after running all the use cases in the task 10 times.



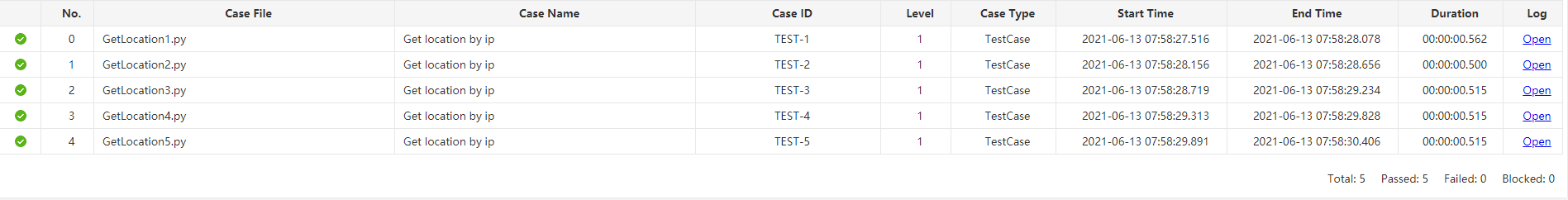
**Operating mode**

AutoTest Studio will run all use cases sequentially by default, but it also provides a random way to run use cases.

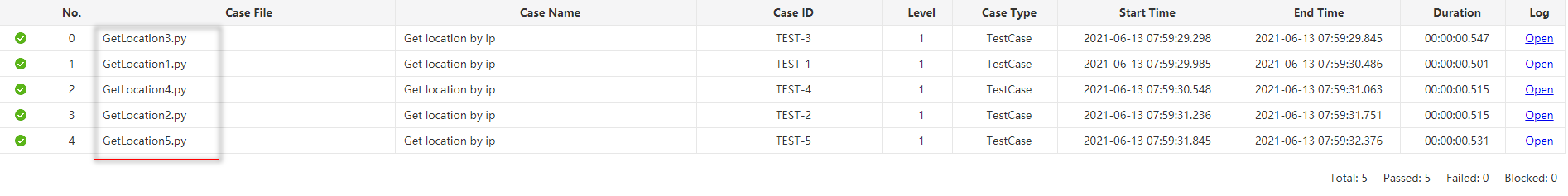
**Sequence**, run the use cases sequentially;

**Random**, run the use cases in a random manner.

The result of the sequential operation:



Random run results

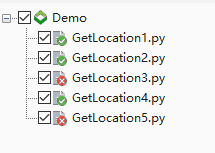


**Tuning operation**

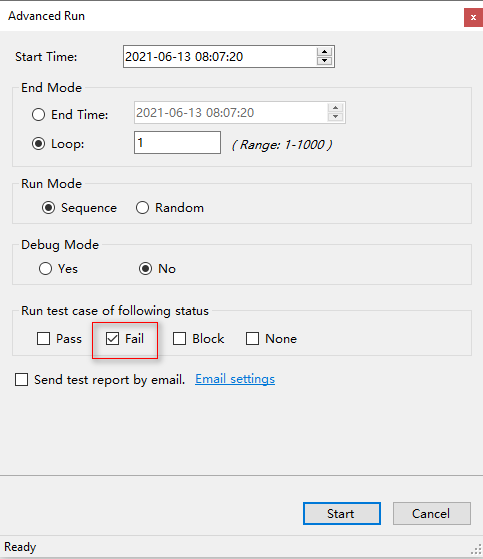
When Debug Mode is set to "Yes", AutoTest Studio can run test cases in a debug mode. This function is very useful when we locate some "strange" problems. We can let the use case enter a breakpoint when it runs to a special condition.

**Run the use case in the specified state**

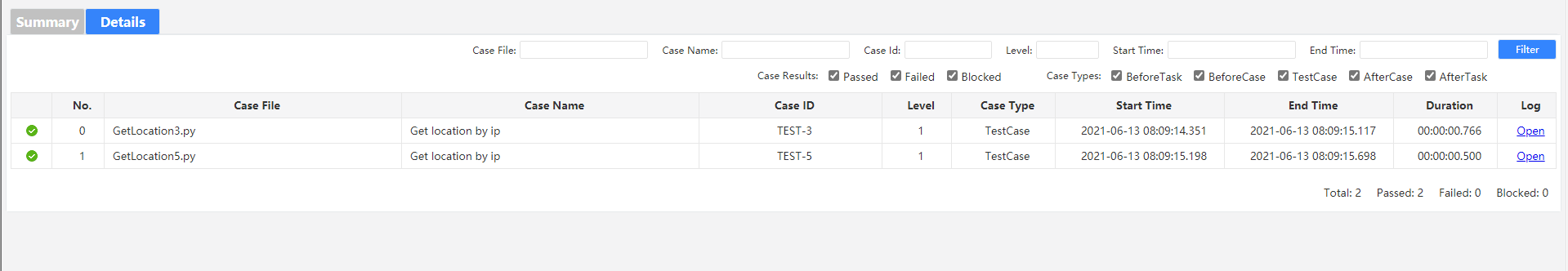
AutoTest Studio can run use cases in a specified state, such as the use cases that run the previous round of failures. We assume that the results of the previous run are as shown in the figure below.



Set the "**Run test case of following status**" of "**Advanced Run**" to run only the use cases with the "Fail" status.

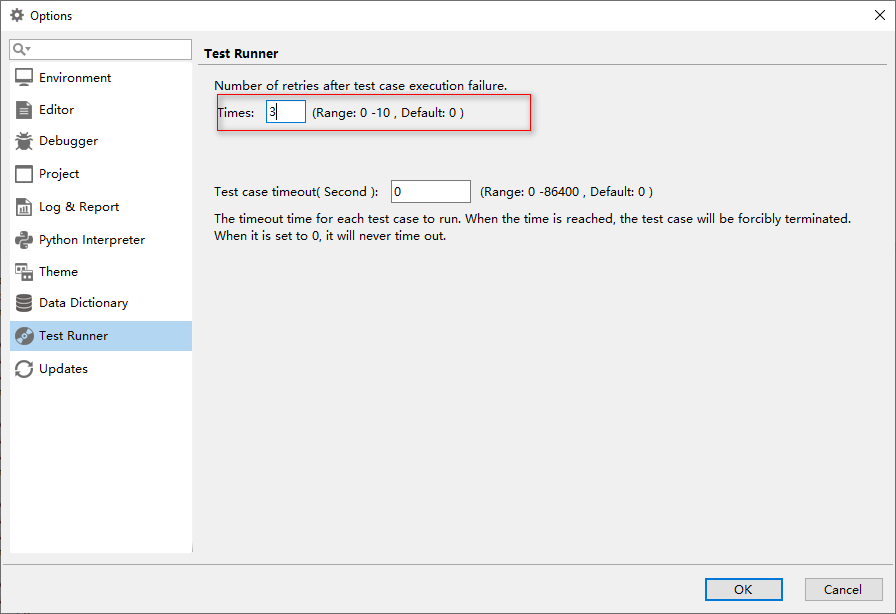


The following is the result of the operation. You can see that AutoTest Studio only ran "GetLocation3.py" and "GetLocation5.py", which are the use cases that failed in the previous run.



**Rerun after failure**

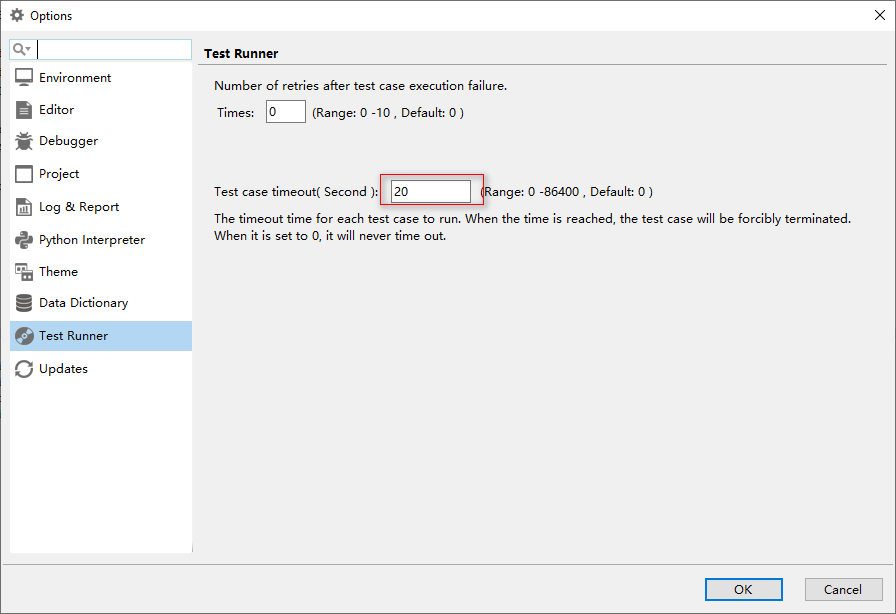
AutoTest Studio provides a global configuration parameter that allows the use case to run again after a failure, and can specify the number of repeated runs. The default is 0, that is, no repetition.



**Use case run timeout setting**

In many cases, the test case may be blocked due to some abnormal reasons, but we do not want the entire task to be blocked due to the blocking of this use case. Therefore, we can set the test case timeout period to force the suspension of the blocked test case and continue to run other cases. Use case, but it should be noted here that the timeout period cannot be set too small, because even when the use case is running normally, if the timeout period exceeds this timeout period, the operation will also be terminated.

Suppose we set the timeout period of the use case to 20 seconds, and we use time.sleep(25) to block the use case in the use case.

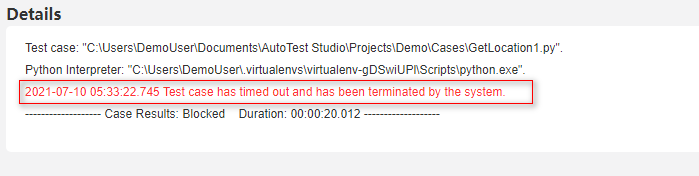


**Sample code:**

import time

time.sleep(25)

Run log results:



# AutoTest Studio Getting started tutorials 5: Preset Use Cases

Pre-built use cases allow you to do some additional things before and after the test task or test case runs, for example, check the environment or set certain parameters before the task runs, and clean the environment or restore parameter settings after the task runs.

AutoTest Studio provides four types of preset use cases.

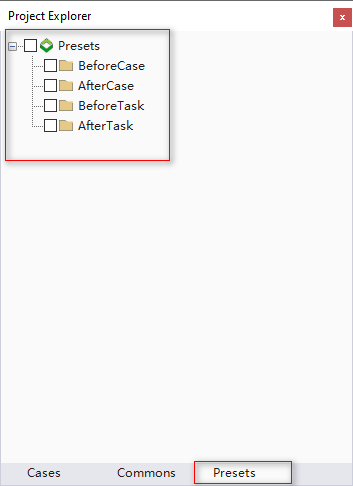
**BeforeTask**, it is executed at the beginning of each task, that is, it is executed before all use cases are run.

**BeforeCase**, execute before each use case runs.

**AfterCase**, execute after each use case is run.

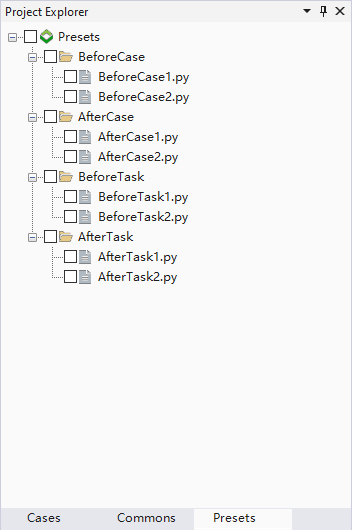
**AferTask**.Execute at the end of each task, that is, execute after all use cases have run.

After creating a new project, there will be four default folders in the "Presets" interface by default, corresponding to the four preset use case types, as shown in the figure below.

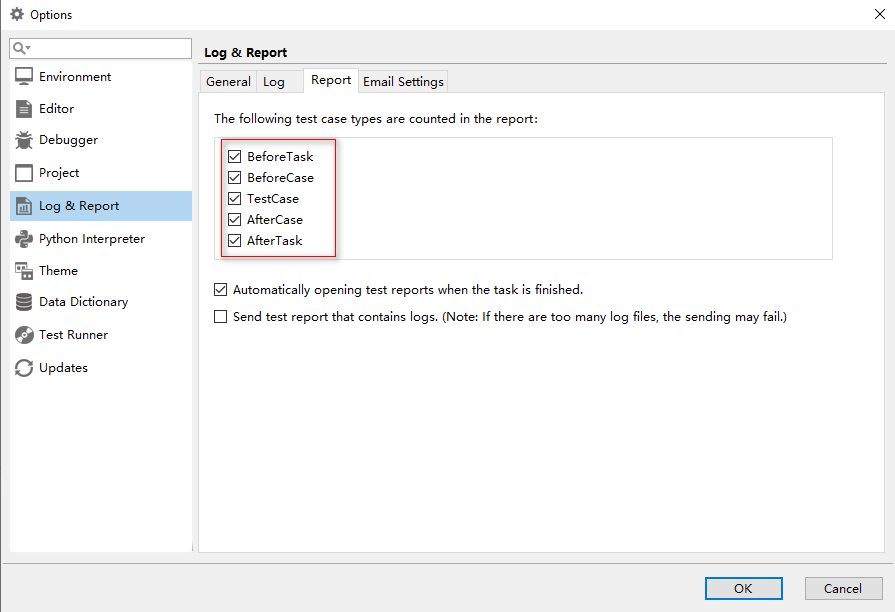


Users only need to write the corresponding use cases in the corresponding folder. You can write multiple use cases in the same type, or none, depending on your needs.

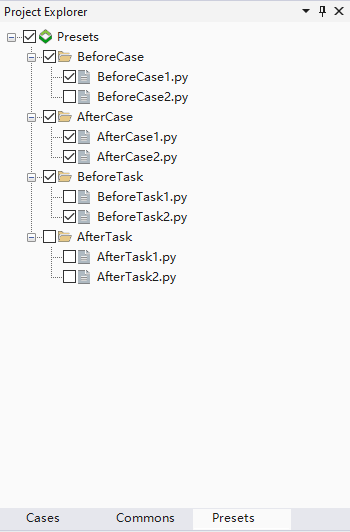
Here we write two use cases for each type to demonstrate the calling sequence of the preset use cases. The content of the use cases can be written at will, and the way of writing preset use cases is no different from that of ordinary use cases.



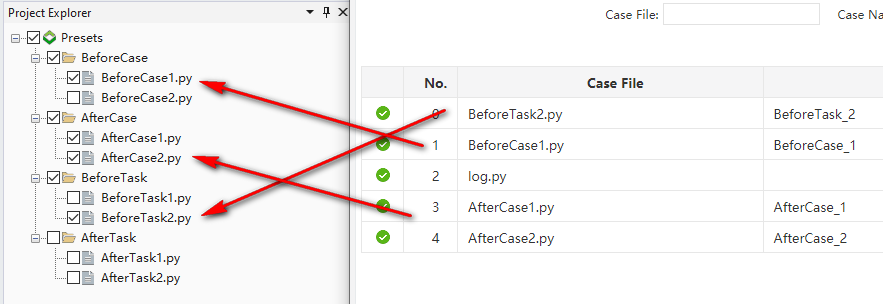
In the settings, we choose to count all types.



Before the task is executed, we only need to check one or more preset use cases that need to be run. Of course, we can also select none of them. For demonstration, we check as follows.

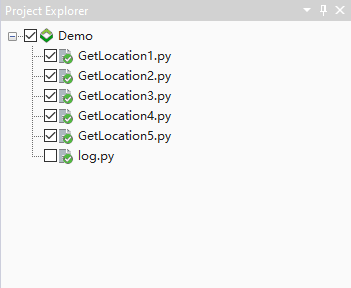


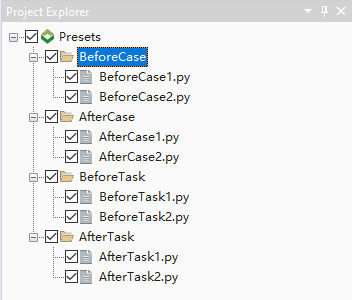
Select the use case to execute the task once, and the execution result is as follows.



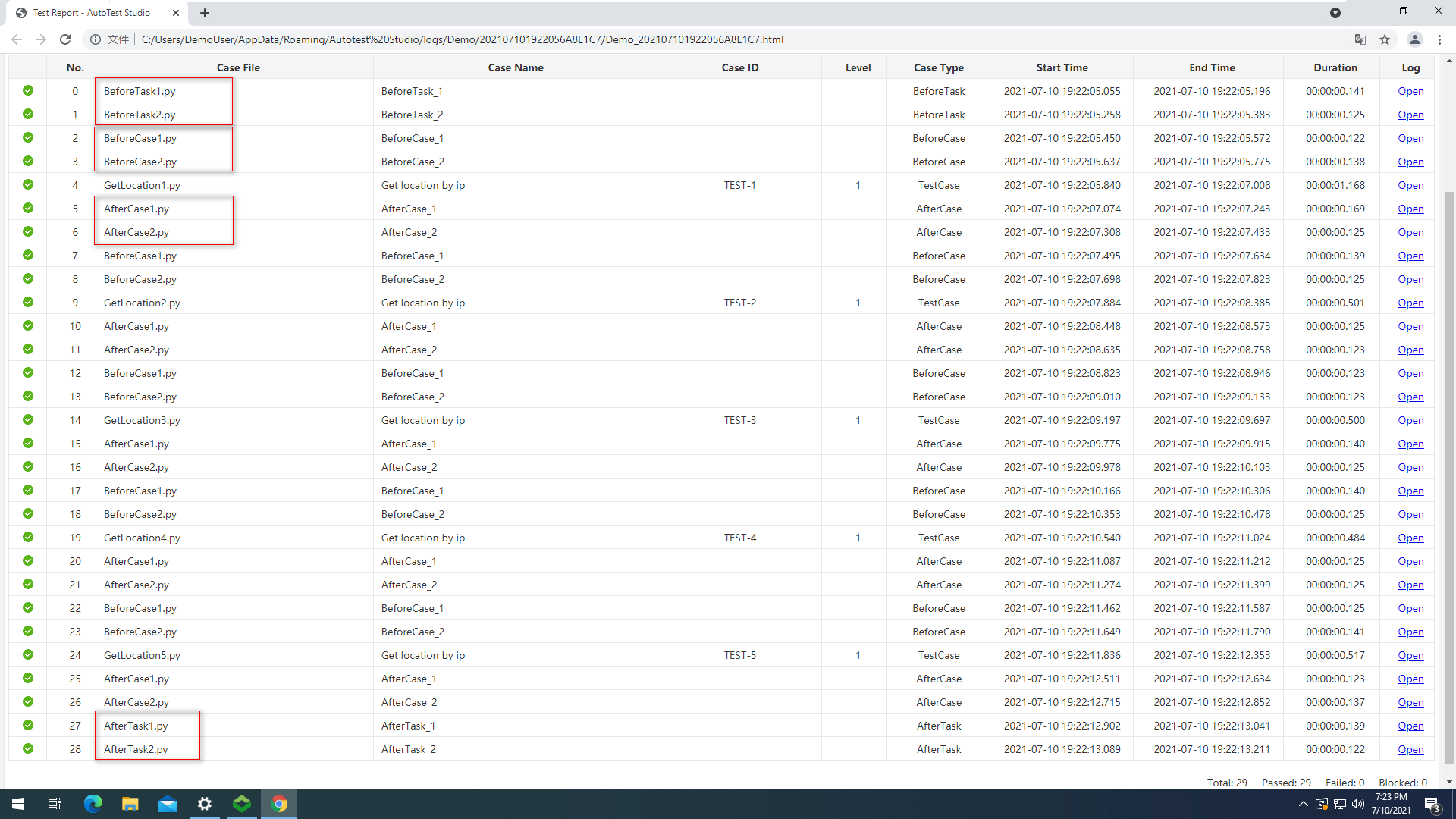
As can be seen from the above screenshot, the preset use cases we selected are executed in accordance with the correct logic. There is no preset use case to be executed in the AfterTask directory, so there is no corresponding record in the log.

Let's execute it again, check all the preset use cases in the example, and check the multiple use cases written in the previous chapters to see the effect.





The results of the operation are as follows (note that the preset use cases are executed before and after each use case is run).



# AutoTest Studio Getting started tutorials 6: Logs and Reports

Logs and reports are one of the important criteria for judging whether an automated testing framework is excellent. AutoTest Studio also has a complete log module and report module. Through these logs and reports, you can evaluate and analyze the details of the automated test task operation.

**Log**

AutoTest Studio has four levels of logs, namely DEBUG, INFO, WARN, ERROR, these four levels of logs, users can flexibly apply according to their actual situation, and can decide which levels of logs to display according to their needs. In practical applications, we can output all logs when developing and debugging test cases, but when the task is running, selectively output some logs to reduce the amount of log output.

Take a piece of test code as an example to introduce how to output the log and configure the displayed log level.

from autotest import \*

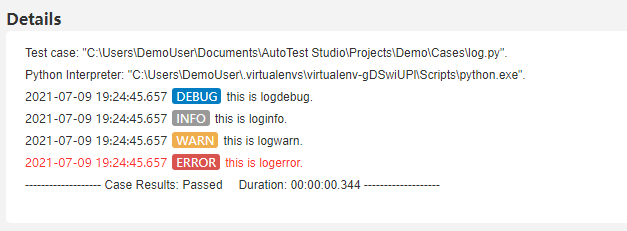
LogDebug("this is logdebug.")

LogInfo("this is loginfo.")

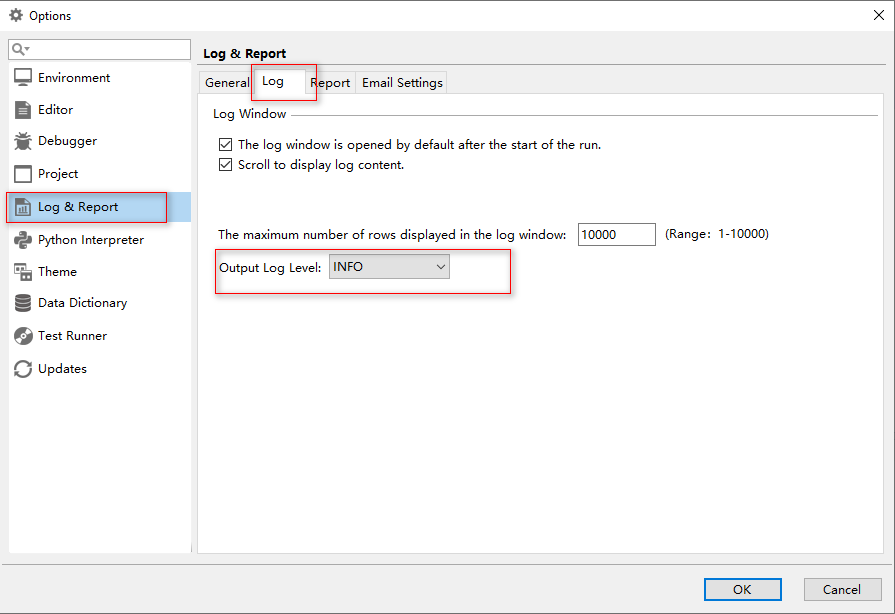
LogWarn("this is logwarn.")

LogError("this is logerror.")

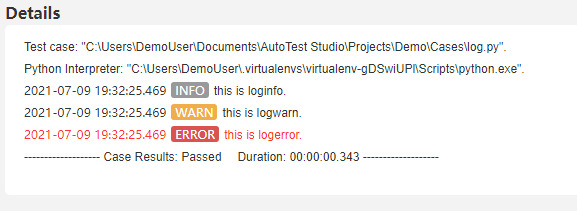
After normal operation, AutoTest Studio will display four levels of logs by default.



In the configuration manager, we can customize the output log level. For example, we only want to output INFO and above logs. The configuration is shown in the figure below.



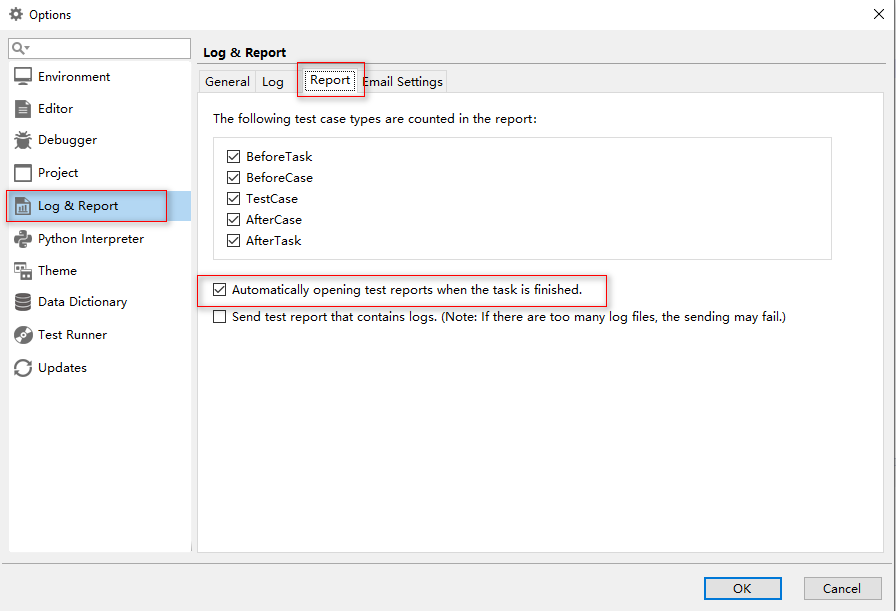
After re-running the code, the output log is as follows, you can see that there is no DEBUG level log.



**Report**

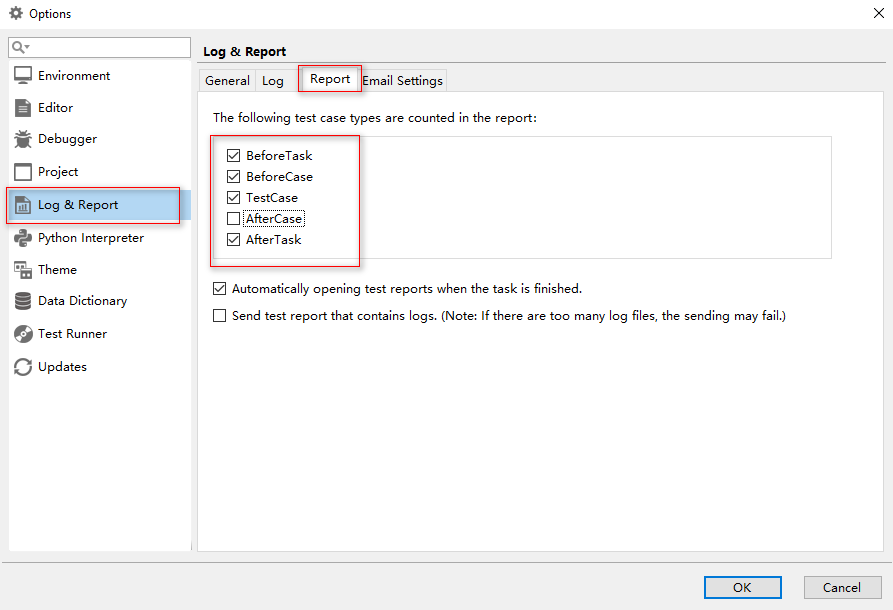
AutoTest Studio takes tasks as the running unit, and each run is a task, even if you only run one test case.

After the task is completed, the test report of this task will be automatically opened by default. Of course, you can also set not to open the test report automatically. The setting method is as follows.

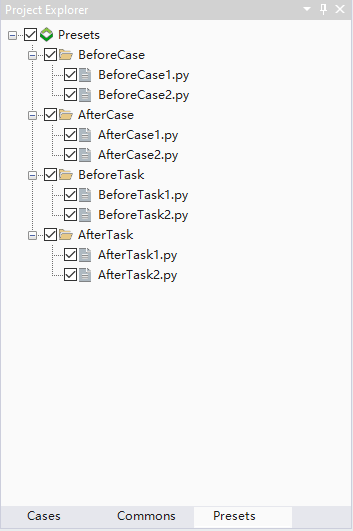


**Set report statistics type**

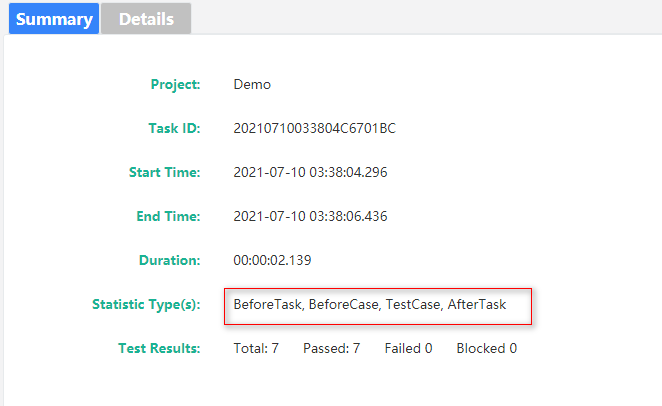
The user can decide the use case type of the report statistics according to the needs, but it must be set before the task is run. The settings are as follows. For comparison, we assume that the AfterCase type is not counted.



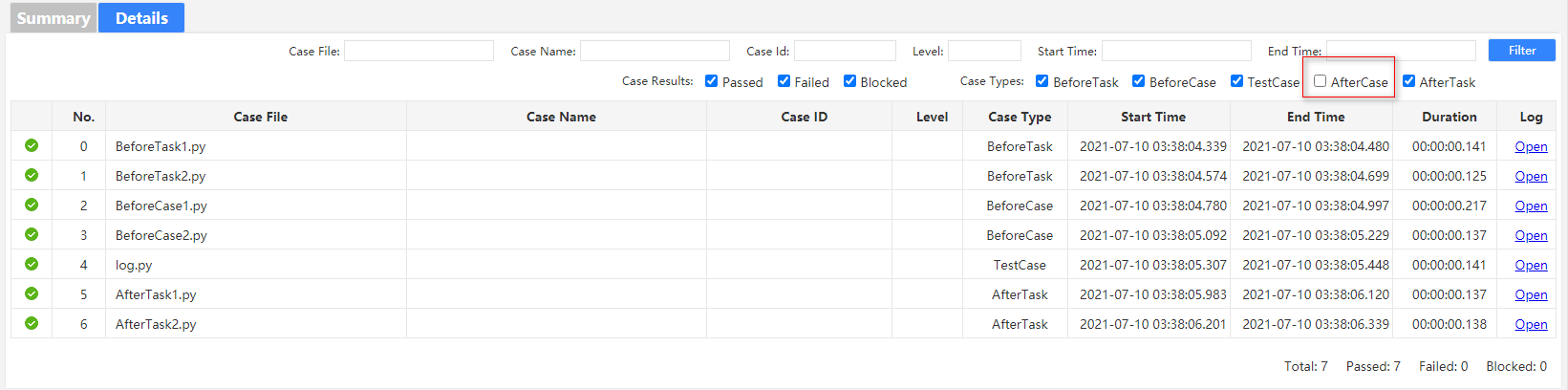
**Select a preset use case in "Presets".**



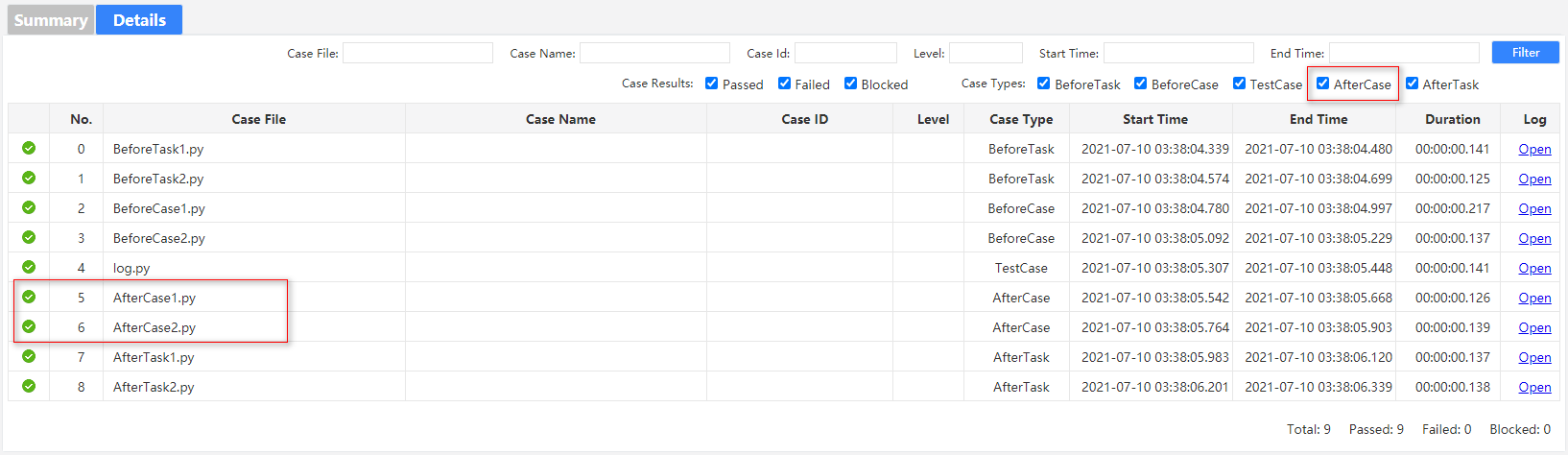
Run the task once, as shown in the figure below, in the statistical report, there is no AfterCase.



In the test log record, the default is the same as the report statistics, and the types without statistics will not be displayed, as shown in the figure below.

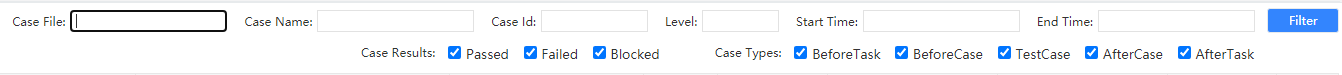


But this does not mean that the log of this type of AfterCase is lost. You can select the type without statistics in the filter and filter again.

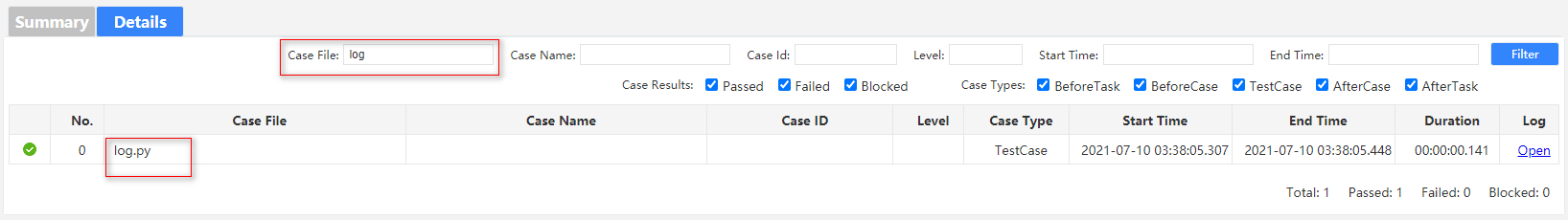


**Log filtering**

In the test report, the user can filter the log records through a variety of conditions. The filter conditions are as follows.

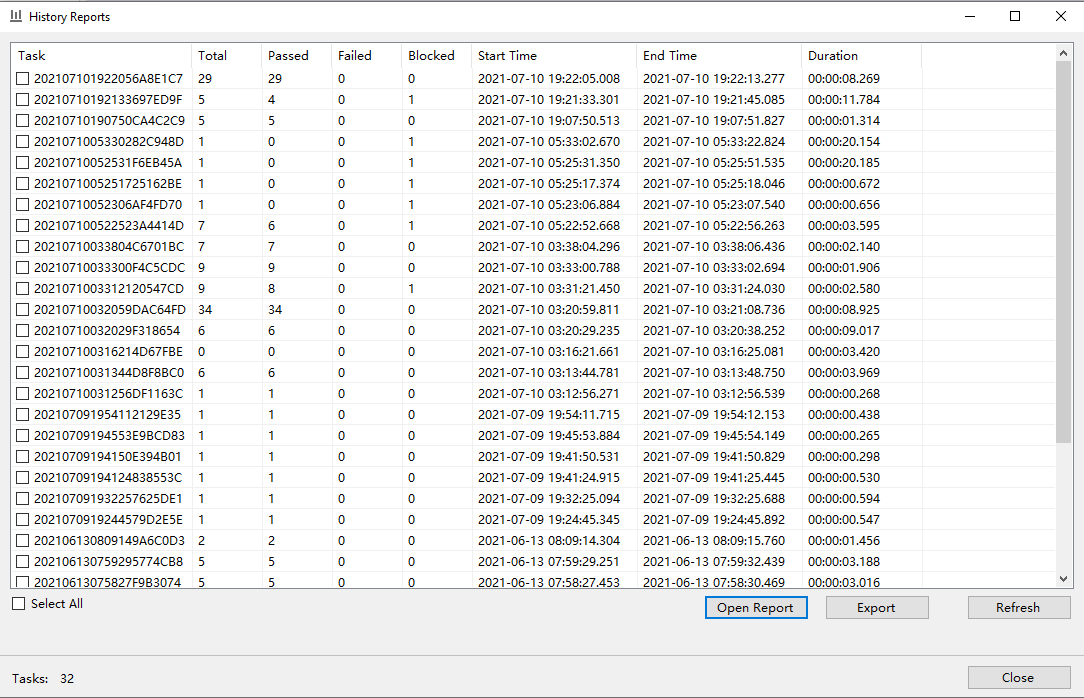


Suppose we want to filter the use cases that contain the "log" keyword in CaseFile.



**Historical report**

In the toolbar "**Tools**" -> "**Reports**", you can open the history report list. With this tool, users can open the task reports that have been run before, or export one or more task reports in the form of a compressed package through the Export button.



# AutoTest Studio Getting started tutorials 7: Data Dictionary

AutoTest Studio provides a data dictionary module with excellent performance. Unlike other test frameworks that use text to store data, AutoTest Stduio uses Sqlite as its internal storage engine. The performance and stability of Sqlite are recognized in the industry. According to the official introduction, AutoTest Stduio can still provide good read performance even if it stores millions of records.

It should be pointed out that the data dictionary of AutoTest Studio is read-only during the running of the task, that is, users cannot write or modify data into the data dictionary through the function interface, and can only write or modify data in advance through the Data Dictionary tool. This design ensures that the test data will not be polluted during the running of the task.

The data dictionary of AutoTest Studio organizes the data hierarchical relationship in a directory structure. This design makes the data dictionary organization relationship with good readability. In the data dictionary, there are two concepts, data set (Dataset) and data item (Dataitem).

**Dataset**, a collection of the same type of data, such as the basic information of a test user, the Dataset is globally unique, that is, in the same data dictionary, the name of the data set cannot be repeated, even in different directories.

**Dataitem**, a data record under the data set, that is, a key-value record.

When the project is created, AutoTest Studio creates an empty data dictionary for each project by default.

The data dictionary module provides two operation interfaces.

Read the entire data set and return a python dictionary structure.

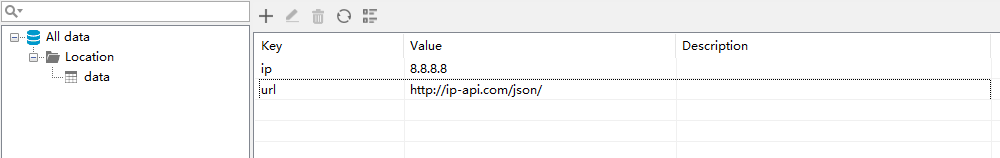
GetDataset (dataset)

Read the value corresponding to the key in the data set.

GetDataValue(dataset, key)

Let's take the demo in the previous chapter as an example, store the values ​​of the two variables directly defined in the demo, ip and ipQueryUrl, in the data dictionary, and then read the values ​​of these two variables through the data dictionary interface.

First, we first create a directory (Location), then create a data set (data), create two new data items ip, ipQueryUrl under "data", as shown in the figure below.



Modify the original code code.

Original code:

import requests

from autotest import \*

#set test case information

SetCase("TEST-1","Get location by ip","1")

ip="8.8.8.8"

ipQueryUrl="http://ip-api.com/json/{0}".format(ip)

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.")

Modified code:

import requests

from autotest import \*

#set test case information

SetCase("TEST-1","Get location by ip","1")

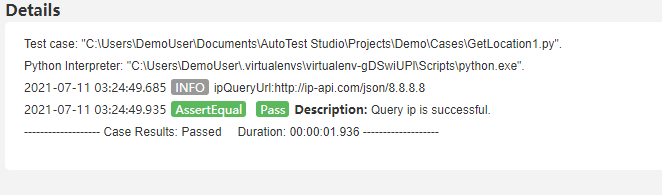
ipQueryUrl="{0}{1}".format(GetDataValue("data","url"),GetDataValue("data","ip"))

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.")

The result of the operation is as follows. From the log, it is consistent with the original code.



Modify the code again, this time, we read the entire data set.

import requests

from autotest import \*

#set test case information

SetCase("TEST-1","Get location by ip","1")

data=GetDataset("data")

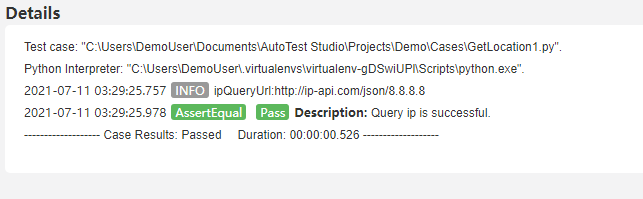
ipQueryUrl="{0}{1}".format(data["url"],data["ip"])

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.")

Run result:



As can be seen from the demo above, the use of the data dictionary is very simple.

# AutoTest Studio Getting started tutorials 8: Code Debugging

AutoTest Studio has a built-in powerful code debugging engine, which is very useful in the process of test case development.

There are many code debugging methods, but in general, the three commonly used steps are breakpoints, monitor variables, and modify variables. This section takes a simple demo as an example to introduce the use of AutoTest Studio's debugging function.

Create a new test case named "debugtest.py" and enter the following content.

from autotest import \*

a=1

b=2

c=3

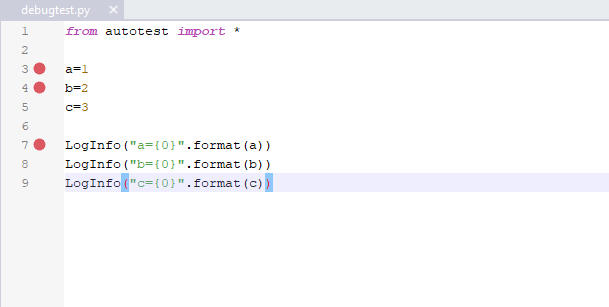
LogInfo("a={0}".format(a))

LogInfo("b={0}".format(b))

LogInfo("c={0}".format(c))

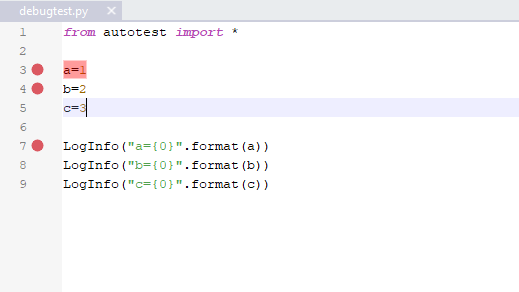
**Breakpoint**

Before the code runs, we set some breakpoints.



**Run the code**

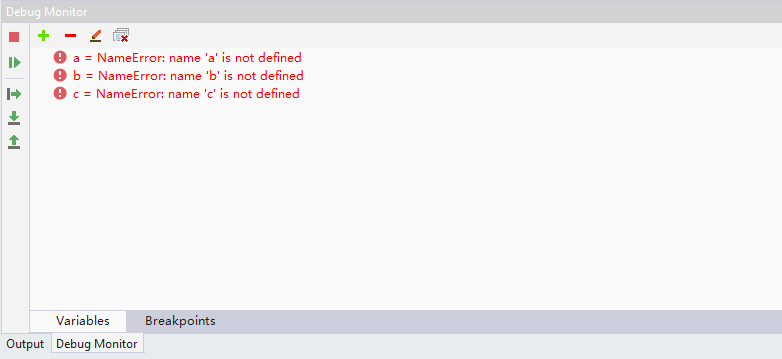
Run the code to the first breakpoint in "**Debug Run**" mode.



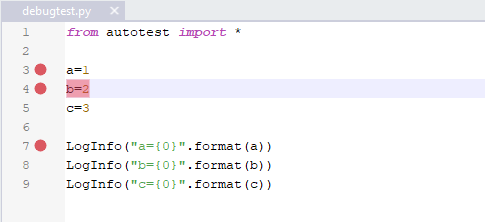
**Monitor variable**

When running the code to the first breakpoint, we create three new monitoring variables, namely a, b, and c. Of course, you can also create new monitoring variables before the code runs.

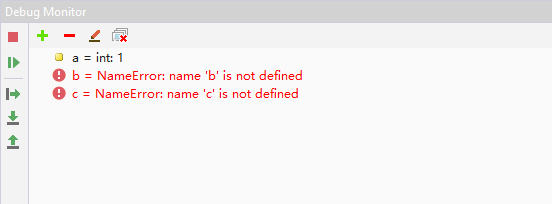
Place.



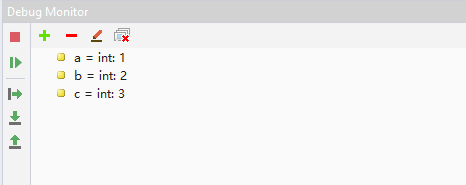
After the new variable is created, the debugger will immediately get the value of the current variable, but in this example, the "**NameError**" error is prompted at this time, because the code has not yet run the assignment operation to a, b, and c. Here continue to run the code to the second breakpoint.



In the Debug Monitor, you can see that a has the correct value.

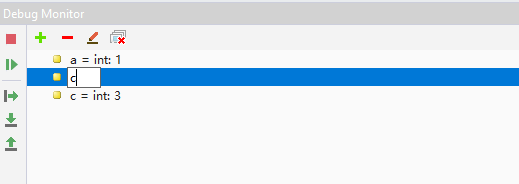


Continue to run the code to the next breakpoint, a, b, c complete the assignment operation, Debug Monitor can see the value of a, b, c.

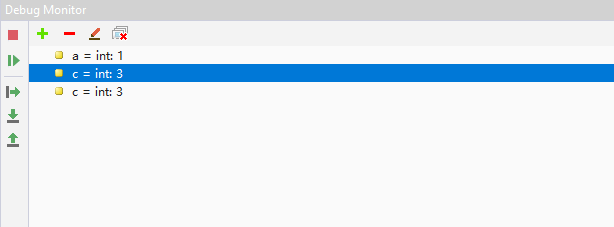


**Modify variables**

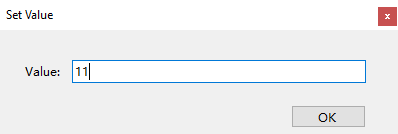
The debugger allows the user to modify the name and value of the monitored variable during the code execution. Following the example above, click the right-click menu "Edit Name", and in the editing state, change the name of b to c.



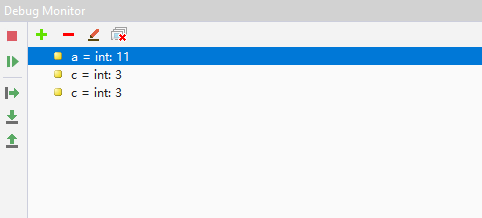
After the modification is completed, double-click the blank area to make the modification take effect. After it takes effect, the debugger will immediately obtain the value of the new variable.



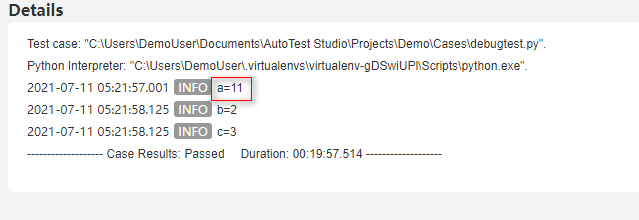
In addition to modifying the variable name, the debugging engine also allows the user to modify the value of the variable, select the a variable, click the right-click menu "Set Value", and set the value of the a variable to "11".



After clicking **OK**, Debug Monitor shows that the value of a has become "**11**".



In order to verify whether the modified variable value takes effect, we run the code to the end and check the log print.



As you can see from the log, the code is already at the modified value when it reaches "LogInfo("a={0}".format(a))".

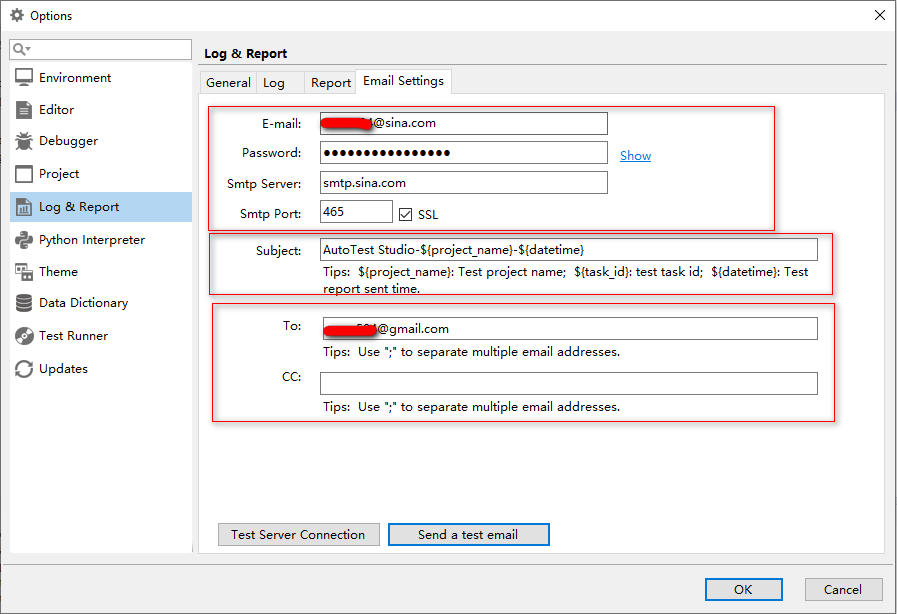
Code debugging is a very important skill in the development process. This article only briefly introduces the basic use of AutoTest Studio's debugging functions. You can explore more complex skills during use.

# AutoTest Studio Getting started tutorials 9: Send test report via email

AutoTest Studio has a built-in mail sending module, which can automatically send the test report of the task to the specified mailbox after the task runs.

**Configure email parameters**

The configuration is divided into three parts, as shown in the figure below, from top to bottom, the configuration of the sender is in turn, the middle part is the header format of the email body, and the lower part is the recipient's email address. Multiple recipient addresses can be used.



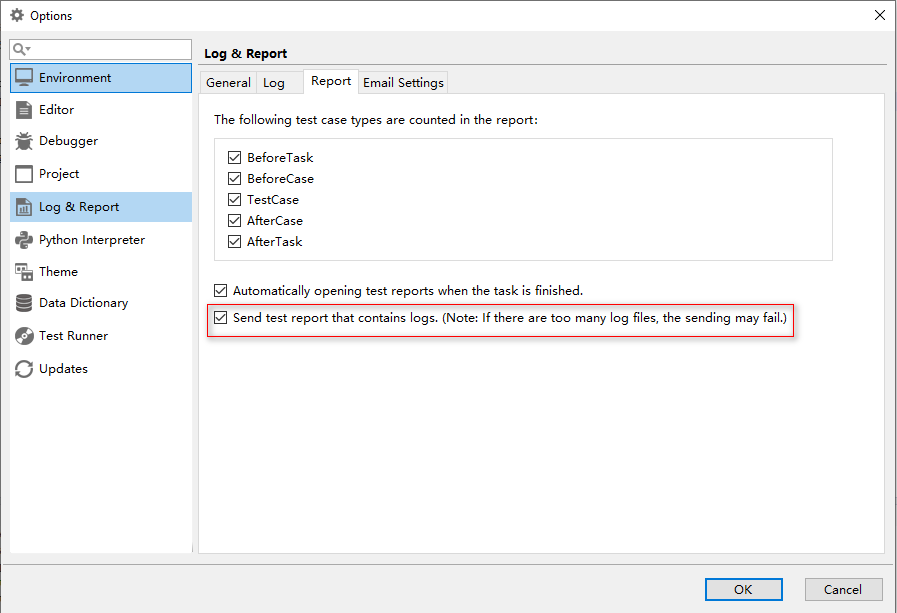
**Test configuration**

After we have completed the parameter configuration of the mailbox, we can click "Send a test email" to check whether the configuration information is correct. As shown in the figure above, we open the received mailbox and receive a subject with the subject "AutoTest Studio Mail Test" s mail.



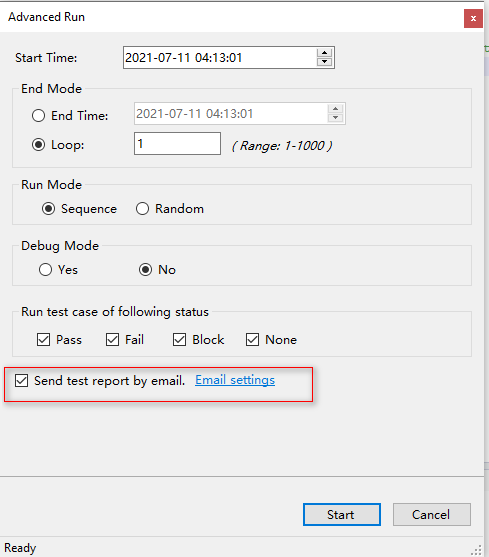
**Carry log**

The report sent by AutoTest Studio does not carry log files by default, because log files are often relatively large, which may cause transmission failure. If your mailbox supports larger attachments, you can set the email to carry log files. The settings are as follows.

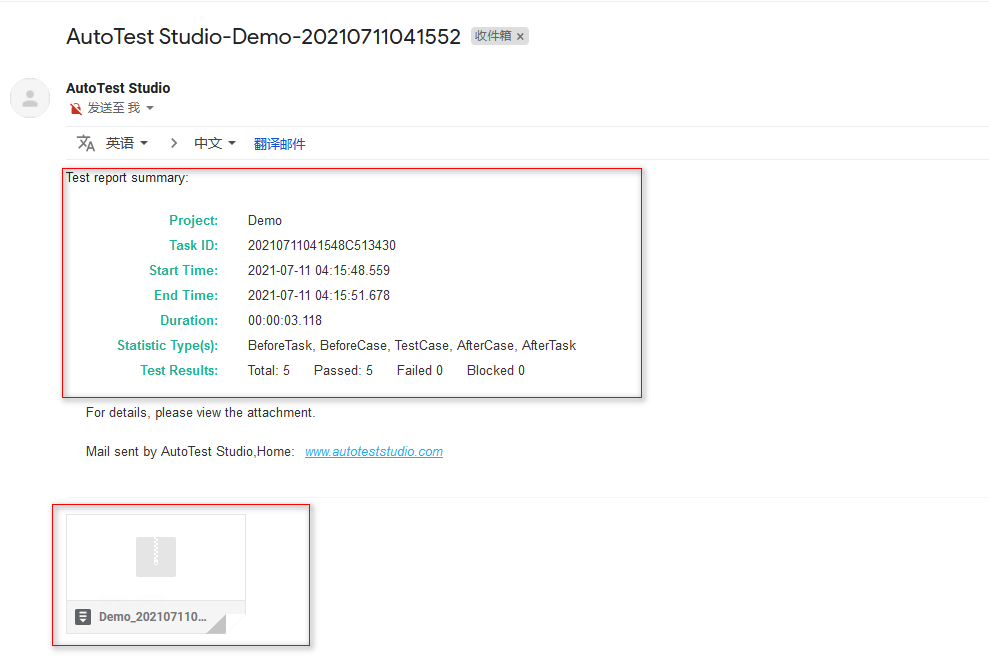


**Send a report after the task is over**

The function of sending test report can only be used when it is running in "Advanced Run" mode. As shown in the figure below, check "Send test report by email" and click "Start" to run the task.



After the task is completed, we check the receiving mailbox and we can see that we have received an email with the subject "AutoTest Studio-Demo-20210711041552". The attachment is the compressed package of the test report and log. The body of the email is as follows.



# AutoTest Studio Getting started tutorials 10: Jenkins integrates AutoTest Studio

With the prevalence of agile development, many projects have their own CI environment. In this section, we will introduce how to integrate AutoTest Studio in Jenkins. Before reading this section, it is assumed that you already have a certain understanding of Jenkins, and the content of this article is only around the integration of Jenkins and AutoTest Studio, and will not describe some basic operations too much.

Jdk and Jenkins installation package:

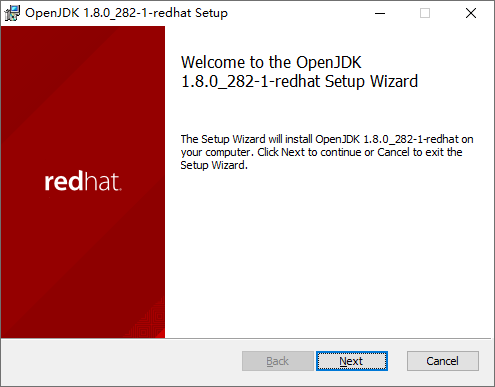
java-1.8.0-openjdk-1.8.0.282-1.b08.dev.redhat.windows.x86\_64.msi

Jenkins.war

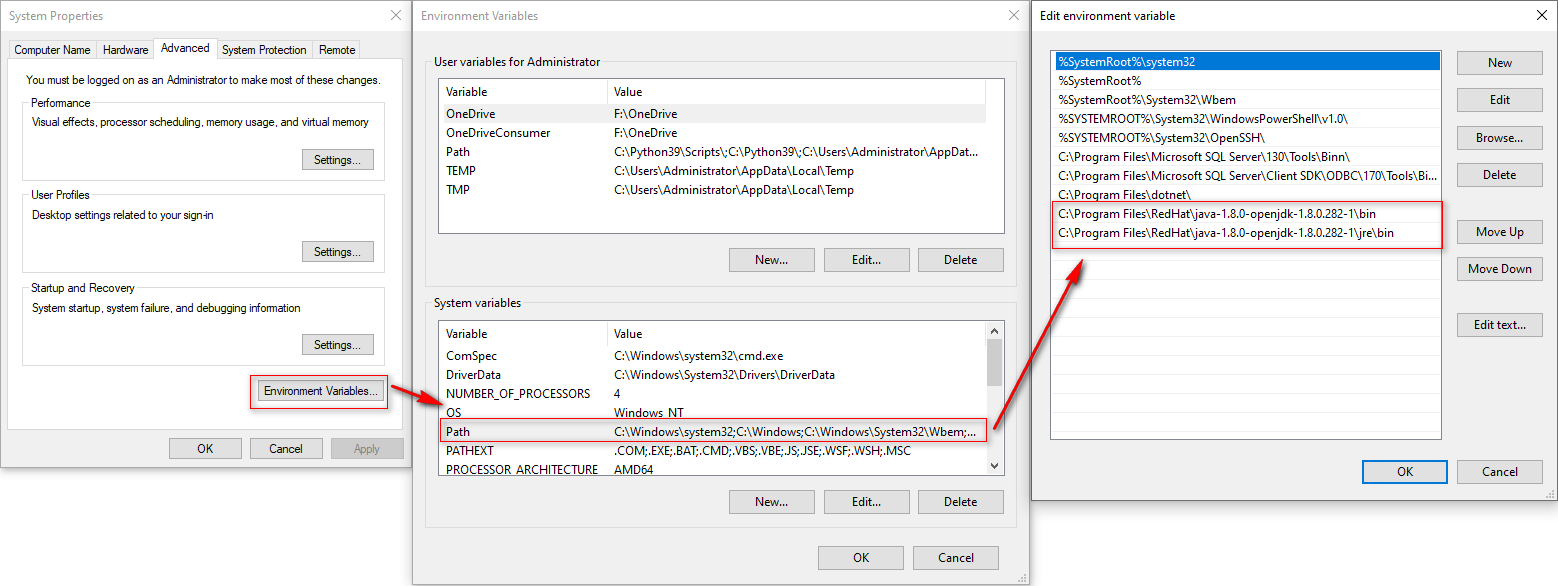
Note: jdk can use other jdk versions, but in the demo of this article, Jenkins must be started by the command line using the war package, because AutoTest Studio is a desktop application and needs to be run in a desktop environment. If Jenkins is installed as a service, it will not be able to Run AutoTest Studio normally.

**Install jdk**

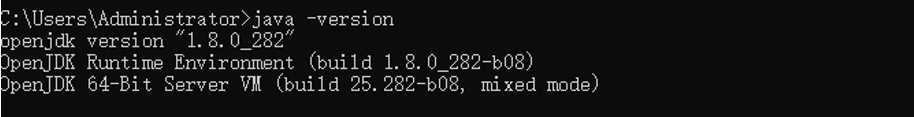
The installation of Jdk is very simple, just click the "Next" button all the time.



After the installation is complete, check the system environment variables. The jdk installation package used in this article will automatically configure the environment variables. If the environment variables are not configured for other jdk versions, you need to configure them manually.

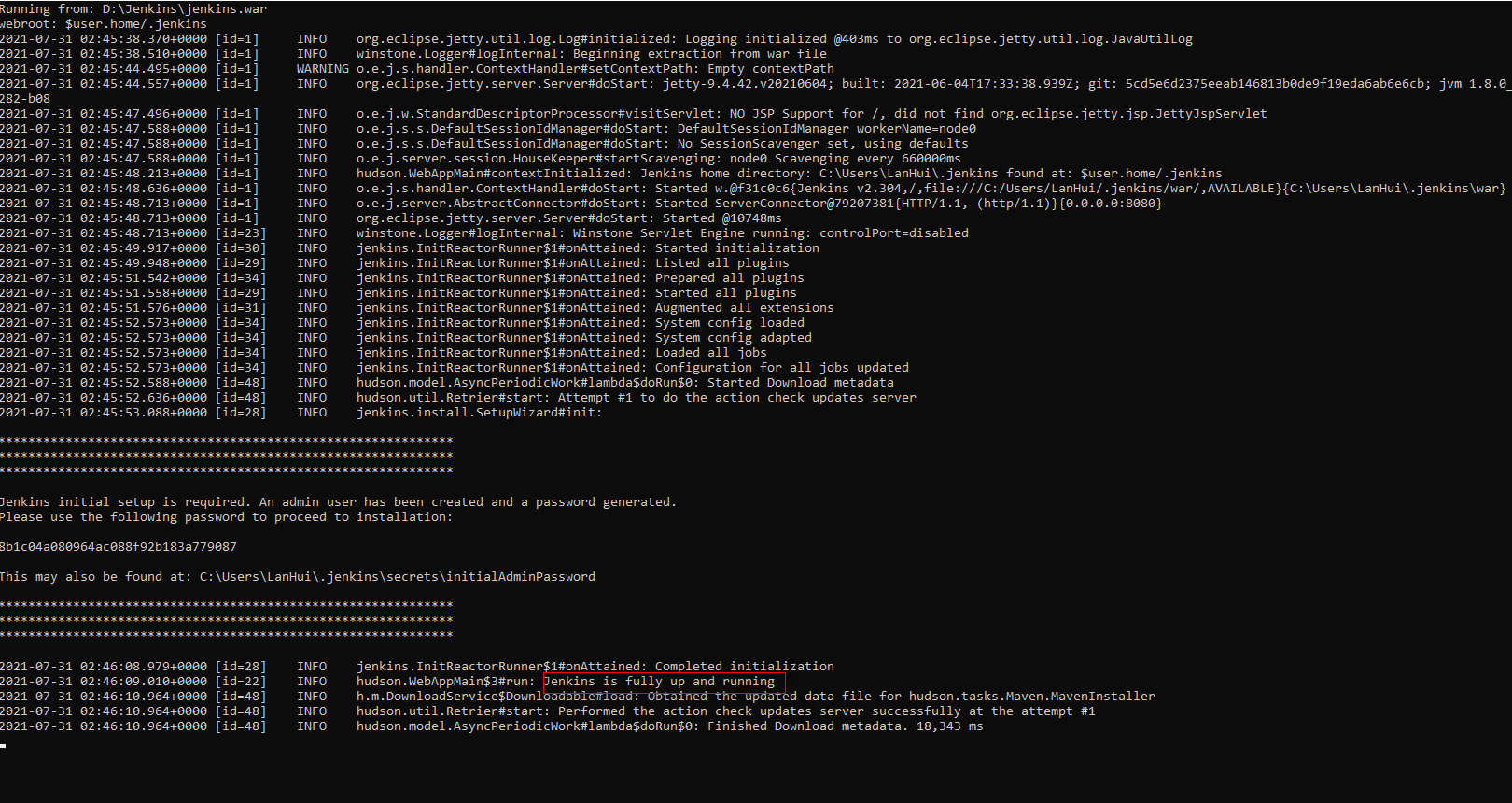


Running "**java -version**" in cmd returns the following version information, indicating that the jdk is installed normally and the environment variables have been configured.

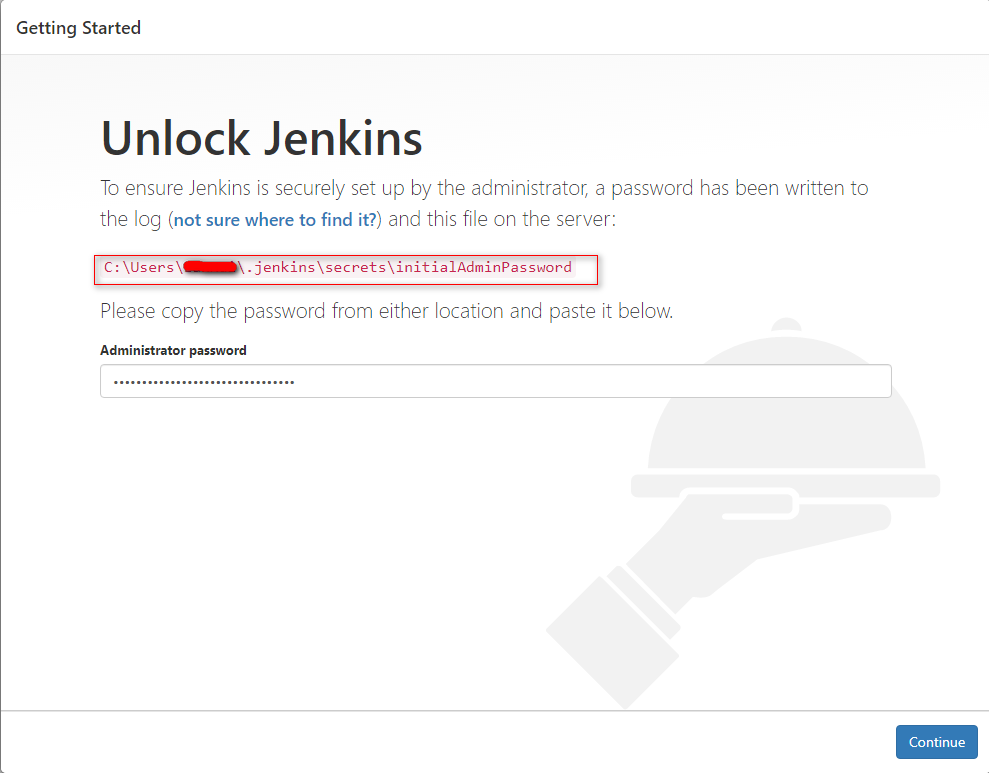


**Start Jenkins**

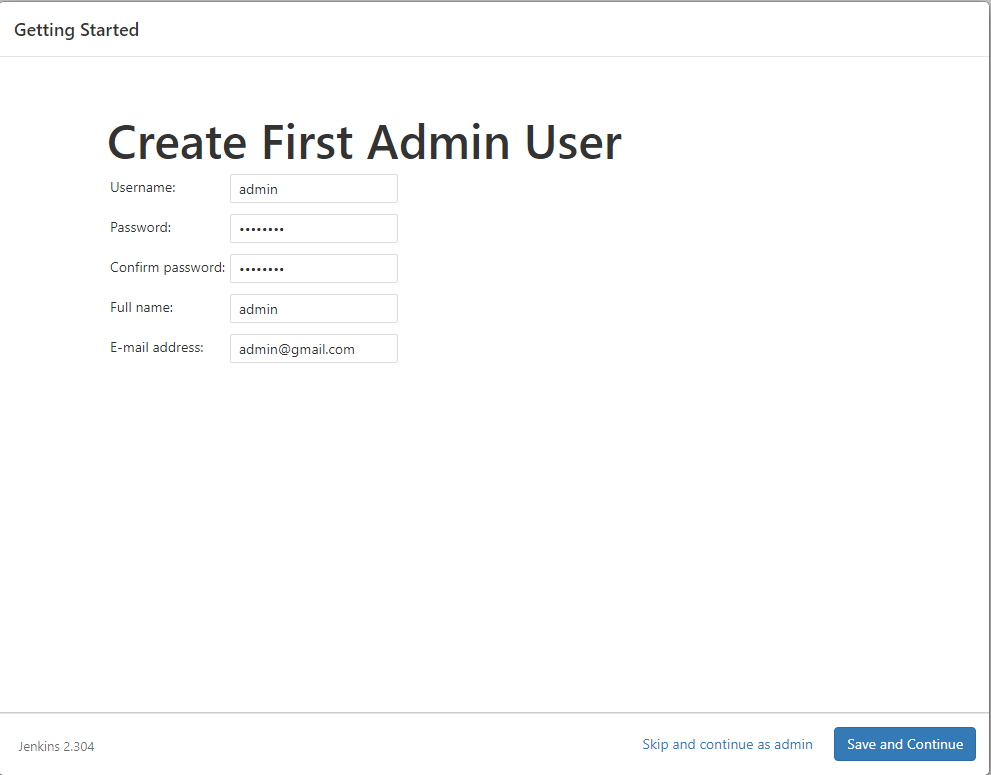
Start Jenkins through the command line, start the command "*java -jar "D:\Jenkins\Jenkins.war*"", when "**Jenkins is fully up and running**." appears in the startup message, it means that Jenkins has been started normally.



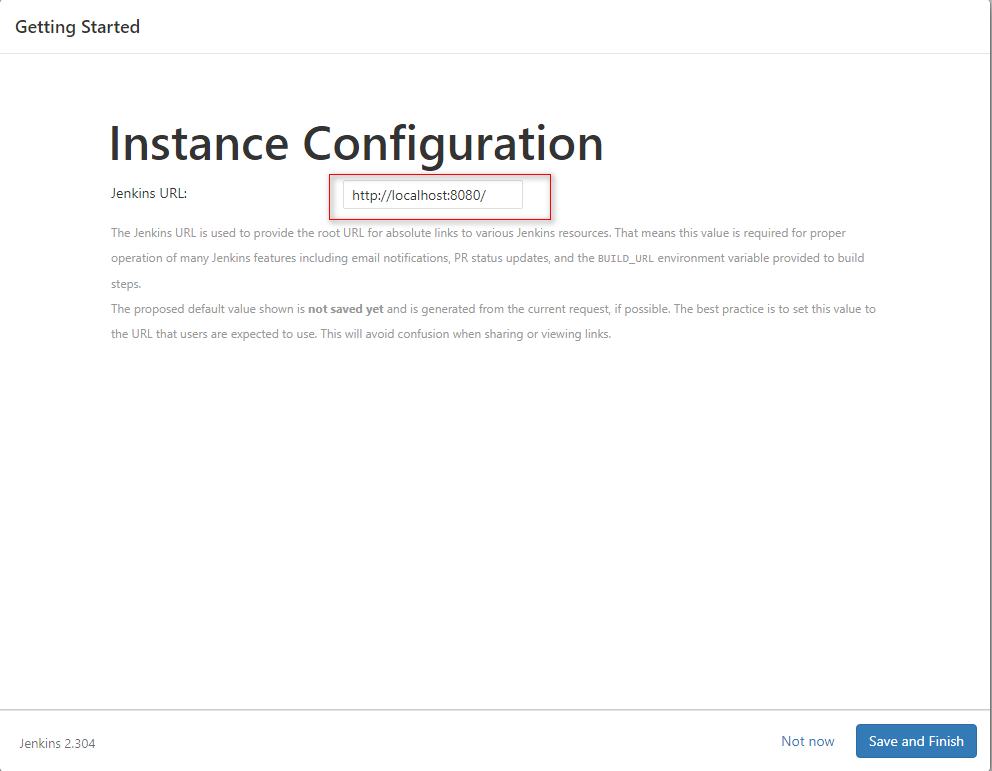
Use a browser to log in to "http://localhost:8080". Administart password is in the prompt file, as shown in the figure below.



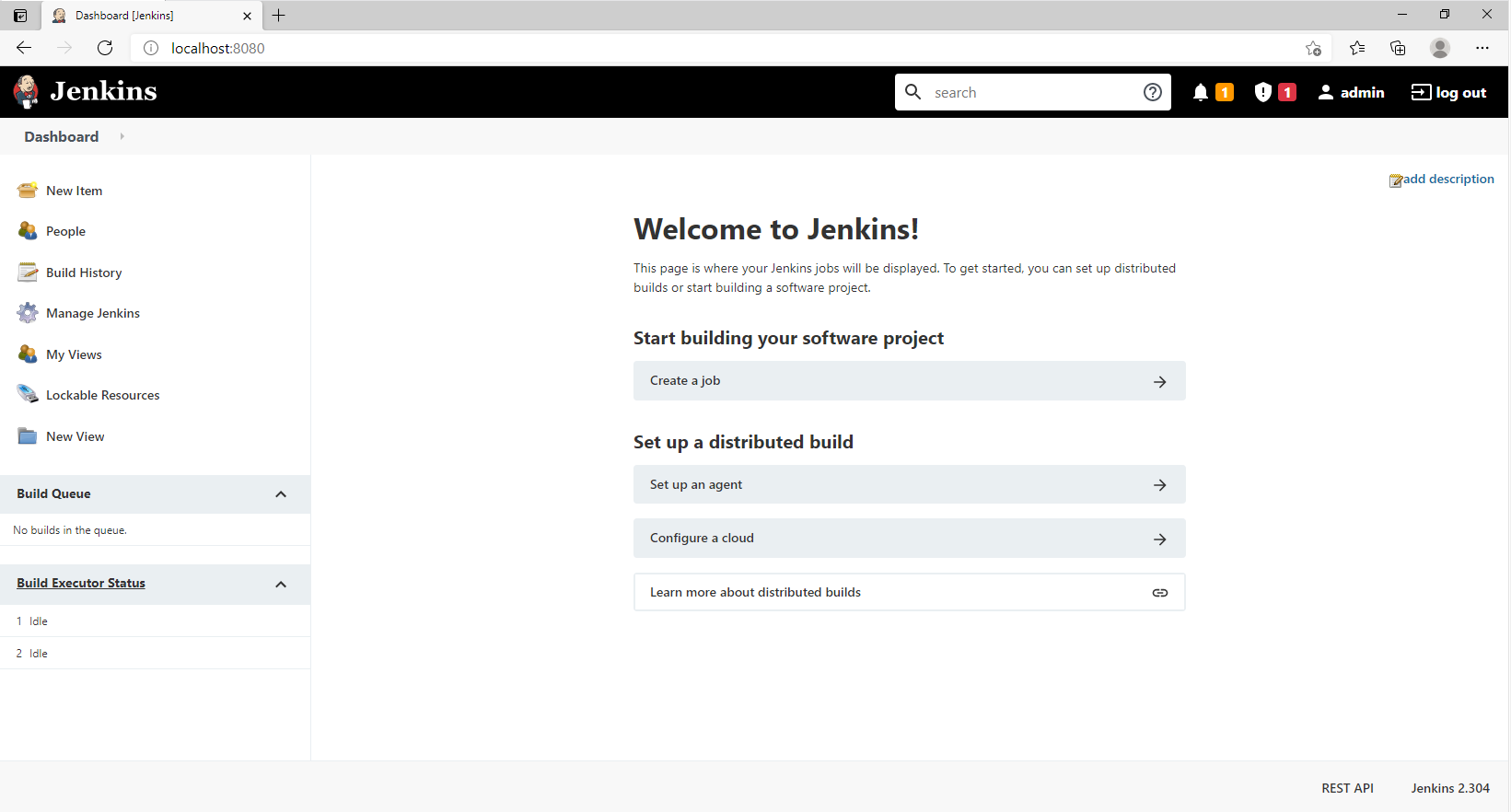
When using Jenkins for the first time, you need to install the default plug-in, just follow the prompts to install it. After the installation is complete, you need to create a user.



After the user is created, Jenkins will configure the login ip address and port. Here we use the default URL, which is "http://localhost:8080/";

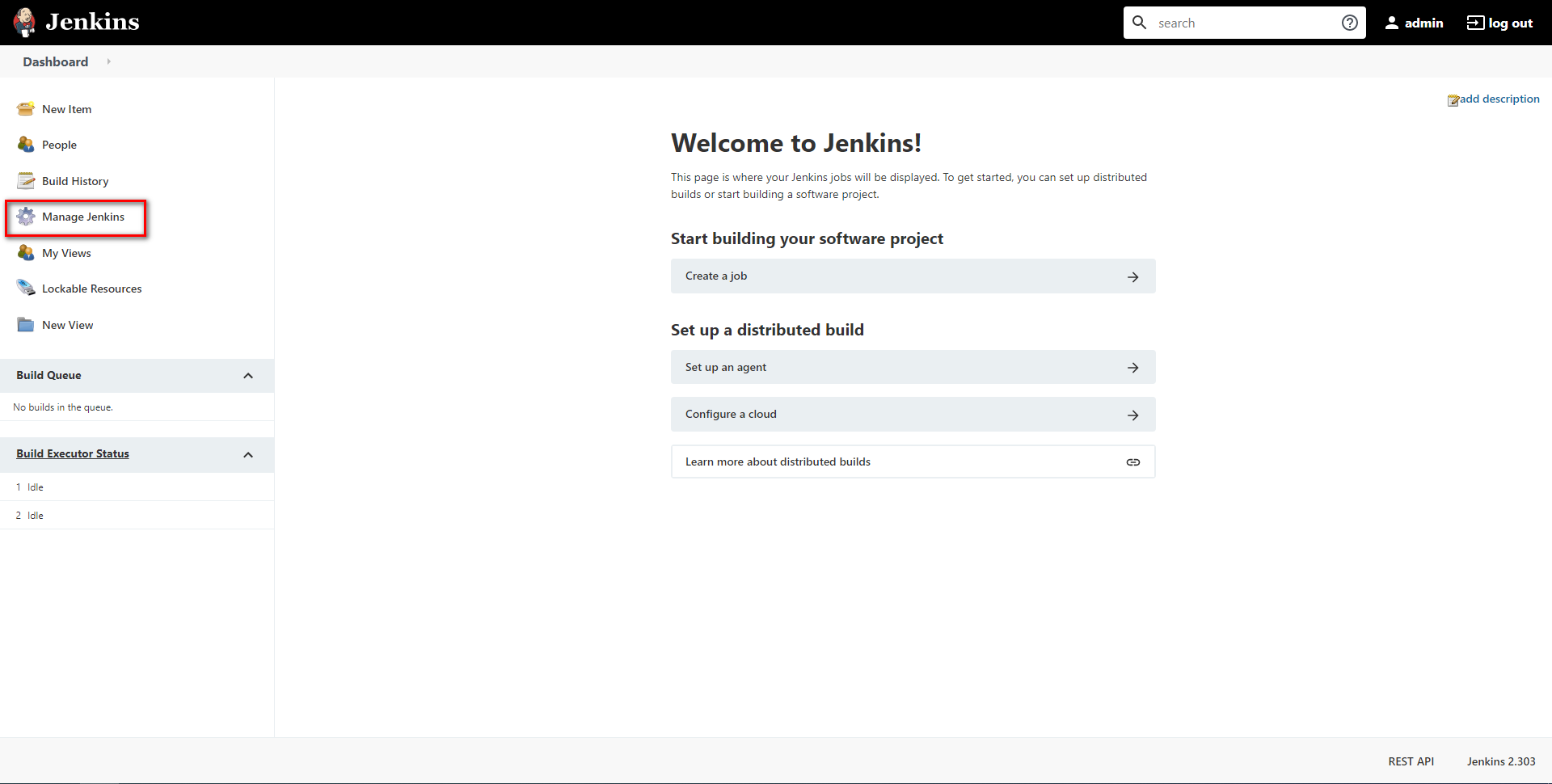


After the basic configuration of Jenkins is completed, enter the Dashboard interface, as shown in the following figure.

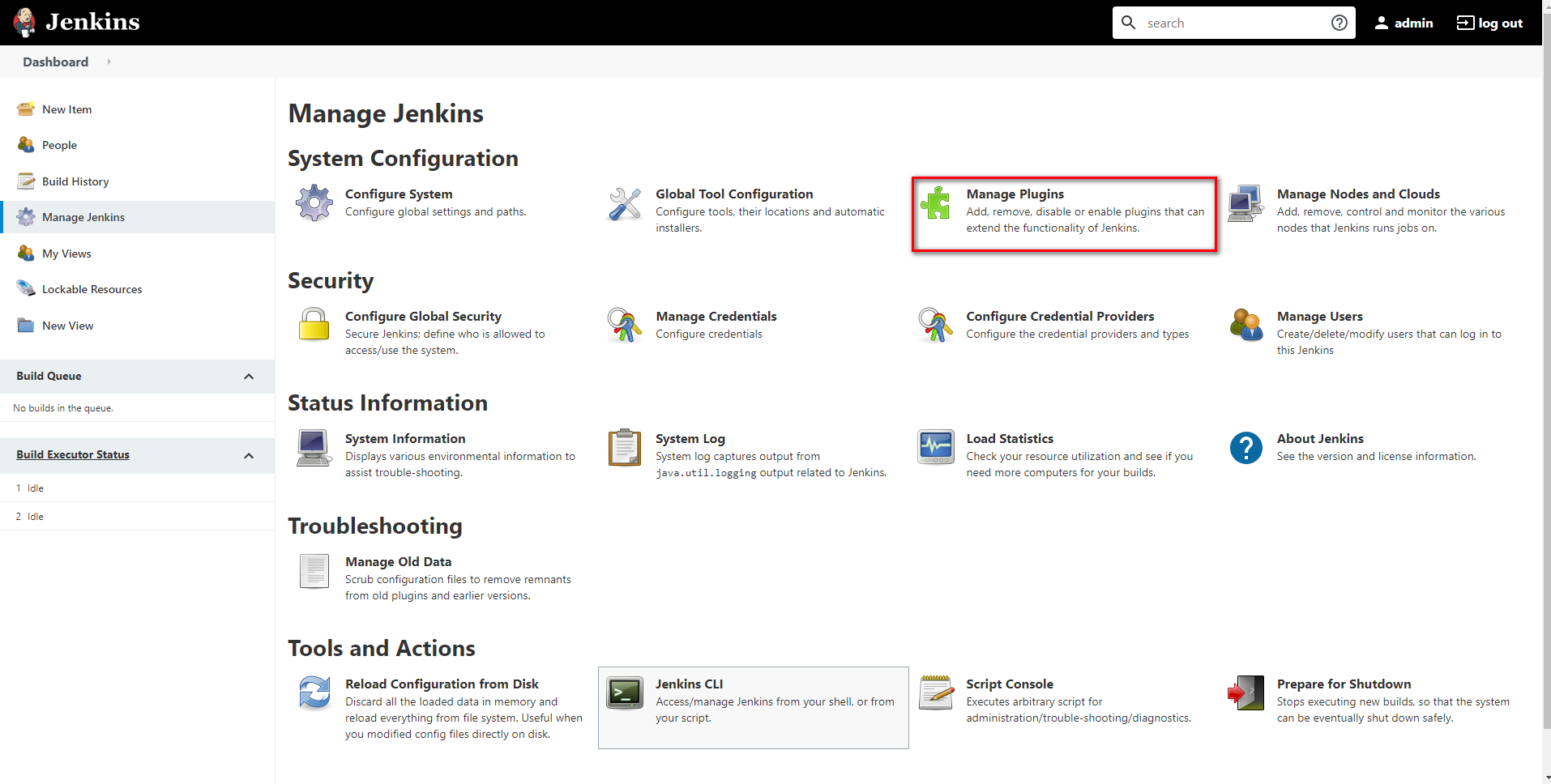


**Install HTML Publisher plug-in**

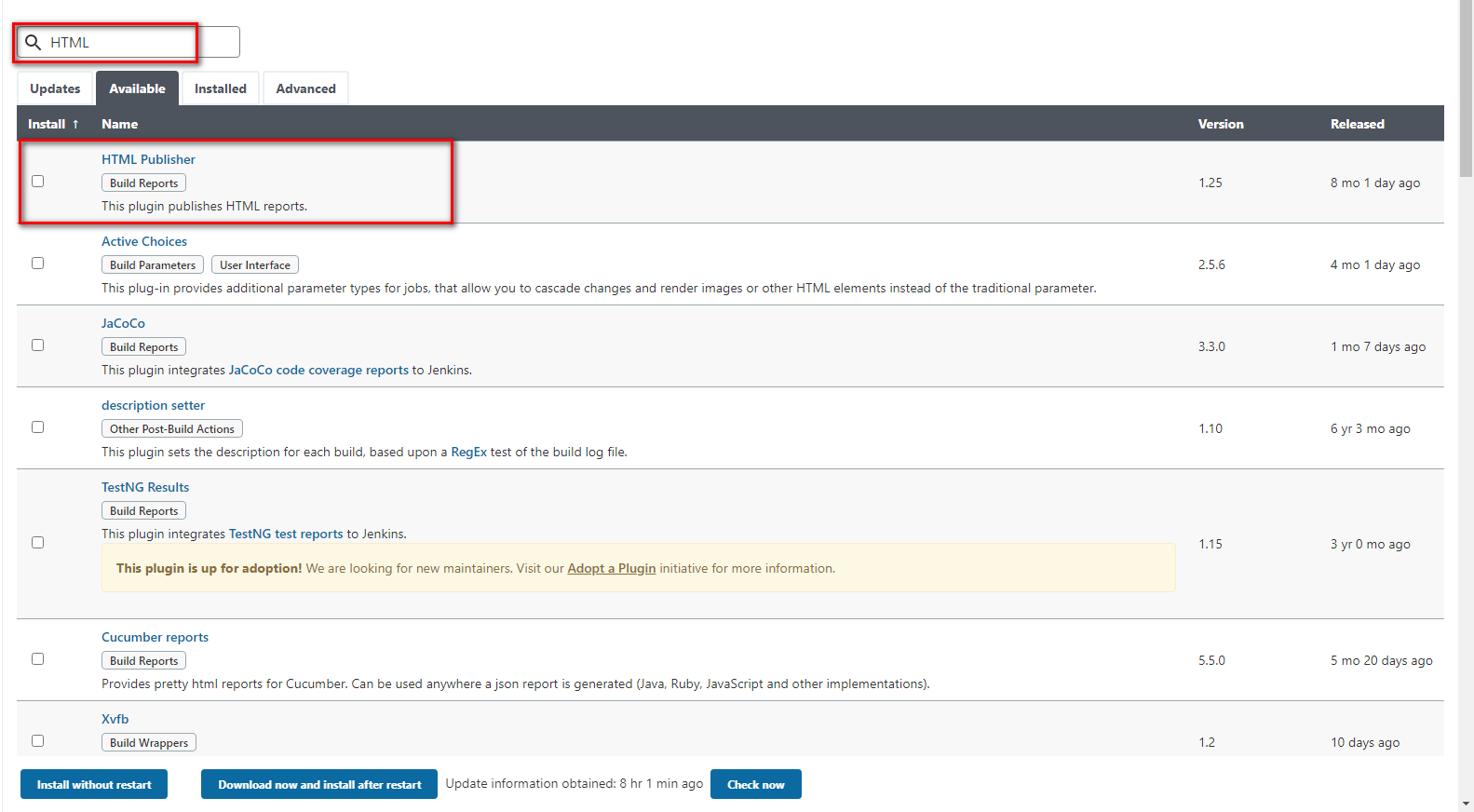
AutoTest Studio uses the "**HTML Publisher**" plug-in to publish test reports in Jenkins, so we need to install the HTML Publisher plug-in. In the Dashboard interface of Jenkins, click "**Manage Jenkins**" -> "**Manage Plugins**" to enter the plug-in management module.



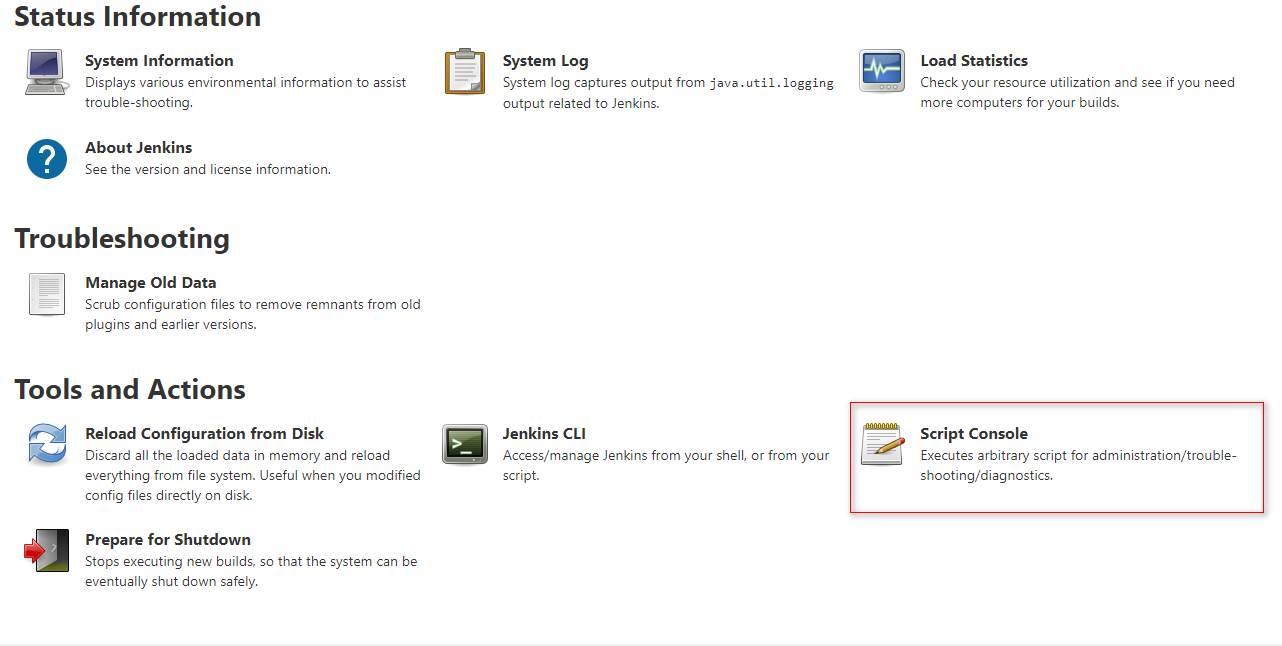
Select "**Manage Plugins**"



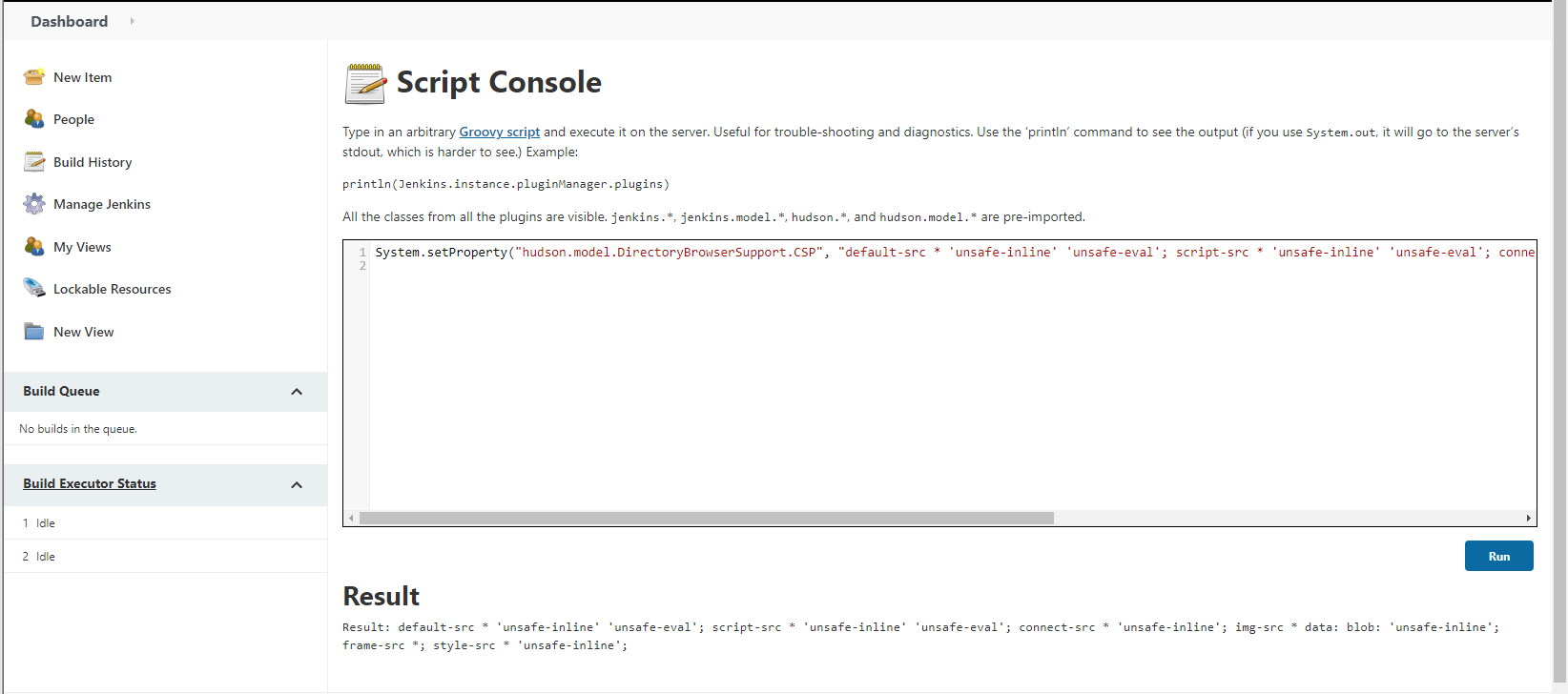
After outputting "**HTML**" in the search bar, select the "**HTML Publisher**" plugin in the list of search results, and click the "**Install without restart**" button at the bottom.



After installing the HTML Publisher plug-in, because Jenkins prohibits running js and css in Html files for security reasons, we also need to modify the CSP security mechanism in Jenkins, and click "**Manage Jenkins**" -> "**Script console**" in the Dashboard. , Enter "**Script console**".

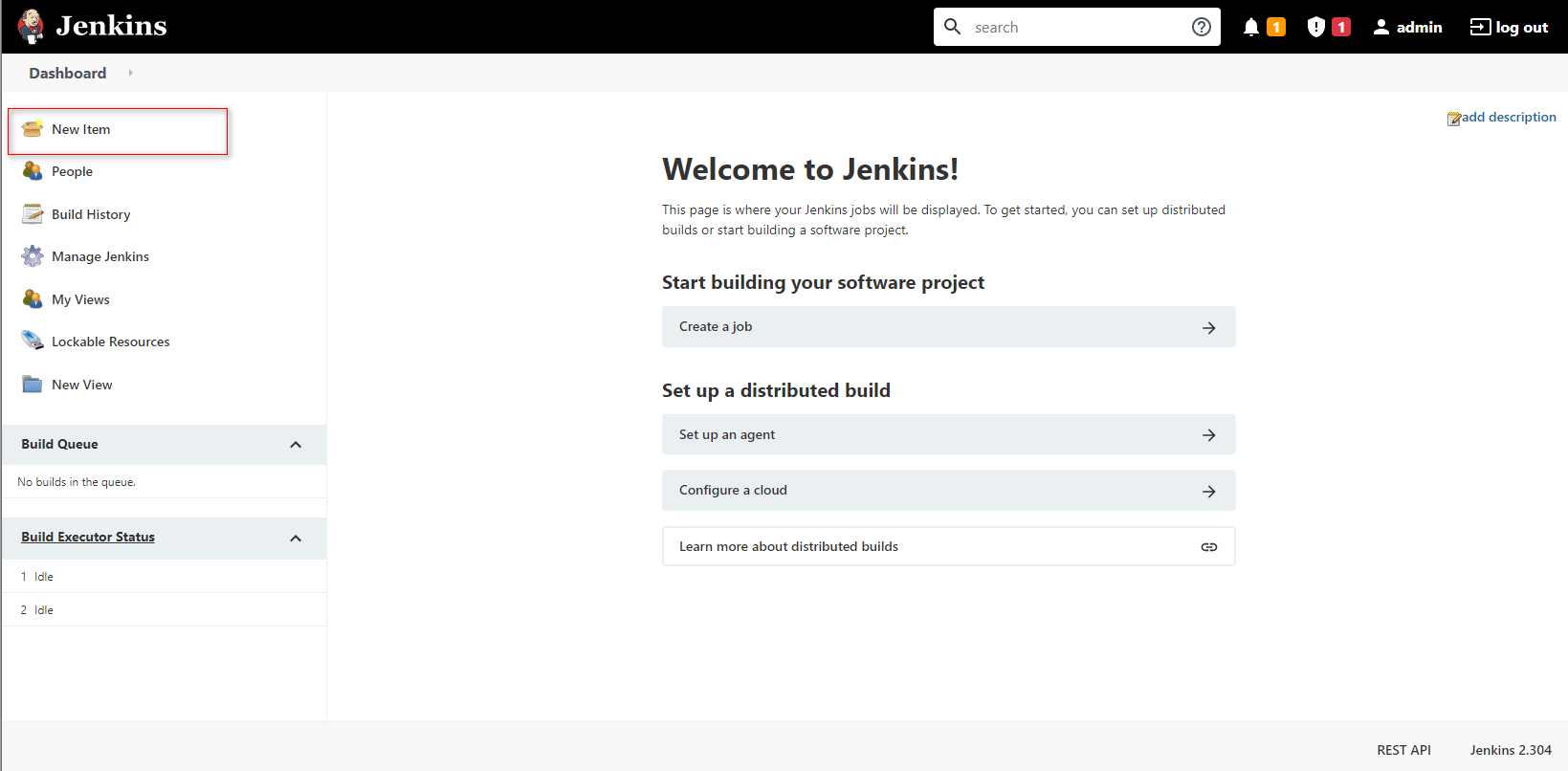


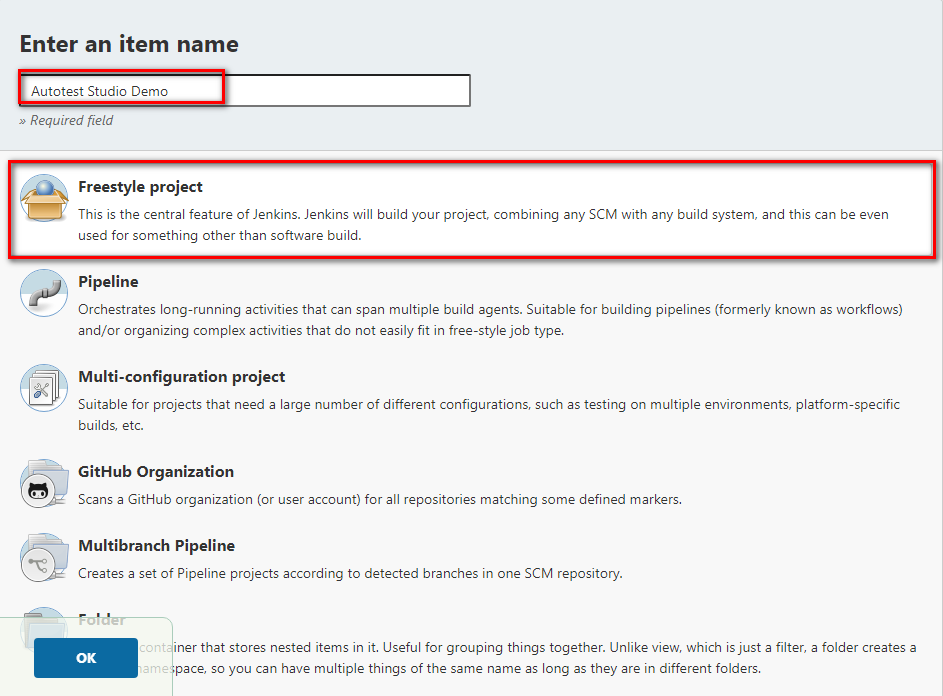
Run the code *System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "default-src \*'unsafe-inline''unsafe-eval'; script-src \*'unsafe-inline''unsafe-eval'; connect-src \*'unsafe-inline'; img-src \* data: blob:'unsafe-inline'; frame-src \*; style-src \*'unsafe-inline';")*, the running result is shown in the figure below.

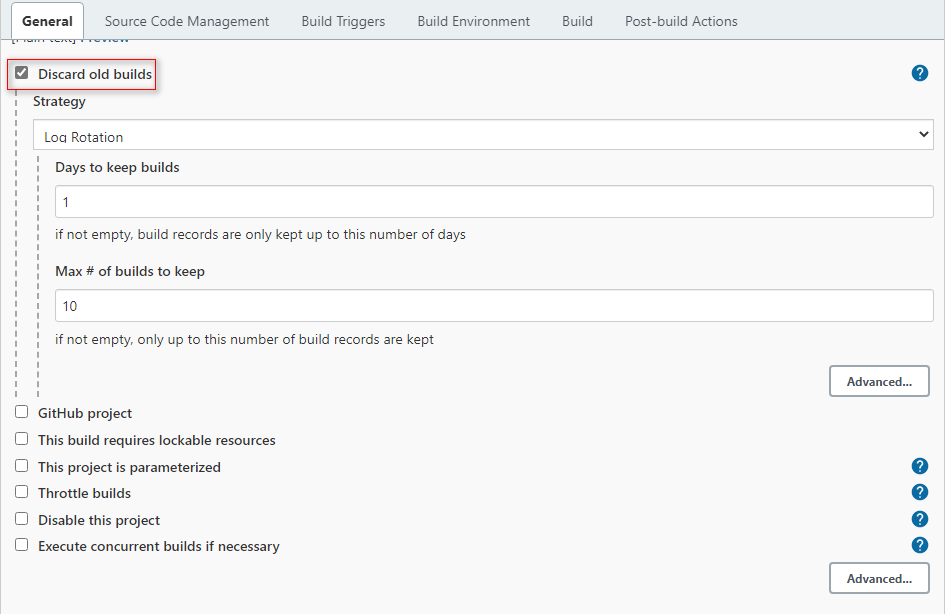


**Create a new item**

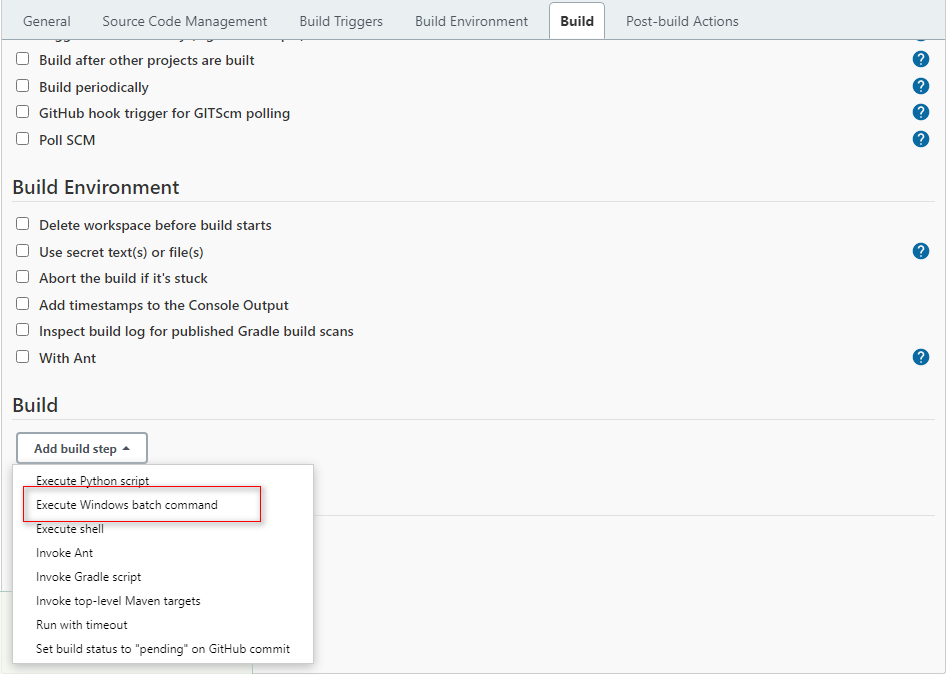
Create a new Item with the name "AutoTest Stdio Demo" and the style "**Freestyle project**".







Select "**Execute Windows batch command**" in Build to add AutoTest Studio call command.



Invoke the command "C:\Program Files\AutoTest Studio\AutoTest Studio.exe" "C:\Users\Demouser\Documents\AutoTest Studio\Projects\ats\_example\ats\_example.tpro" -r -q.

Command meaning:

**"C:\Program Files\AutoTest Studio\AutoTest Studio.exe"**, AutoTest Studio.exe path.

**"C:\Users\Demouser\Documents\AutoTest Studio\Projects\ats\_example\ats\_example.tpro"**, the project path, you need to select the script to be executed in the project before the task runs, and AutoTest Studio will automatically save it to the project file.

**-r**, run the task immediately after startup. Required parameters.

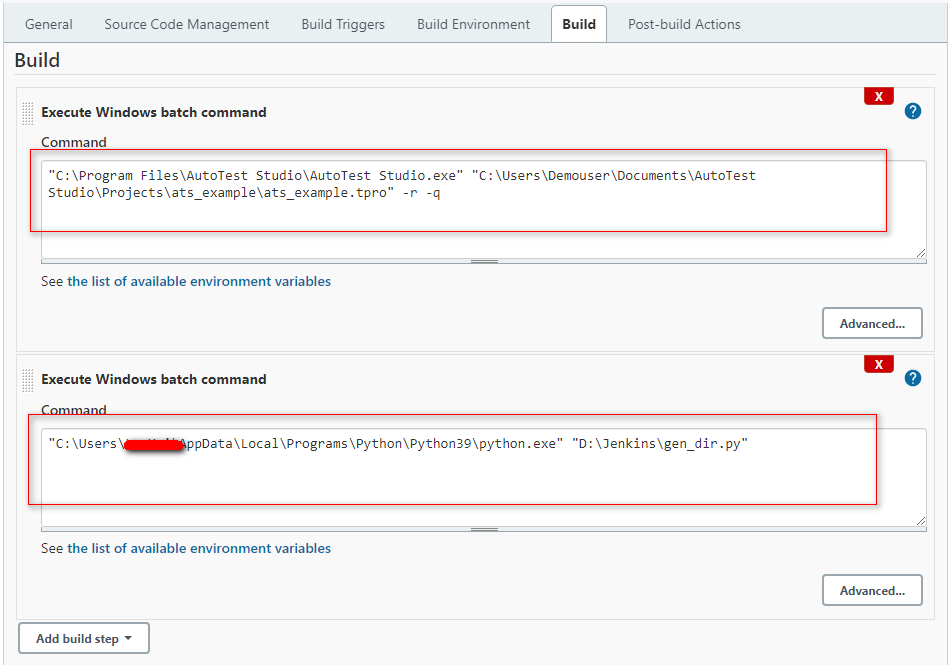
**-q**, after the task is completed, exit the AutoTest Studio program. Required parameters.

Since the report and log directories of AutoTest Studio are generated dynamically, we need to parse the log directory in the task output file "output.json" through a script and map it to the workspace.

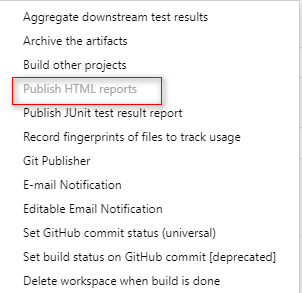
Save the following python code to a file, such as: "D:\Jenkins\jen\_dir.py", but there is no restriction on the location of the script.

import json  
import os  
  
  
file=".\\output.json"  
target\_report\_dir="report\_logs"  
if not os.path.exists(file):  
 print("\"output.json\" does not exist.")  
 exit(-1)  
  
fd=open(file,"r")  
log\_json=json.loads(fd.read())  
fd.close()  
log\_dir=log\_json["LogDirectory"]  
  
cmd='rmdir /s /q "{0}" & mklink /j "{0}" "{1}"'.format(target\_report\_dir,log\_dir)  
os.system(cmd)

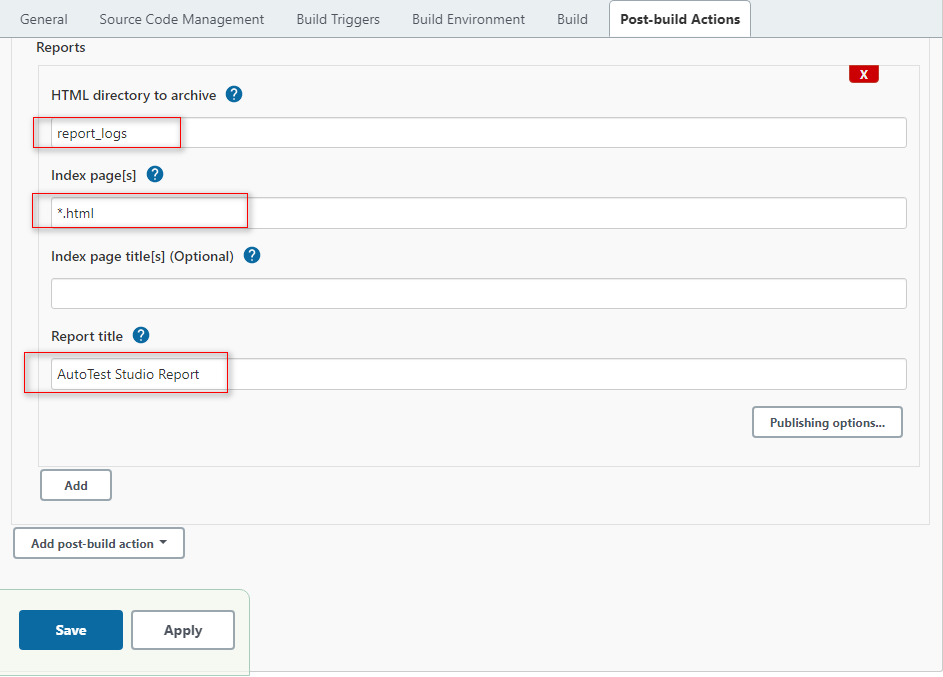
These two build commands are shown in the figure below.



In "**Post-build Actions**", use the "**Publish HTML reposts**" plug-in to send AutoTest Studio test reports to Jenkins.



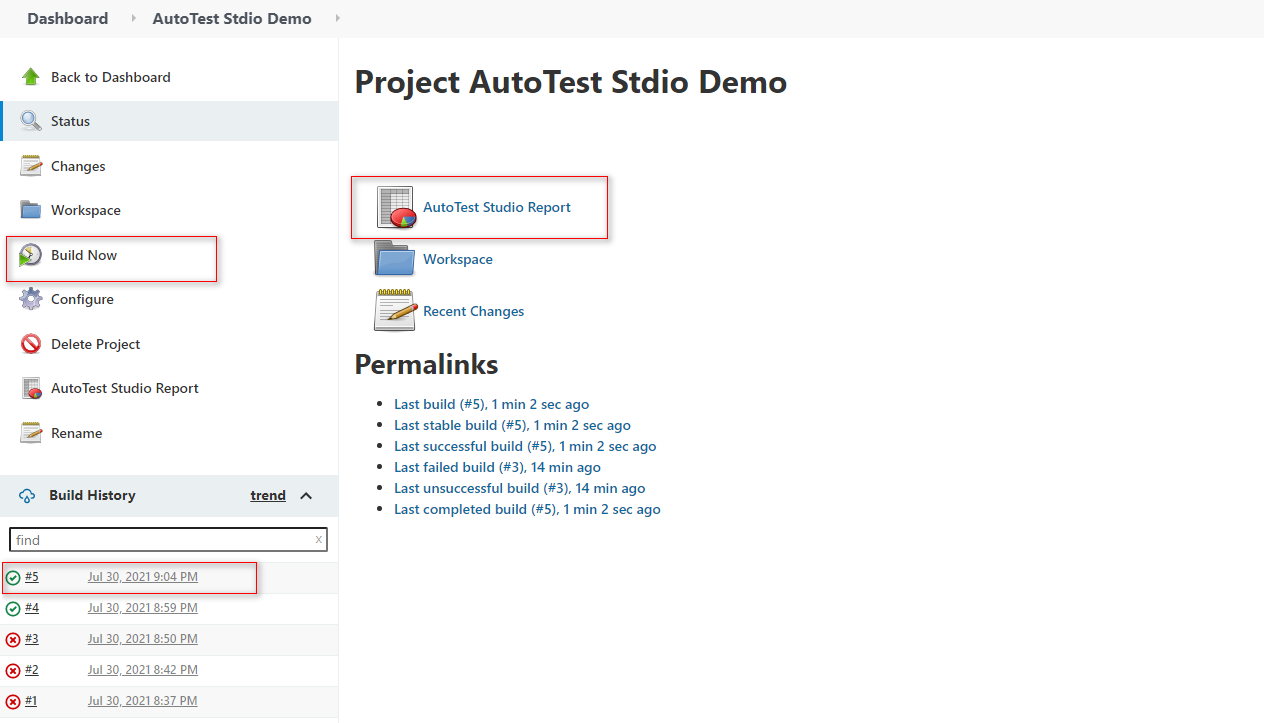
The "**Publish HTML reposts**" plug-in configuration parameters are as follows. Note that the directory "**report\_logs**" must be consistent with the commands in the above script.



After the configuration is complete, click the "Save" button at the bottom to save the configuration.

**Perform build tasks**

Click "Bulid New" in the Dashboard to perform a build operation.



After the build operation is completed, we click "**AutoTest Studio Report**" to open the test report, and you can see the content as shown in the figure below. This page is captured by the "Publish HTML report" plugin. Click "**Zip**" in the upper right corner of the page. Download the test report and log of this task run.

