SAS® Infrastructure for Risk Management 3.6: Programmer's Guide for Python

SAS® Documentation
April 21, 2022

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2019. *SAS® Infrastructure for Risk Management 3.6: Programmer's Guide for Python*. Cary, NC: SAS Institute Inc.

SAS® Infrastructure for Risk Management 3.6: Programmer's Guide for Python

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

April 2022

3.6-P1:irmpg3python

Contents

About This Book

Audience

This book is intended for Python programmers who want to learn how to easily create parallel programs that run on the SAS Infrastructure for Risk Management platform.

What's New in SAS Infrastructure for Risk Management 3.6

New Features and Enhancements

SAS Infrastructure for Risk Management 3.6 provides the following new Python programming features and enhancements:

- The SAS Infrastructure for Risk Management Python scripting client. The Python scripting client is a Python package that simplifies the process of creating job flows (parallel programs).
- Support for Python tasks.
- A Python sample federated area (fa.sample.3.6.py) that contains Python sample content that you can use to create Python parallel programs and to use as a reference.
- Support for automatic conversion of SAS data sets to and from Pandas Data Frames.
- Configuration sets and entities that are defined in CSV files.

Chapter 1

Before You Begin

About SAS Infrastructure for Risk Management

SAS Infrastructure for Risk Management is a high-performance job execution engine that leverages the many-task computing model. SAS Infrastructure for Risk Management enables Python programmers to quickly and easily develop highly parallel programs.

Use SAS Infrastructure for Risk Management to create Python programs that function as follows:

- execute computationally intensive analytics as fast as possible
- use any Python integrated development environment (IDE) such as PyCharm
- automate the handling of the complexity of creating parallel programming, task scheduling, and job distribution
- automatically generate documentation

Key Concepts

As a programmer who uses SAS Infrastructure for Risk Management to create parallel programs, you must understand the following SAS Infrastructure for Risk Management key concepts and components:

task
: Tasks are pre-constructed analytical or reporting programs. Inputs and outputs are defined for these programs, and each program performs an independent unit of work. Tasks are the basic building blocks of SAS Infrastructure for Risk Management.

job flow (parallel program)
: A *job flow*, or parallel program, is the method that SAS Infrastructure for Risk Management uses to organize the tasks that make up a job.

  A *job flow definition* specifies the tasks and subflows that are contained in the job flow.

  A *job flow instance* is a single occurrence of a job flow definition.

  In this guide, the term, job flow, might be used to refer to both a job flow definition and a job flow instance. The exact meaning depends on the context in which it is used.

subflow
: A subflow is a job flow definition that is contained within another job flow definition.

federated area
: A federated area is a folder structure that conforms to specific SAS Infrastructure for Risk Management rules. It is a storage area for content and it is the mechanism by which programmers add custom content to SAS Infrastructure for Risk Management. SAS Infrastructure for Risk Management provides a platform federated area and a sample federated area. In addition, programmers have their own personal federated areas, which are writable only to them. A personal federated area is the location in which a single programmer develops custom content.

content
: Content refers to any file that can be delivered in SAS Infrastructure for Risk Management, but is not necessarily part of the SAS Infrastructure for Risk Management installation. Examples of content include data, code, and configuration information. Because programmers deploy content for SAS Infrastructure for Risk Management in a federated area, federated areas and content are synonymous in SAS Infrastructure for Risk Management.

data objects
: In SAS Infrastructure for Risk Management, all data is defined as a data object. A data object can be data input to one or more tasks, but it can be the output data for only one task. Data objects are immutable. A data object can model any data that can be contained in a file.

  Dynamically-computed data, like SAS views, are not valid data objects. This data must be modeled as a task.

Python scripting client
: The SAS Infrastructure for Risk Management Python scripting client is a Python package that is used to simplify the process of creating job flows (parallel programs).

Typical Development Scenario

The requirements to create parallel programs include the following:

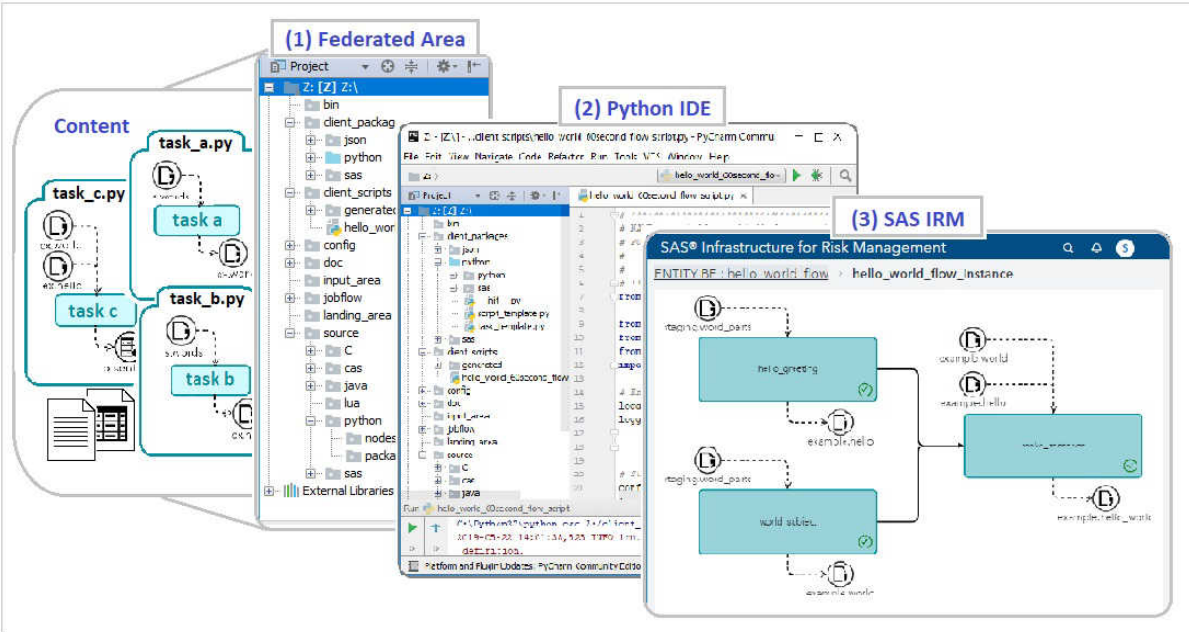
- You must have access to a SAS Infrastructure for Risk Management 3.6 (server and web application) and a Python IDE.
- Your development environment must be created on the SAS Infrastructure for Risk Management server. (See "Set Up Your Development Environment".)
- The Python scripting client must be installed and configured on your local client machine. (See "Install and Set Up the Python Scripting Client".)

Here is a typical development scenario for a SAS Infrastructure for Risk Management Python programmer:

1. Create your content (task files and related content) and save it to your personal federated area.
2. In your Python integrated development environment (IDE), write a program (job flow script) that uses the SAS Infrastructure for Risk Management Python scripting client package to create a job flow definition. In the job flow definition, you add tasks and you can execute the job flow definition to create a job flow instance.

   When you run the program, SAS Infrastructure for Risk Management automatically handles the complexity of creating the parallel program. SAS Infrastructure for Risk Management organizes the tasks into a job flow, schedules the tasks, and manages job distribution.
3. View the job flow instance in the SAS Infrastructure for Risk Management web application.

Here is a high-level visualization of the development scenario that was previously described. The detailed information in the visualization is intentionally blurred.



Set Up Your Development Environment

Before you can develop parallel programs in SAS Infrastructure for Risk Management or follow the development examples that are included in this guide, you must complete the following steps:

1. Ensure that Python and a Python IDE have been installed and configured on your local client system.

   Note: Python 3.6 or later is recommended for running the SAS Infrastructure for Risk Management 3.6 Python scripting client.

2. Ensure that you have access to the SAS Infrastructure for Risk Management 3.6 server and the SAS Infrastructure for Risk Management web application.

3. Consult with your system administrator about meeting the following requirements:

   - Enable development mode on the SAS Infrastructure for Risk Management server.
   - Create a developer's account for you.

     The configuration of your programmer's account enables files and folders that you create to be discovered by stored process servers and SAS workspace servers.

   For information about these administrative tasks, see "Configure the Development Environment" in *SAS Infrastructure for Risk Management: Administrator's Guide*.

4. Create your personal federated area.

   - Using your programmer's account, log on to the SAS Infrastructure for Risk Management web application.

     When you log on to the SAS Infrastructure for Risk Management web application for the first time, your personal federated area is automatically created in the following location:

     /*sas-configuration-directory*/Lev1/AppData/SASIRM/pa/fas

     By default, the personal federated area is named after your user name. For example, if your user name is "user1," the name of your personal federated area is as follows:

     /*sas-configuration-directory*/Lev1/AppData/SASIRM/pa/fas/fa.user1

     Note: The user name of your programmer's account cannot contain any special characters (\ / : * ? " < > |). If your user name contains a special character, your personal federated area will not be created when you log on to SAS Infrastructure for Risk Management for the first time.

   - After you create your personal federated area, contact your system administrator to obtain Write permissions to all folders in your personal federated area.

   For information about your personal federated area, see "Your Personal Federated Area ".

5. Install the SAS Infrastructure for Risk Management Python scripting client on your local machine. For details, see "Install and Set Up the Python Scripting Client".

Install and Set Up the Python Scripting Client

Overview

Before you can create parallel programs with the SAS Infrastructure for Risk Management Python scripting client, you must install and configure the Python scripting client on your local client machine. You can install the Python scripting client for any Python Integrated Development Environment (IDE).

This section explains how to install and use the Python scripting client on a local client on which Python 3.7 and the PyCharm IDE have been installed.

Note: To run the Python scripting client in PyCharm, the Python version where you install the scripting client and the version of the *Project Interpreter* that you use in the PyCharm IDE must be the same. In this guide, the scripting client is installed on a local client machine on which Python 3.7 and the PyCharm IDE using Project Interpreter for Python 3.7 are also installed.

To install and set up the SAS Infrastructure for Risk Management Python scripting client:

1. Download the SAS Infrastructure for Risk Management wheel file.
2. Install the downloaded wheel file.
3. Locate the installed Python scripting client wheel file package.
4. Initialize SAS Infrastructure for Risk Management on your local client machine.
5. Mount your personal federated area on the SAS Infrastructure for Risk Management server to your local client machine.
6. Open Pycharm.

Download the SAS Infrastructure for Risk Management Wheel Package

1. In a web browser, log on to the SAS Infrastructure for Risk Management WebDAV server and navigate to the platform **`client_packages`** folder:

   http://*middle-tier-host:port*/SASIRMServer/irmwebdav/*user-name*/fas/0.3.6/client_packages/

   Here is the URL for a user (sasdemo) on the SAS Infrastructure for Risk Management server (myIrmHost) at port 7980:

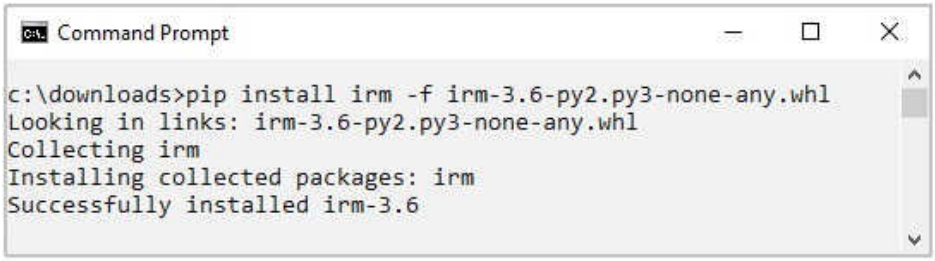
   http://myIrmHost:7980/SASIRMServer/irmwebdav/sasdemo/fas/0.3.6/client_packages/

2. Download the SAS Infrastructure for Risk Management wheel package (**`irm-3.6-py2.py3-none-any.whl`**) to your local machine.

Note: In this example, the package was saved to **`c:\downloads`** on the local machine by right-clicking the package name and selecting **Save link as** on the pop-up menu.

Install the Downloaded Wheel File

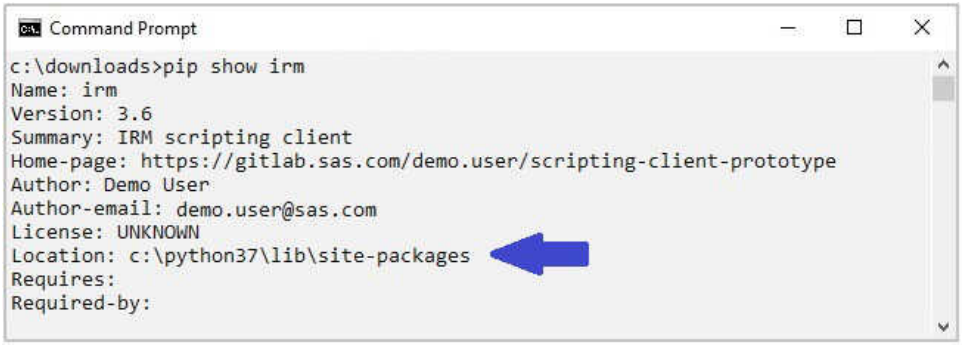
To install the wheel file on your local machine, enter the **`pip install irm -f irm-3.6-py2.py3-none-any.whl`** command:



Locate the Wheel File for the Installed Python Scripting Client Package

The location of the wheel file for the installed Python scripting client package is where you run **`irmdev`** commands.

To locate the SAS Infrastructure for Risk Management Python scripting client package that was installed from the downloaded wheel file, enter the **`pip show irm`** command:



If the wheel package was installed correctly, the location of the Python scripting client package is displayed in the command output. In this example, the package was installed correctly and is located in **`c:\python37\lib\site-packages`**.

Where a package is installed varies according to the location of the Python and pip installation on your local machine. For example, if your local machine is running Windows and Python has been installed in a default location, the package might be located in the following directory:

`c:\users\username\appdata\local\programs\python\python36\scripts`

If your local machine is a Mac and Python has been installed in a default location, the package might be located in the **`/Users/username/.virtualenvs/SC-TEST/lib/python3.6/site-packages`** directory.

The **`irmdev`** commands and the **`irmdev`** script are located in **`/Users/username/.virtualenvs/SC-TEST/bin`**.

Initialize SAS Infrastructure for Risk Management on Your Local Machine

To initialize SAS Infrastructure for Risk Management on your local machine, enter the **`irmdev init`** command in the directory in which the Python scripting client was installed and answer the prompts.

Note: For information about how to locate where your Python scripting client was installed, see "Locate the Wheel File for the Installed Python Scripting Client Package".

```
Command Prompt
c:\Python37\Scripts>python irmdev init
IRM server URL [Hit enter for default "localhost"]: myIrmHost.unx.sas.com
IRM server port [Hit enter for default "7980"]:
Use https? [Y/y/N/n or Hit enter for default "N"]:
User name [Hit enter for default "sasdemo"]:
Password:
Succeeded in creating IRM development environment.
```

Note: If this is the first time that you have initialized SAS Infrastructure for Risk Management on your machine, the prompts might vary from those in the previous figure. Also, when you enter the SAS Infrastructure for Risk Management server URL, ensure that you enter the fully qualified domain name, and not only the host name.

The information that you are prompted to enter when you run the **`irmdev init`** command is stored in hidden files (.irm and .authinfo). Storing this information in hidden files prevents the SAS Infrastructure for Risk Management server URL and your user name and password from being visible in your Python code.

To change the configuration, use the **`python irmdev reset`** command:

```
Command Prompt
c:\Python37\Scripts>python irmdev reset
Delete IRM config file: C:\Users\sasdemo\.irm\irm_config.json?
[Y/y/N/n]: y
Delete authinfo file: C:\Users\sasdemo\_authinfo? [Y/y/N/n]: y
```

Share Your Personal Federated Area on a Remote Server

If your SAS Infrastructure for Risk Management server is on a Windows machine, you can share your personal federated area with the name fa.*user-name* and enable Read/Write capabilities. For information about how to share folders, refer to your operating system documentation.

If your SAS Infrastructure for Risk Management server is on a Linux machine, add the following entry to the **smb.conf** file in the**/etc/samba/** directory:

```
[fa.user-name]
    comment = user-name personal FA
    path = SAS-configuration-directory/Levn/AppData/SASIRM/pa/fas/fa.user-name
    force user = SAS-spawned=servers-account
    valid users = user-name
    browseable = yes
    writable = yes
```

Here is an example of the **/etc/samba/smb.conf** file on a Linux server with the user name sasdemo:

```
smb.conf
[fa.sasdemo]
    comment = sasdemo personal FA
    path = /local/install/Config/Levl/AppData/SASIRM/pa/fas/fa.sasdemo
    force user = sassrv
    valid users = sasdemo
    browseable = yes
    writable = yes
```

Mount the Shared Personal Federated Area to the Remote Server

After you have shared your personal federated area, enter the following **python irmdev mount** command and answer the prompts.

Note: The **python irmdev mount** command prompts differ depending on the client's operating system.

```
python irmdev mount
CIFS server URL [Hit enter for "remote-host"]:
CIFS remote mount [Hit enter for "fa.user-name"]:
CIFS local mount directory: /Users/user-name/dev/fa.user-name
User domain: middle-tier-host
User account [Hit enter for "user-name"]: user-name
Enter IRM CIFS mount password:
Mount CIFS to /Users/user-name/dev/fa.user-name successfully.
```

When mounting your personal federated area, note the following key points:

- Some versions of MacOS seem to be prone to performance problems with Samba. If you experience performance problems with a Samba, you can switch to NFS.
- If you do not want to (or cannot) use Samba, you can mount your personal federated area using NFS. The instructions given here are for MacOS and Linux. You need an empty directory at the mount point (for example, **~/dev/fa.*user-name*** in the following example).

```
mount remote-host:/home/local/install/Config/Lev1
/AppData/SASIRM/pa/fas/fa.user-name ~/dev/fa.user-name/
```

- Depending on your umask, you might have to open your personal federated area so that sassrv can manage your files for you:

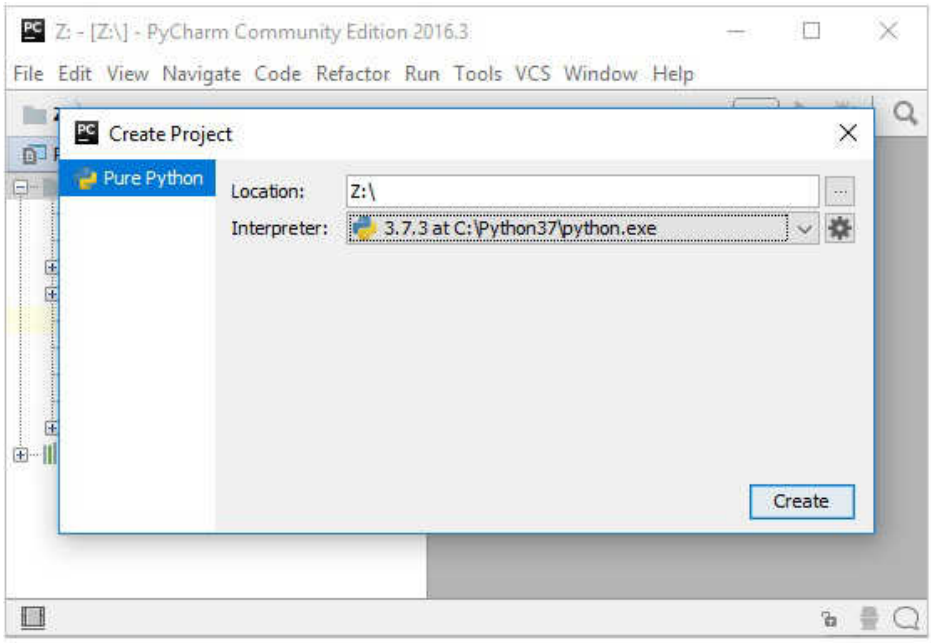
```
chmod -R 775 /home/local/install/Config/Lev1/AppData/SASIRM/pa/fas
/fa.user-name
```

If you need to unmount your personal federated area, use the **`python irmdev umount`** command.

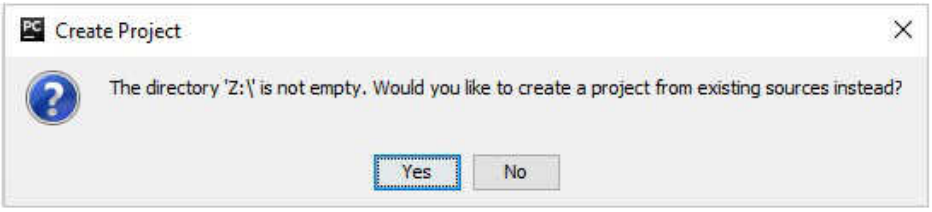
Open PyCharm

When you open PyCharm, use your personal federated area as the root of the new project.

Assuming a local client Windows machine with Z: drive mounted to the personal federated area for user sasdemo, create a new project and select the Project Interpreter that matches the irmdev package installation:



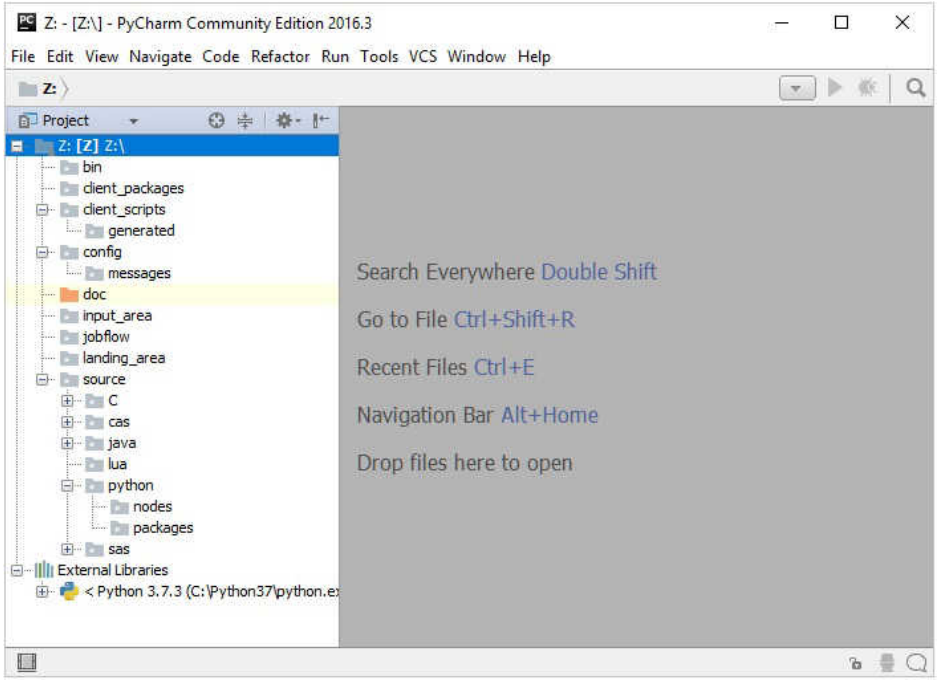
Because you are assigning project to your personal federated area on the remote host, you might receive a message that indicates that the directory is not empty. If you receive this message, select **Yes**.



TIP At this point, it is a good idea to exclude the **`doc`** folder from PyCharm's indexing. This folder is automatically generated, it contains a lot of folders, and it

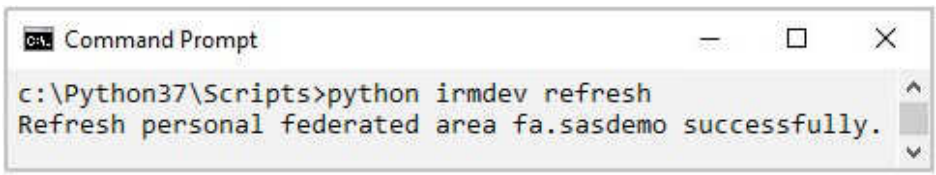
contains no information required by PyCharm. To exclude the **doc** folder, right-click the **doc** folder in the navigation tree and mark the folder as **excluded**.

Here is an example of how your PyCharm window should appear with the doc folder excluded:



Set Up PyCharm

1. To initialize the Python scripting client and to generate the API for your personal federated area, run the **python irmdev refresh** command:



   Note: In addition to generating the task API, when you run the **irmdev refresh** command, the template scripts (script_template.py and task_template.py) are generated in the **client_packages/python/sas folder** of your personal federated area.

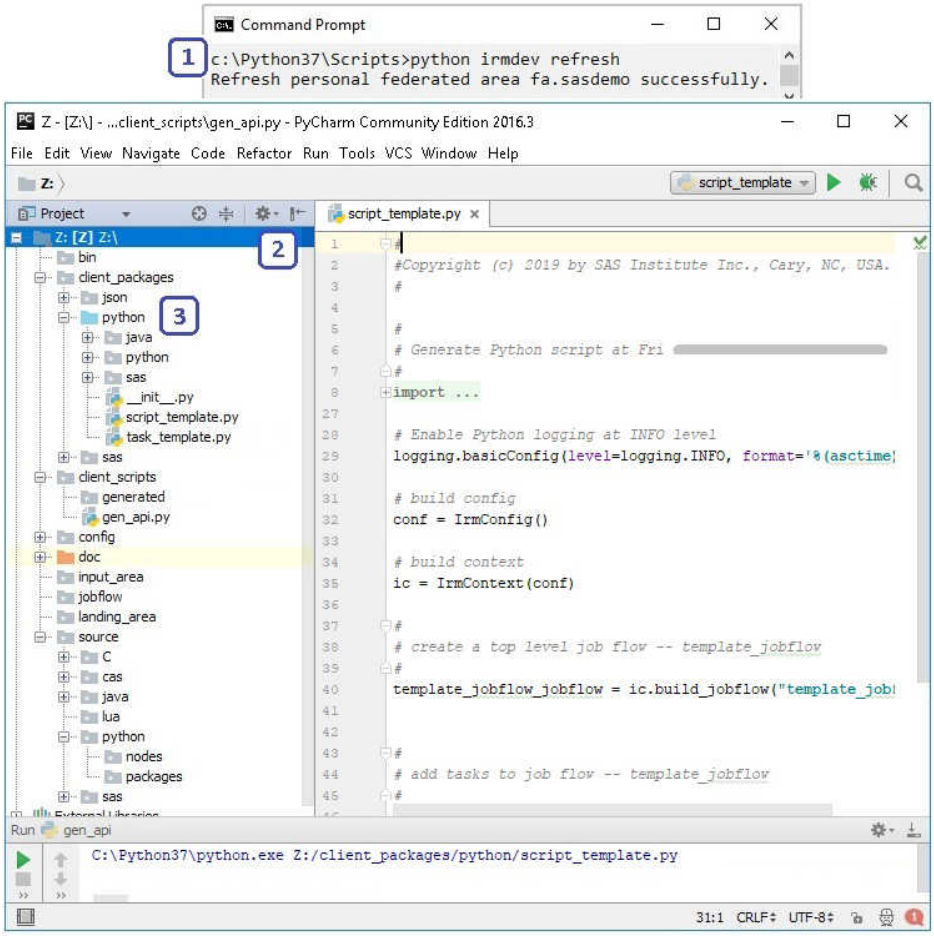
2. Synchronize the project root folder. To synchronize the project root folder, right-click the project root folder and select **Synchronize**.

   Note: You should synchronize periodically to update your local view of your remote personal federated area.

3. Mark the **client_packages/python** folder as the sources root.

   To mark the folder as the sources root, right-click **client_packages/python** and select **Mark Directory as ⇨ Sources Root**.

Here is an image that shows where you perform these three steps in PyCharm:



Sample Federated Area

About the Sample Federated Area

SAS Infrastructure for Risk Management is shipped with several federated areas that are installed when SAS Infrastructure for Risk Management is installed.

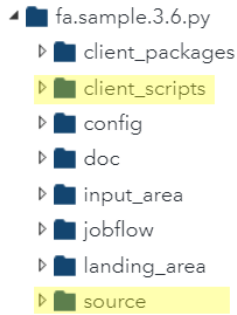
One of the SAS Infrastructure for Risk Management federated areas contains Python sample content such as Python tasks and Python job flows scripts. You can use this sample content to create Python parallel programs or use it as a reference.

The Python sample federated area is named fa.sample.3.6.py. It is located in ***/sas-configuration-directory/*Levn/ AppData/SASIRM/**.

For more information about federated areas, see Chapter 3, “Federated Areas”.

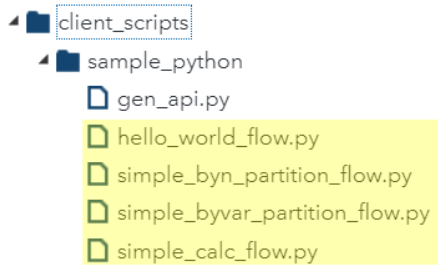
Note: The sample content that you can use to complete the examples in this guide or use as templates to create custom content are highlighted in yellow. The following sections describe that sample content.

Here is an example of the Python sample federated area:



client_scripts Folder

Here is an expanded view of the client_scripts folder:



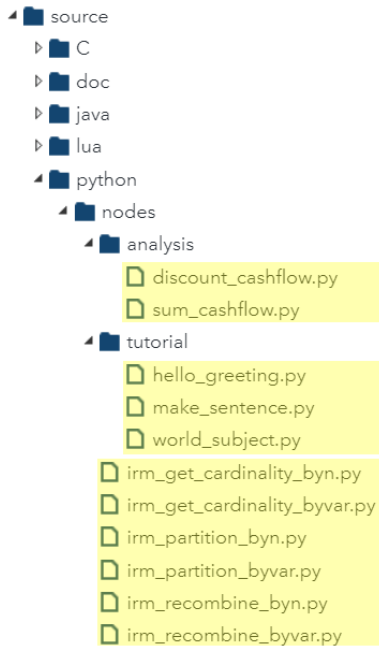
The client_scripts folder contains the following sample job flow scripts that you can use:

- hello_world_flow.py — creates the hello_world_flow job flow. (See "Creating Your First Parallel Program".)
- simple_byn_partition_flow.py — creates the simple_byn_partition__calculation_flow job flow. (See "Using Partitions in a Job Flow".)
- simple_byvar_partition_flow.py — creates the simple_byn_partition__calculation_flow job flow. (See "Using Partitions in a Job Flow".)
- simple_calc_flow.py — creates the simple_calc_flow.py job flow. (See "Using Partitions in a Job Flow".)

source Folder

The source folder contains the source code for the sample content by programming language (for example, C, CAS, Java, Lua, Python, and SAS).

Here is an expanded view of the jobflow folder:



The jobflow folder contains the following Python sample task files (in the **`python/nodes/`** subfolder) that you can use:

- analysis folder
  - discount_cashflow.py — discounts the cashflows for different instruments.
  - sum_cashflow.py — sums the (NPV) for the input data set.
- tutorial folder — contains Python tasks for the "Creating Your First Parallel Program" example.
  - hello_greeting.py — collects a greeting from an input table (word_parts.csv located in **`/sample.fa.3.6.py/landing_area/03312019/`**).
  - make_sentence.py — combines a greeting and a subject into a complete sentence.
  - world_subject.py — collects the subject for a greeting from an input table.
- irm_get_cardinality_byn.py — calculates cardinality for instrument tables based on user-defined number of partitions that is specified in an input configuration data set. If the number of partitions is not specified, the task sets the cardinality to 1.
- irm_get_cardinality_byv.py — calculates cardinality for instrument tables based on user-defined BY variables that are specified in an input configuration data set. If a BY variable is not specified, or no valid BY variable is provided, the tasks sets cardinality to 1 and creates an empty ByVar mapping table.
- irm_partition_byn.py — partitions the input data based on the ByN cardinality table.
- irm_partition_byvar.py — partitions the input table based on the ByVar cardinality table and mapping table.
- irm_recombine_byn.py — recombines a set of ByN partitioned input data into one non-partitioned data output.
- irm_recombine_byvar.py — recombines a set of ByVar partitioned input data into one non-partitioned data output. During the recombine process, the ByVar variables are added back to the output data.

Chapter 2

Create Your First Parallel Program in 60 Seconds

About Creating Your Parallel Program Example

Note: The exercise in this chapter is a high-level example that shows you how quick and simple it is to create a parallel program using SAS Infrastructure for Risk Management. This example uses a sample script and sample tasks that are located in a sample federated area that is included with SAS Infrastructure for Risk Management 3.6. Subsequent chapters in this guide (for example, "Tasks" and "Job Flows") show you how to create your own tasks and parallel programs that use the tasks that you create.

SAS Infrastructure for Risk Management provides a Python scripting client that enables programmers to quickly develop parallel programs (job flows).

Before you can complete the parallel program example in this chapter, you must complete the following tasks:

1. Create your development environment.
2. Install and set up the Python scripting client.

Here are the steps to create a parallel program:

1. Log on to your Python IDE.
2. Develop a job flow script using the Python scripting client package that you will use to create a parallel program.
3. Save and run the job flow script to create the parallel program.

To follow the example in this chapter, note the following key points:

- You use task files that are available to you in the SAS Infrastructure for Risk Management sample federated area (**`fa.sample.3.6.py`**). Therefore, this sample federated area must be installed on the SAS Infrastructure for Risk Management

server and you must copy the task files from the sample federated area to your personal federated area. Use the same directory structure.

The task files are named hello_greeting, make_sentence, and world_subject.

- Copy the sample script or scripts that are located in the **`client_scripts/sample_python`** in the sample federated area **`(fa.sample.3.6.py)`** to your personal federated area. Use the same directory structure.

  In this example, you use the hello_world_flow.py sample script.

- Do not copy and paste code samples from the PDF version of this guide. The required indents and spacing are not preserved when a code sample is copied and pasted from the PDF.

Creating Your First Parallel Program

High-Level View of Creating the Job Flow

1. In PyCharm, open a new project. Use your personal federated area as the root of the new project.
2. Create a new Python file in the **`client_scripts`** folder of your personal federated area.
3. In the new Python file, copy the following job flow script. The job flow script consists of Python scripting client methods that you combine with tasks to create a parallel program.

```
# ***************************************************************
# NAME:     hello_world_60second_flow_script.py                 *
# PURPOSE:  Python Scripting Client program (script) to create  *
#           the job flow hello_world_flow using 3 python tasks  *
#           that are in the shipped sample federated area       *
# ***************************************************************
from __future__ import absolute_import, division,\
    print_function, unicode_literals
from irm.context import IrmContext
from irm.config import IrmConfig
from python.nodes.tutorial import tutorial
import logging

# Enable Python logging at INFO level
logging.getLogger().setLevel(logging.INFO)
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s'
                           ' %(name)s - %(message)s')

# Step 1 Scripting Client Initialization - Build config and context
conf = IrmConfig()
ic = IrmContext(conf)

# Step 2  Create a job flow - hello_world_flow in sample_python
example_jf_ref = ic.build_jobflow(
    "hello_world_flow", "sample_python")
```

```
# Step 3  Add tasks from fa.sample.3.6.py
example_jf_ref.add(tutorial.hello_greeting(substitution_dict={
        "%in_words":"staging.word_parts.csv",
        "%out_greeting":"example.hello.csv",
}),name="hello_greeting")

example_jf_ref.add(tutorial.world_subject(substitution_dict={
    "%in_words":"staging.word_parts.csv",
    "%out_subject":"example.world.csv",
}),name="world_subject")

example_jf_ref.add(tutorial.make_sentence(substitution_dict={
    "%in_greeting":"example.hello.csv",
    "%in_subject":"example.world.csv",
    "%out_sentence":"example.hello_world.csv",
}),name="make_sentence")

# Step 4  Save the jobflow definition to the IRM Server
example_jf_ref.save()

# Step 5  Execute the job flow instance in the IRM Server
example_jf_ref.run()
```

4. Save and run the job flow script in PyCharm by right-clicking the script's file name and selecting **Run 'hello_world_60second_flow_script.py'**.

Detailed View of Creating the Job Flow

The first few lines of the job flow script include the following details:

- comments that describe the parallel program name and its purpose
- import statements that import Python modules that are needed for the program
- logging statements that enable Python logging

The rest of the job flow script includes the Python scripting client steps to create your first parallel program.

Here is a detailed description of each step that is highlighted in the job flow script that you just created:

1. Initialize the scripting client.

   ```
   conf = IrmConfig()
   ic = IrmContext(conf)
   ```

   This step initializes the Python scripting client and sets the context that is your entry point to the scripting client.

   `IrmConfig()` uses configuration information that you supplied when you ran the **`irmdev init`** command. Therefore, you must have successfully run the **`irmdev init`** command before you run this job flow script. (See "Initialize SAS Infrastructure for Risk Management on Your Local Machine".)

Note: This example uses the default values for the job flow configuration parameters (for example, the configuration set ID and base date). However, in a typical development scenario, you will likely override the default configuration parameters. For information about Python getter and setter methods in IrmConfig that you can use to override default values, see Appendix 1, "Scripting Client Reference".

2. Create the job flow definition.

```
example_jf_ref = ic.build_jobflow(
    "hello_world_flow", "sample_python")
```

Use the context method **`build_jobflow`** to create a job flow object. This job flow object (**`example_jf_ref`**) is used in subsequent steps of the job flow script.

In this example, you create a job flow definition that is named hello_world_flow to be built in the **`sample_python`** category.

3. Add tasks to the job flow definition.

```
example_jf_ref.add(tutorial.hello_greeting(substitution_dict={
        "%in_words":"staging.word_parts.csv",
        "%out_greeting":"example.hello.csv",

}),name="hello_greeting")

example_jf_ref.add(tutorial.world_subject(substitution_dict={
    "%in_words":"staging.word_parts.csv",
    "%out_subject":"example.world.csv",
}),name="world_subject")

example_jf_ref.add(tutorial.make_sentence(substitution_dict={
    "%in_greeting":"example.hello.csv",
    "%in_subject":"example.world.csv",
    "%out_sentence":"example.hello_world.csv",
}),name="make_sentence")
```

Use the **`add`** method to add the tasks to the hello_world_flow job flow definition, which was assigned to a job flow object named example_jf_ref in Step 2.

In this step, the following three tasks are added to the hello_world_job_flow_definition.

- tutorial.hello_greeting
- tutorial.world_subject
- tutorial.make_sentence

You can add the tasks in any order to the job flow definition. SAS Infrastructure for Risk Management automatically deduces data dependencies from the tasks and schedules the parallel execution of the tasks.

Note: These three tasks are available in the SAS Infrastructure for Risk Management Python sample federated area **`fa.sample.3.6.py`** in **`source\python\nodes\tutorial`**. Their corresponding task APIs were created when you ran the **`irmdev refresh`** command. (See "Set Up PyCharm".)

4. Save the job flow definition to the SAS Infrastructure for Risk Management server.

```
example_jf_ref.save()
```

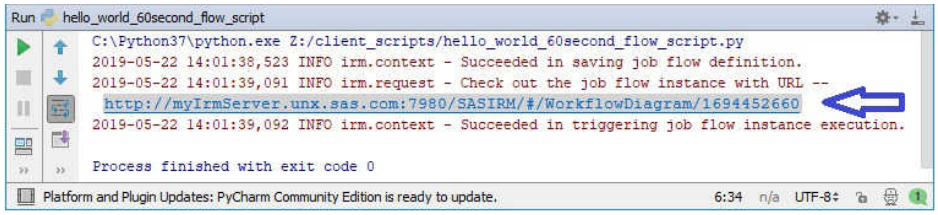
Use the **`save`** method to save the hello_world_flow job flow definition. It was assigned to a **`job flow`** object that was named example_jf_ref in Step 2 to the SAS Infrastructure for Risk Management server.

5. Run the job flow on the SAS Infrastructure for Risk Management server.

```
example_jf_ref.run()
```

Use the **`run`** method to create and execute a job flow instance of the hello_world_flow job flow. In addition to creating and executing the job flow instance, SAS Infrastructure for Risk Management generates a job flow instance diagram that you can view in the SAS Infrastructure for Risk Management web application.

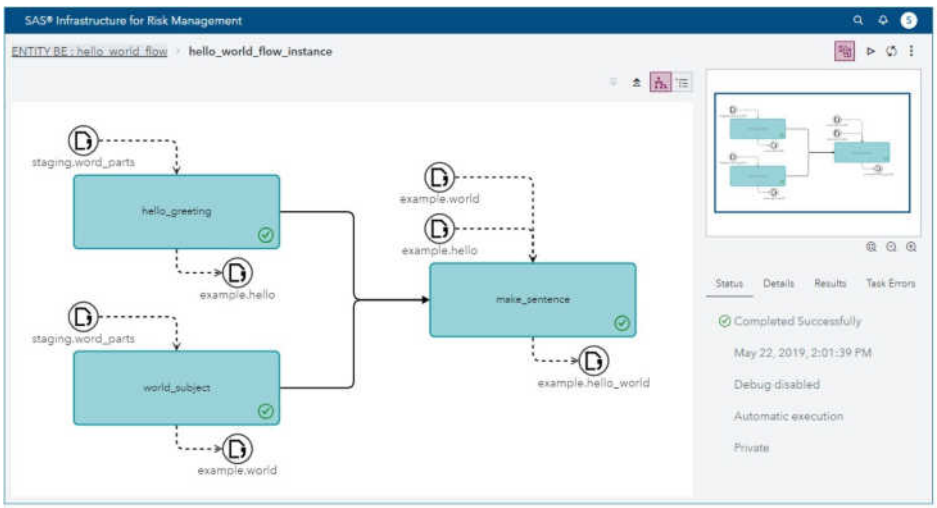
When you run the scripting client program to create your first parallel program in PyCharm, logging messages are written to your PyCharm console. If the job is successful, the URL of the created job flow instance is displayed in the log message:



Viewing Your First Parallel Program

To view the job flow instance that you created, click the URL that is displayed in the PyCharm Run Tools window from the successful execution of the job flow creation script. (See the preceding figure.) Alternatively, you can log on to theSAS Infrastructure for Risk Management web application and select your job flow instance from the instance list view.

Here is how your first parallel program appears in the SAS Infrastructure for Risk Management web application:



Chapter 3

Federated Areas

About Federated Areas

A federated area is a folder structure that conforms to specific SAS Infrastructure for Risk Management rules. It is a storage area for content. As a SAS Infrastructure for Risk Management programmer, you must organize the content that you create in a federated area.

Federated areas enable content developers to deploy content independently of the SAS Infrastructure for Risk Management platform.

In addition, federated areas provide reproducibility: the ability to run existing content repeatedly. Deploying new content or a new platform should never break existing content.

The content of a federated area consists of the following elements:

- job flow definition — a file that contains the tasks and subflows required to complete a job flow.
- code — task files (nodes), string message data, and macros
- input files — SAS data sets, CSV files, Microsoft Excel templates, or XBRL templates
- documentation and tooltips files — information that is presented to the end user through the user interface

When you install SAS Infrastructure for Risk Management 3.6, the following federated areas are installed:

- `fa.0.3.6` — contains only elements that are required to make the platform run. There is no content in the platform federated area.

- **`fa.sample.3.6`** — contains SAS sample content that you can use to test the SAS Infrastructure for Risk Management installation, create SAS parallel programs, and to use as a reference.
- **`fa.sample.3.6.py`** — contains Python sample content that you can use to test the SAS Infrastructure for Risk Management installation, create Python parallel programs, and to use as a reference.
- **`fa.`*`user_name`*** — a personal federated area that is created on demand. You must have a personal federated area if you want to develop your own content.

It is important to note the following rules about federated areas:

- Are independent, but they must be designed to not conflict with other federated areas.
- They must not be altered, modified, or deleted after they are deployed.

  The exceptions to this rule are:

  - You can upload data to the input area or the landing area of a federated area other than the platform federated area.
  - You can modify your personal federated area.
- Tasks, job flows, and input data are *federated content*, which can be shared across all federated areas. Therefore, do not change the definition of tasks, job flows and data. Doing so can cause unpredictable results.

  All other content is local to a federated area and cannot be shared with other federated areas.
- As a SAS Infrastructure for Risk Management programmer (content developer), you can modify your personal federated area. Typically, you use the scripting client to make modifications. The SAS Infrastructure for Risk Management manages the integrity of the job flow instances that reference your personal federated area. You cannot delete your personal federated area. However, you can delete the contents of your personal federated area.

  Note: Before you delete the contents of your personal federated area, you must delete any job flows that you created using the content from your personal federated area.

Your Personal Federated Area

Your personal federated area is where you develop your content.

It is important to note the following key points about your personal federated area:

- SAS Infrastructure for Risk Management can read all personal federated areas. However, you cannot see other programmer's personal federated areas — just your own.
- Your personal federated area is at the top of the federated area precedence chain. It is always the first federated area that is searched by SAS Infrastructure for Risk Management for federated content. An exception is if you select the federated area below your personal federated area (time machine) when you create a job flow instance in the SAS Infrastructure for Risk Management user interface. For more information about creating a job flow instance based on a federated area that is not at

the top of the inheritance lineage, see "Create an Instance" in *SAS Infrastructure for Risk Management: User's Guide*.

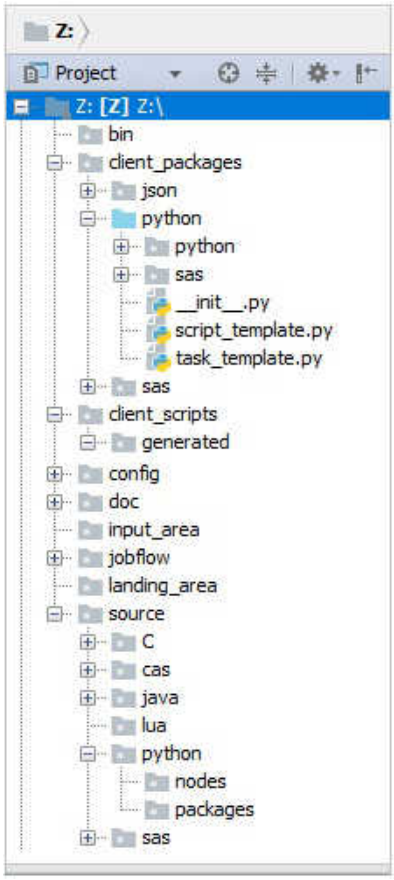
- You cannot publish any job flow instances that you create from your personal federated area.

For information about generating your personal federated area, see "Set Up Your Development Environment".

Folders in a Federated Area

All federated areas have the same basic folder structure. Custom federated areas might contain additional folders or fewer folders than the ones described in this section. SAS Infrastructure for Risk Management does not require all folders to be included in a federated area. However, specific types of content must be in the appropriate folder.

Here is an example of the basic folder structure of a federated area:



Here is a partial list of the folders that are in a federated area:

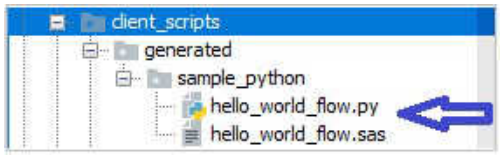
- **`client_packages`** — contains generated API packages and other required Python scripting client files. In addition, the folder includes the following sample templates:
  - **`script_template.py`** – scripting client program sample that you can use to create a simple job flow
  - **`task_template.py`** – simple task program sample that includes example syntax for the required task header

The client_packages folder is added to your personal federated area by the scripting client. If you create a read-only federated area that you want to share with others, the client_packages must be the federated area that you want to share in order for other users to be able to create job flows that use tasks that are defined in the federated area.

Note: To generate or refresh the files in the **client_packages** folder, run the **python irmdev refresh** command. The client_packages folder should be read-only.

- **client_scripts** — contains the scripting client programs (job flow scripts) that you create and store here. The **generated** folder in the **client_scripts** folder contains the automatically generated scripting client scripts (both a SAS version and a Python version). These scripts are generated for you when you create a job flow using the scripting client. For example, if you execute a scripting client program that creates a job flow that is named hello_world_flow in the **sample_python** folder, SAS Infrastructure for Risk Management automatically generates both a SAS version and a Python version of the scripting client program files. You can use these files to re-create the job flow.

  Here is an example of the two scripting client program files in the **sample_python** folder:

- **config** — contains files that a programmer uses to configure the behavior of job flows. Specifically, this folder contains the following files and folders:
  - **job_flow_definitions.csv** — lists the job flows that are available in the SAS Infrastructure for Risk Management web application.
  - **libnames.txt** — maps libraries for the input data that you use in your job flows.
  - **messages** — contains the labels to the nodes, job flows, and the inputs and outputs that are visible in the SAS Infrastructure for Risk Management web application.
- **doc** — contains the task documentation that is generated when you invoke the scripting client **generate_task_doc(self)** method.
- **input_area** — is the area in which you can load data directly into a federated area when the SAS Infrastructure for Risk Management server is running. For information about loading data, see "Loading Data into Your Personal Federated Area Using Live ETL".
- **jobflow** — contains sample Python job flow definitions or subdirectories that contain job flow definitions. Subfolders within the job flow folder are displayed as categories in the SAS Infrastructure for Risk Management web application. Only a single level of folder can contain a subflow folder. A job flow subfolder can contain a subflow folder.
- **landing_area** — is the read-only data mart of a federated area. The landing area contains the data objects that are required for the job flows that are defined in that federated area. The SAS Infrastructure for Risk Management server must be shut down before you load data into the landing area unless you use live ETL to upload data. For information about loading data, see "Loading Data into Your Personal Federated Area Using Live ETL".

- **`source`** — contains the source code for your content by language (for example, C, CAS, Java, Lua, Python, and SAS).

  The folders that appear in a language-specific folder is dependent on the language. However, each language-specific folder contains a **`nodes`** folder that contains the task files. For example, **`source / python / nodes`** contains the .py task files that are directly called by job flows. To make code more manageable, you can create a single-level hierarchy of subfolders in the nodes folder.

Base Date Folder Entity Table

At least one federated area in a SAS Infrastructure for Risk Management deployment must contain a **`base date`** folder in its landing_area. The **`base date`** folder contains input data for a period of time. The naming convention for a **`base date`** folder is MMDDYYYY (for example, 03312019).

Each **`base date`** folder must contain an *entity table*. SAS Infrastructure for Risk Management uses entity tables to configure aspects of the SAS Infrastructure for Risk Management web application. For example, the options that are available on the drop-down menus are determined by the configuration of the entity table.

Typically, the data in the entity table is static. However, data can be updated via see live ETL.

CAUTION:

Entities that are being used cannot be deleted. Deletion of an entity row being used would render existing job flow instances invalid.

Note: Entity tables in the highest federated areas take precedence over the entity tables in lower federated areas.

Here is an example of the entity table:

	ENTITY_ID	ENTITY_NM	ENTITY_ROLE_CD	GROUP_ID	COUNTRY_CD	GROUP_ASSESSMENT_CD	REPORTING_CURRENCY
1	MAIN	MAIN	BOTH		NL	CFI	EUR
2	ENTITY_BE	ENTITY BE	SOLO	MAIN	BE	CFI	EUR
3	REGIONAL_GROUP	REGIONAL GRO...	GROUP	MAIN		CFI	EUR
4	ENTITY_CH	ENTITY CH	SOLO	REGIONAL_GROUP	CH	CFI	CHF
5	ENTITY_IT	ENTITY IT	SOLO	REGIONAL_GROUP	IT	CFI	EUR

where:

- ENTITY_ID — (Required) alphanumeric identifier that specifies the organizational or operational unit of an organization. A value that contains spaces or any special characters except for an underscore (_) is rejected when SAS Infrastructure for Risk Management is started.
- ENTITY_NM — specifies the name of the organizational or operational unit of an organization. If a value is not specified for this attribute, the value for the ENTITY_ID displays in the SAS Infrastructure for Risk Management web application.
- ENTITY_ROLE_CD — (Required) specifies which calculation level options are available in an entity. Here are the possible values:
  - BOTH — configures SAS Infrastructure for Risk Management to enable calculations at the solo and group level.
  - GROUP — configures SAS Infrastructure for Risk Management to enable calculations at a group level, which includes the subsidiary units within an entity.

  - SOLO — configures SAS Infrastructure for Risk Management to enable calculations for the chosen entity as a single unit.
- GROUP_ID — (Required) specifies the identification of the group to which a particular entity belongs. Note that a solo entity can belong to a group (ENTITY_BE), and also a group can belong to another group (REGIONAL_GROUP).

Note: The entity table can contain any number of variables, but the ENTITY_ID, GROUP_ID, and ENTITY_ROLE_CD table attributes are required.

Loading Data into Your Personal Federated Area Using Live ETL

Typically, SAS Infrastructure for Risk Management reads initial input data objects from the **`landing_area`** folder of a federated area. Because SAS Infrastructure for Risk Management can simultaneously run multiple job flows and tasks that might access the input data objects from the landing area, you should avoid manually loading your data to the landing area while the SAS Infrastructure for Risk Management server is running. Loading data to the **`landing_area`** folder of a federated area while the SAS Infrastructure for Risk Management server is running can cause errors (for example, file locking issues or incorrect results).

To avoid potential issues and errors, load your input data to the **`input_area`** folder of your personal federated area. Then SAS Infrastructure for Risk Management automatically uses live ETL to copy the input data objects from the **`input_area`** folder to the **`landing_area`** folder.

When using live ETL to upload data, note that you can upload all existing data using live ETL. There is no restriction on the number of tables that you can upload at the same time. In addition, you can create entities, configuration sets, and new base dates via live ETL. Live ETL supports the creation of new input data objects. However, it does not support deleting input data objects.

To load input data when the SAS Infrastructure for Risk Management server is running, follow these steps:

1. Load your input data objects to the **`input_area`** folder of your personal federated area.
2. Modify the marker file to update the file's timestamp. It is named last_update.txt and is located in the **`input_area`** folder. The timestamp triggers live ETL,and SAS Infrastructure for Risk Management copies the input data from the **`input_area`** folder to the **`landing_area`** folder when it determines that input data objects are not being accessed by any job flow tasks.

Here is a simple Python program that you can execute in your Python IDE that loads a CSV data set to the landing area of your personal federated area:

```
# ------------------------------------------------------------------
# NAME:     load_mydata_via_trigger_live_etl.py
#
# PURPOSE:  Create my_words.csv, update libnames.txt, trigger live
#           ETL to load the data file to my Personal Landing Area
# ------------------------------------------------------------------
import logging
```

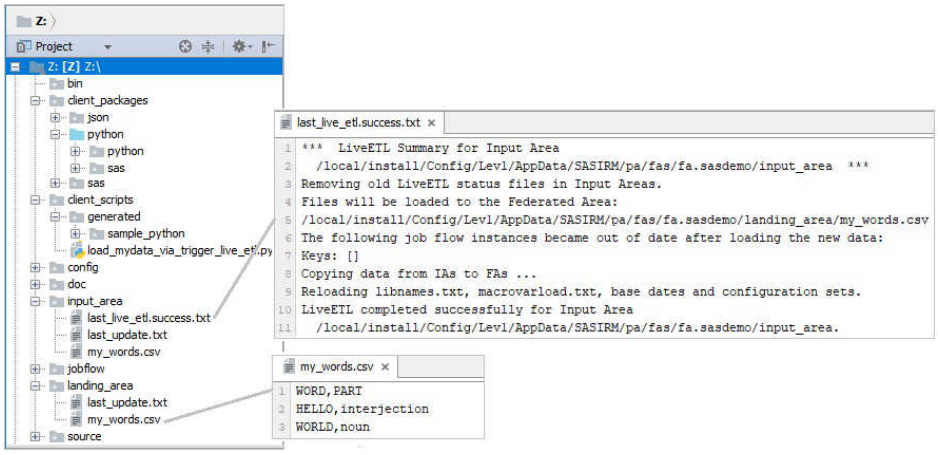
```
# Enable Python logging at INFO level
logging.basicConfig(level=logging.DEBUG,
    format="%(asctime)s %(levelname)s %(name)s - %(message)s")

# Step 1 Create new csv file my_words in my personal FA's input_area
with open("../input_area/my_words.csv", "w") as f:
    f.write("WORD,PART\n")
    f.write("HELLO,interjection\n")
    f.write("WORLD,noun\n")

# Step 2 Create the library entry for MYDATA in libnames.txt
with open("../config/libnames.txt", "w") as f:
    f.write("MYDATA=%la")

# Step 3 Touch last_update.txt to trigger liveETL, refresh libnames
with open("../input_area/last_update.txt", "w") as f:
    f.write("")
```

Here is a detailed description of each step that is highlighted in the preceding Python program example:

1. Creates a data file (my_words.csv) in the **`input_area`** folder of your personal federated area. In this example, the program dynamically creates the data file. However, a typical use case would be a process to copy or extract the data to this location.
2. Adds an entry to the libnames.txt file in the **`config`** folder of your personal federated area. In this example, the libref MYDATA is created and refers to the **`landing_area`** (%la) folder of the personal federated area.
3. Modifies the last_update.txt file in the **`input_area`** folder of your personal federated area.

   Modifying the last_update.txt file in the **`input_area`** folder triggers SAS Infrastructure for Risk Management to invoke live ETL. Live ETL runs in real time and copies the input data objects from the **`input_area`** folder to the **`landing_area`** folder.

After you execute the sample program, you should see the files added to your personal federated area. After a short delay (up to 30 seconds), the landing area is populated with your data:



After the updates have occurred, note the following:

- To reflect these changes to your personal federated area, you must synchronize your mounted drive.
- SAS Infrastructure for Risk Management adds an additional file (last_live_etl_success.txt) to the **`input_area`** folder to notify you of the success or failure of the live ETL process.

Sharing Content from Your Personal Federated Area

After developing and testing your content in your personal federated area, you might want to make your content available to others.

As a SAS Infrastructure for Risk Management programmer, here is the process for sharing the content that you developed in your personal federated area:

1. Develop and test your content in your personal federated area.
2. From the top folder of your personal federated area, compress the federated area with any percentage of compression to create a ZIP file of your personal federated area.
3. Ask your system administrator to install your zipped personal federated area as a stand-alone federated area on the SAS Infrastructure for Risk Management server.

For more information about installing a stand-alone federated area, see "Install a Stand-Alone Federated Area without a Server Restart" in *SAS Infrastructure for Risk Management: Administrator's Guide*.

Chapter 4

Tasks

About Tasks

The basic building block of a SAS Infrastructure for Risk Management job flow is the *task*. Tasks are independent programs that you combine in a job flow to complete a job.

A task is contained in only one file (task file). The task file consists of two required parts:

- file header — a syntax-specific header section in which you must declare the task-specific inputs, outputs, or both inputs and outputs. It is also a good practice to document the function of the task in the header. The file header is used by SAS Infrastructure for Risk Management to identify your program as a task and create task packages that are available for you to use when developing job flows.
- program — the analytical or reporting element of processing that is packaged as a complete program. This is your code that accomplishes a complete unit of work.

Here are the file header and program parts of the hello_greeting.py task file that is located in the SAS Infrastructure for Risk Management sample federated area (**fa.sample.3.6.py**):

hello_greeting.py

File Header

```
##  \file
##  \brief The node gets a greeting from an input table
##
##  <b> Identified Inputs and outputs </b>
##
##  \param[in]  %in_words      Input word parts data
##
##  \param[out] %out_greeting  Extracted hello greeting data
##
##  \ingroup tutorial
```

Program

```
import pandas as pd
# input_output_df_dict {'in_words': df_in, 'out_greeting': df_out }
def irm_run(input_output_df_dict):
    # Specify task input table as df_in
    df_in = input_output_df_dict['in_words']

    # Get subset (rows where word='Hello') from input table df_in
    df_out = pd.DataFrame({'GREETING': df_in['WORD']})
    df_out = df_out.loc[df_out['GREETING'] == 'Hello']

    # Put the subset in the task output table
    input_output_df_dict['out_greeting'] = df_out
```

Before you begin to use or create tasks, note the following key points:

- Tasks must declare all inputs and outputs.
- You can combine individual tasks in a job flow.
- When you use multiple tasks in your job flow, SAS Infrastructure for Risk Management determines the order and parallelism in which the tasks are executed based on the availability of the inputs and the number of cores that are available for processing.
- You can use predefined tasks (for example, tasks that are delivered in a federated area) in your job flow. In addition, you can use tasks that you create in your own personal federated area.
- The name of the task file, which is the Python program, should be short and in lowercase letters. You can use underscores to improve readability. The task file name, excluding the file extension, is the name of the task as it appears in the SAS Infrastructure for Risk Management web application.

Anatomy of the Task File Header

About the Anatomy of the Task File Header

SAS Infrastructure for Risk Management requires that all tasks contain a properly constructed header that conforms to Doxygen syntax guidelines. (See http://doxygen.nl/.) At a minimum, the file header must contain the task's inputs and outputs and the location of the tasks.

The task header is a type of comment header that SAS Infrastructure for Risk Management uses to create tasks and the API. The task header is not a typical comment header that is ignored by SAS Infrastructure for Risk Management. It is recognized and processed by SAS Infrastructure for Risk Management as a task header.

The Doxygen comment lines in the task header must be a group of consecutive lines. Each line must begin with a ## (double hash mark). In the following example, a regular comment (single hash mark) is used in lines 1 through 3. These lines are ignored by SAS Infrastructure for Risk Management and are not considered part of the task header. The remaining lines in the task header (lines 4 through 16) that begin with double hash mark are recognized and processed by SAS Infrastructure for Risk Management as the task header:

```
1   #
2   #   Copyright (C) 2019 SAS Institute Inc. Cary, NC, USA
3   #
4   ##
5   ##  \file
6   ##  \brief   The node partitions input based on cardinality byN.
7   ##
8   ##  <b> Identified Inputs </b>
9   ##  \param[in] %DS_TO_PART         Input that needs the partition
10  ##  \param[in] %CARDINALITY_DS     Input cardinality data
11  ##
12  ##  <b> Identified Outputs </b>
13  ##  \param[out] %PART_DS           Output data after partition.
14  ##
15  ##  \ingroup nodes
16  ##
```

Documenting the Task

In the file header, you can create user documentation that describes the function of the task. When you process the content that you are developing, this documentation is parsed and made available in HTML format for users to access. You must document a task for tooltips and node-level help.

Here is an example:

```
## \file
## \brief  The node gets a greeting from an input table
## \details  Include detailed information about the task, what it does.
```

Note: Use the scripting client **`generate_task_doc()`** method in your program to generate documentation for the task.

Defining Task Inputs and Outputs

The inputs and outputs for a task are treated as input and output function parameters. Therefore, in each task file, you must document inputs and outputs using the following Doxygen syntax:

```
## \param[dir] data-object-name data-object-description
```

where:

- *dir* — specifies whether the data object is an input (in) or an output (out).

- *data-object-name* — specifies the name of the data object that you are defining as input data or output data for the task.
- *data-object-description* — (Optional) describes the data object. The description can contain multiple spaces and tabs.

Here is an example:

```
##  \param[in]  %in_words      Input word parts data
##  \param[out] %out_greeting  Extracted hello greeting data
```

Defining Task Inputs and Outputs Using Parameter Substitution

Parameter (or token) substitution enables you to reuse existing tasks for different jobs. When you specify a parameter substitution for input data or output data, the code that is associated with the task references the input data or output data by the parameter. The convention is the same for all tasks, regardless of the programming language that you use.

To use parameter substitution, specify the parameter name:

```
##  \param[dir] %parameter-name table-description
```

where % (the percent sign) denotes parameter substitution.

Here is an example:

```
##  \param[in]  %in_words      Input word parts data
##  \param[out] %out_greeting  Extracted hello greeting data
```

Defining Task Inputs and Outputs Using Parameter Substitution with Default Value Specification

If there is a default substitution value, you assign the value using the equal sign (=) and the corresponding parameter substitution value.

For example, if you have substitution parameters named in_words and out_greeting and you assign their default values to staging.word_parts.csv and example.hello.csv, you use the following syntax:

```
##  \param[in]  %in_words=staging.word_parts.csv  Input word parts data
##  \param[out] %out_greeting=example.world.csv   Extracted world subject data
```

Note: When defining parameter substitution, ensure that there are no spaces or tabs before or after the equal sign (=).

Anatomy of the Task Program

About the Anatomy of the Task Program

The program part of the task is simply the Python program that performs the analytical or reporting element work of the task. SAS Infrastructure for Risk Management triggers execution of the Python code via a public entry point function that is named **irm_run**. The **irm_run** entry point function is defined with the input and output data objects of the task that are passed in as arguments:

```
def irm_run(input_output_df_dict):
```

The arguments (input_output_df_dict) is a dictionary that contains the input and output Pandas DataFrames of the task (when the **`irm_run`** function is invoked). The keys for the dictionary are the input/output names, which can be the following elements:

- file names of the input or output data objects
- variable names if substitution is used

About SAS Data Conventions

Here is information about SAS 9.4 data that you should be aware of before working with tasks.

SAS Column Naming Conventions

- A column name is always given as a string, whereas in Pandas, it is possible to use numbers as column names as well.
- The length of a column name can be up to 32 characters. If a column name is longer than 32 characters, it is truncated to 32 characters. If truncating a column name would result in identical names, SAS generates a unique name by replacing the last character with a number.
- If the name of the column is a number, SAS adds an underscore (_) in front of the number to make the name a string. For example, the column name **`2020`** is replaced with **`_2020`**.
- Spaces and special characters, such as &, -, (, and + are replaced with an underscore (_).

SAS Data Types

There are three data types that are supported for SAS — character data, numeric data, and date and time data. Therefore, the Pandas DataFrames will contain these three types of data.

- A character data value is given as a string.
- A numeric data value is given as a floating-point value.
- A date and time data value is given as a NumPy **`datetime64`**.

If you need the values of a column to be in another data type than one of the above, you will need to convert the data types. For more information about converting data types, see the Python documentation.

Input and Output Data Objects

All input and output data objects must be declared in the task file header. For example, here is a task file header input declaration that uses parameter substitution with a default value:

```
## param[in] %input=static.analytics_option.csv
```

Note: The three-part data name of the default input object shown in the preceding example specifies the location: SAS library, file name, and file extension.

To process the task inputs and outputs, SAS Infrastructure for Risk Management provides the following support:

- Provides a Python dictionary that is named input_output_df_dict. This dictionary reads inputs and writes outputs. This dictionary contains the task's input and output Pandas DataFrames.
- Provides support for multiple file types (.csv, .sas7bdat, .pickle, and so on).
- Loads the Pandas DataFrame objects from the input data objects and puts them into the input_output_df_dict dictionary using keys in the form of %*var-name* or *three-part data name*.
- Sets the resulting output Pandas DataFrame objects into the input_output_df_dict using keys in the form of %*var-name* or *three-part data name*.
- Provides transparent support of SAS data sets by automatically handling the serialization (Pandas DataFrame to sas7bdat) and deserialization (sas7bdat to Pandas DataFrame). (The user simply uses the Pandas DataFrame passed as input to input_output_df_dict or the Pandas DataFrame returned as output.)

Here is an example of a task that shows the task file header declarations and the task program that uses the input_output_df_dict dictionary:

```
1   ## \file
2   ## \brief The node illustrates input_output_df_dict
3   ## \param[in] %my_input    my task input file
4   ## \param[out] %my_output  my task output file
5   ##
6   import pandas as pd
7
8   # input_output_df_dict {'my_input':df_in, 'my_output':df_out}
9   def irm_run(input_output_df_dict):
10
11      # Specify task input table as df_in
12      df_in = input_output_df_dict['my_input']
13
14      # Subset task input to create output DataFrame
15      df_out = pd.DataFrame({'GREETING': df_in['WORD']})
16
17      # Put the subset in the task output table
18      input_output_df_dict['my_output'] = df_out
19
```

1. The task file header must declare the task input and task output data objects. This example uses parameter substitution without a default value and defines a single input file (variable named my_input) and a single output file (variable named my_output).
2. The **`irm_run`** entry point function is defined with the task's input and output data objects that are passed in as arguments via the input_output_df_dict dictionary. In this example, the input_output_df_dict dictionary name-value pairs (**`'my_input' : df_in, 'my_output' : df_out`**) are listed in the comment:

   ```
   # input_output_df_dict {'my_input':df_in,'my_output':df_out}
     def irm_run(input_output_df_dict):
   ```

3. The substitution token (my_input) is used to retrieve the Pandas DataFrame from the SAS Infrastructure for Risk Management server:

   ```
   # Specify task input table as df_in
     df_in = input_output_df_dict['my_input']
   ```

4. The substitution token (my_output) is used to write the output Pandas DataFrame to the SAS Infrastructure for Risk Management server:

```
# Put the subset in the task output table
  input_output_df_dict['my_output'] = df_out
```

Macrovars Inputs

SAS Infrastructure for Risk Management supports the concept of *marcrovars*. A marcrovar is a table that contains a set of names, values, and descriptions and is exposed to a task with these values.

To expose a macrovar to the Python task, use the **get_macrovar** function:

```
def get_macrovar(input):
"""
    Obtain the dictionary of name-value pairs for the
    macrovar data frame passed in.

Args:
    input: The data frame with the content of the macrovar
    table.

Returns:
    The dictionary of the name: value pairs.
"""
```

Note: To view the description field of a macrovar, use the Pandas DataFrames that are passed as inputs.

Creating Tasks

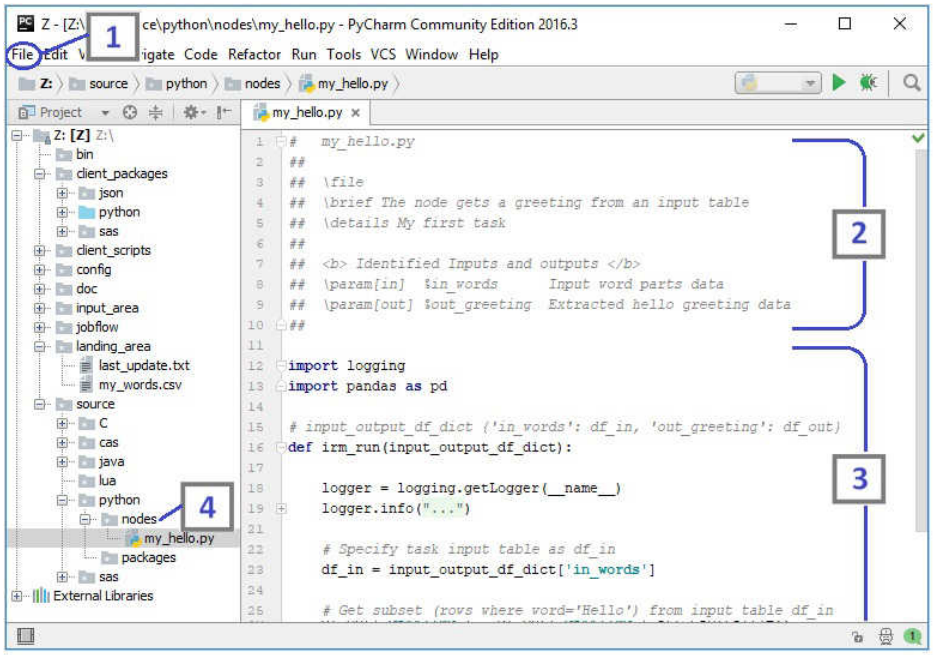
About Creating Tasks

1. In PyCharm, create your task as a new Python program by selecting **File** ⇨ **New** ⇨ **Python File** and name it my_hello.py.

   Note: Python task files are stored in the **source/python/nodes** folder in your personal federated area. If necessary, you can create subfolders in the **nodes** folder to organize your tasks.

2. Enter the task file header. You might need to edit the header after you create the program and identify all your inputs and outputs.
3. Enter the task program. Make sure that you have tested and verified your program code.
4. Save the task file (using a valid name) to the **nodes** folder in your personal federated area. The task file name must conform to the standard Python naming conventions.

Here is an illustration of the four steps:



Here are three Python task files that you will use later in this document to create a job flow:

- my_hello.py
- my_sentence.py
- my_world.py

You can copy the code samples for the three example tasks in order to create your tasks. These are tasks that you will create and save in your personal federated area in the **`source/python/nodes`** folder.

If you choose to follow the examples in this guide, ensure that the task file names that you use are the same as those that are used in the examples. This enables you to use the example code without having to make changes to the code.

Note: To make the process easier, you can copy and paste the code samples without having to edit the code.

Ensure that the following requirements have been met:

- The my_words.csv file has been loaded to the landing area. (See "Loading Data into Your Personal Federated Area Using Live ETL".)
- The libnames.txt file has been loaded to the **`config`** folder and contains the entry MYDATA=%la. (See "Loading Data into Your Personal Federated Area Using Live ETL".)
- You have Read and Write access to your personal federated area **`source/python/nodes`** folder.

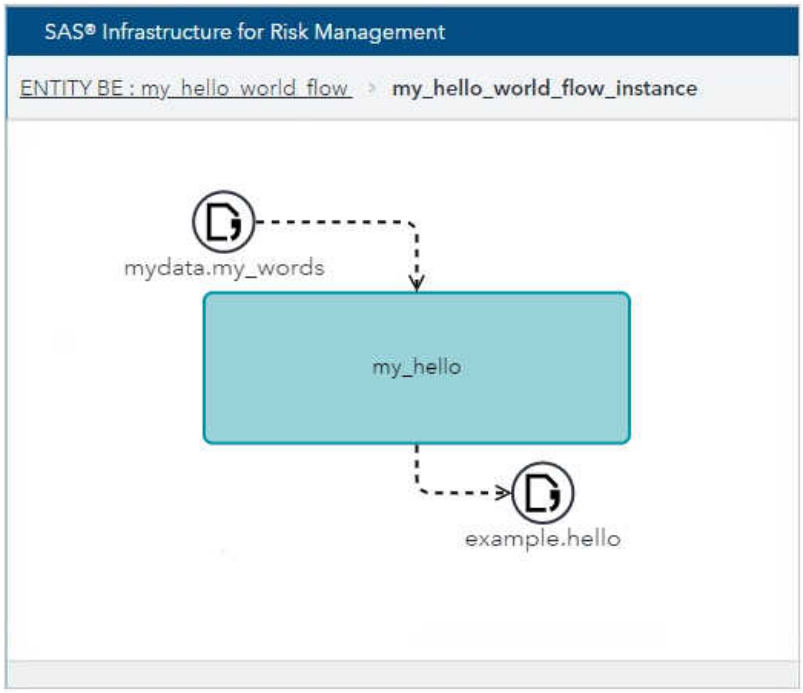
Example 1: my_hello Task with Parameter Substitution

This task uses parameter substitution to identify its input and output data objects. (See "Defining Task Inputs and Outputs Using Parameter Substitution".)

Here is a code sample that you can use to create a Python task that is named my_hello.

```
#   my_hello.py
##
##  \file
##  \brief The node gets a greeting from an input table
##
##  <b> Identified Inputs and outputs </b>
##  \param[in]  %in_words      Input word parts data
##  \param[out] %out_greeting  Extracted greeting
##
import logging
import pandas as pd

# input_output_df_dict {'in_words': df_in,
#                       'out_greeting': df_out }
def irm_run(input_output_df_dict):

    logger = logging.getLogger(__name__)
    logger.info("Entering execution of task hello_greeting.py with"
                " input in_words, output out_greeting")

    # Specify task input table as df_in
    df_in = input_output_df_dict['in_words']

    # Get subset (rows where word='Hello') from input table df_in
    df_out = pd.DataFrame({'GREETING': df_in['WORD']})
    df_out = df_out.loc[df_out['GREETING'] == 'HELLO']

    # Convert case of greeting
    df_out['GREETING'] = df_out['GREETING'].str.capitalize()

    # Put the subset in the task output table
    input_output_df_dict['out_greeting'] = df_out

    logger.info(
        "Exiting execution of task hello_greeting.py. "
        "\ndf_in:\n{} \ndf_out:\n{}"
            .format(df_in.head(), df_out.head()))
```

Here is how the newly created task appears in the SAS Infrastructure for Risk Management web application when you use it in a job flow:



Example 2: my_world Task with Parameter Substitution with Default Values

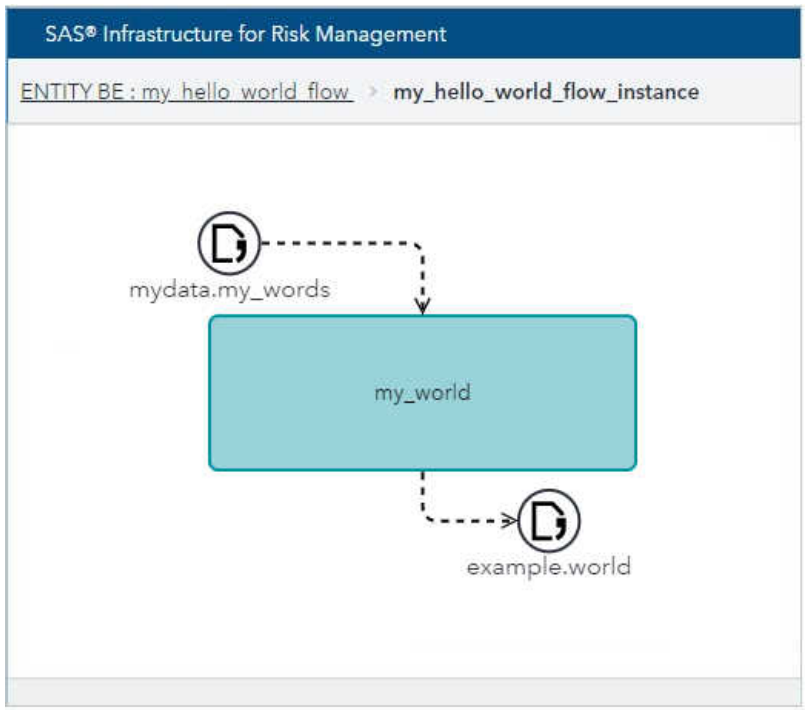
This task uses parameter substitution with default values to identify its input and output data objects. (see "Defining Task Inputs and Outputs Using Parameter Substitution with Default Value Specification".)

Here is a code sample that you can use to create a Python task that is named make_sentence:

```
#   my_world.py
##
##  \file
##  \brief The node gets the subject for a greeting from an
##         input table, specified default parameter values
##
##  <b> Identified Inputs and outputs </b>
##  \param[in]  %in_words=mydata.my_words.csv   Input word parts
##  \param[out] %out_subject=example.world.csv  Extracted subject
##
import logging
import pandas as pd

# input_output_df_dict {'in_words': df_in,
#                       'out_subject': df_out }
def irm_run(input_output_df_dict):

    logger = logging.getLogger(__name__)
    logger.info("Entering execution of task world_subject.py with"
                " input IN_WORDS, output OUT_SUBJECT")

    # Specify task input table as df_in
    df_in = input_output_df_dict['in_words']
```

```
# Get subset (rows where word='World') from input table df_in
df_out = pd.DataFrame({'SUBJECT': df_in['WORD']})
df_out = df_out.loc[df_out['SUBJECT'] == 'WORLD']

# Convert case of subject
df_out['SUBJECT'] = df_out['SUBJECT'].str.capitalize()

# Put the subset in the task output table
input_output_df_dict['out_subject'] = df_out

logger.info(
    "Exiting execution of task world_subject.py. "
    "\ndf_in:\n{} \ndf_out:\n{}"
        .format(df_in.head(), df_out.head()))
```

Here is how the newly created task appears in the SAS Infrastructure for Risk Management web application when you use it in a job flow:



Example 3: my_sentence Task with Inputs from Other Tasks

This task uses parameters to identify its input and output data objects.

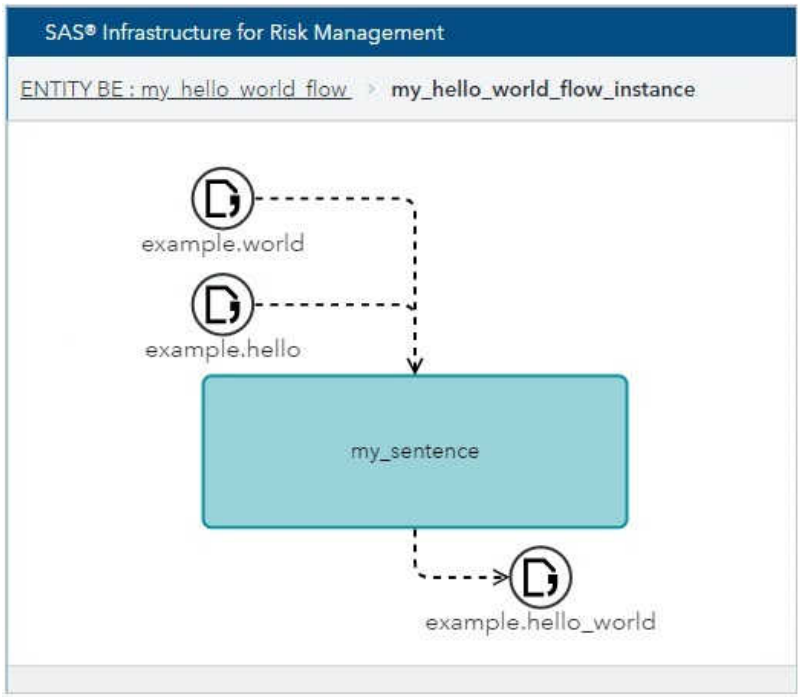
The input data objects in this task will not be found in the landing area (such as in Example 1 and Example 2), but instead is the output data objects that are produced by other tasks.

Note: The task header syntax is the same as when the input data objects are read from the landing area. When these tasks are combined in a job flow, SAS Infrastructure for Risk Management automatically determines the order and parallelism in which the tasks are executed based on the availability of the inputs.

Here is a code sample that creates a Python task that is named my_sentence:

```
#   my_sentence.py
##
##  \file
##  \brief The node combines a greeting and a subject into
##         a complete sentence, takes inputs that area output
##         from other tasks.
##
##  <b> Identified Inputs </b>
##  \param[in]  %in_greeting      Input greeting data
##  \param[in]  %in_subject       Input subject data
##
##  <b> Identified Inputs </b>
##  \param[out] %out_sentence     Output sentence with punctuation data
##
import logging
import pandas as pd

# input_output_df_dict {'in_greeting': df_in_01,
#                       'in_subject': df_in_02,
#                       'out_sentence': df_out }
def irm_run(input_output_df_dict):

    logger = logging.getLogger(__name__)
    logger.info("Entering execution of task make_sentence.py with"
                " inputs IN_GREETING IN_SUBJECT, output OUT_SUBJECT")

    # Specify task input tables
    df_in_01 = input_output_df_dict['in_greeting']
    df_in_02 = input_output_df_dict['in_subject']

    # Combine greeting and subject to make a sentence
    df_out = pd.DataFrame({'SENTENCE':
                           [df_in_01['GREETING'].values[0] + ', '
                           + df_in_02['SUBJECT'].values[0] + '!']})

    # Put the new sentence in the task output table
    input_output_df_dict['out_sentence'] = df_out

    logger.info(
        "Exiting execution of task make_sentence.py. "
        "\ndf_in_01:\n{} \ndf_in_02:\n{} \ndf_out:\n{}"
            .format(df_in_01.head(), df_in_02.head(), df_out.head()))
```

Here is how the task appears in the SAS Infrastructure for Risk Management web application when you use it in a job flow:


SAS® Infrastructure for Risk Management
ENTITY BE : my_hello_world_flow
my_hello_world_flow_instance
example.world
example.hello
my_sentence
example.hello_world

Chapter 5

Job Flows

About Job Flows

A job flow, or parallel program, is a way to organize tasks. The term, job flow, can be used to mean a job flow definition or a job flow instance. The exact meaning depends on the context in which it is used. (See "Key Concepts".)

To simplify the process of creating job flows, SAS Infrastructure for Risk Management provides a Python scripting client. The Python scripting client is a Python package that you use to create a job flow. You can then use the job flow with SAS Infrastructure for Risk Management.

Python Scripting Client Overview

A job flow script is a Python program that you write. This job flow script uses the SAS Infrastructure for Risk Management Python scripting client and the job flow tasks to create your parallel program.

SAS Infrastructure for Risk Management determines the order and parallelism in which the parallel program executes the tasks based on the availability of the task's specified inputs. Therefore, you can add the tasks to the scripts in any order.

Before you can use the scripting client, ensure that the following tasks have been completed:

1. The development environment has been configured .
2. The Python scripting client has been installed and set up .

To create a parallel program (job flow) using the Python scripting client:

1. Ensure that all needed content files (input data files, libnames.txt config file, task files, and so on) are available in your personal federated area.
2. Run the **`irmdev refresh`** command to ensure that the task APIs are current and reflect the current version of all available tasks.
3. Develop your job flow script (program) to build your job flow.

   A basic job flow script performs the following tasks:

   - imports the necessary Python packages and enables logging
   - loads your configuration information that was stored after running the **`irmdev init`** command
   - uses the configuration information to create an **`IrmContext`** object
   - defines your parallel program and assigns it a name
   - adds tasks to the parallel program
   - saves the job flow definition and runs the job flow to create an instance of the job flow
4. View the job flow (parallel program) in the SAS Infrastructure for Risk Management web application.

Here are the most commonly used scripting client components that you will use in your job flow scripts:

Table 5.1 *Common Scripting Client Components*

Component	Description
IrmConfig()	Loads the configuration information that was stored when you ran the **`irmdev init`** command. Creates the **`IrmConfig`** object from the irm.config file.
IrmContext()	Names and builds the job flow definitions. Serves as your entry point to the scripting client. Uses the loaded configuration information to create an **`IrmContext`** object. This object is used to interact with the scripting client to create your parallel program.
`your-IRM-context.build_jobflow`	Builds a job flow object from the current context. Uses **`.build_jobflow`** to name and build the job flow definition.

Component	Description
your-IRM-jobflow-object.add	Adds a task to the job flow definition.
your-IRM-jobflow-object.run	Executes the job flow definition and creates an instance of the job flow on the SAS Infrastructure for Risk Management server.
your-IRM-jobflow-object.save	Saves the job flow definition to the **jobflow** folder in your personal federated area.

For a complete list of the scripting client components and methods, including descriptions, syntax, and defaults, see Appendix 1, "Scripting Client Reference.

Example 1: Creating a Simple Job Flow with Your Three Tasks

Here is an example of how to create a simple job flow that contains the three tasks that you created in Chapter 4, "Tasks".

High-Level View of Creating the Job Flow

1. In PyCharm, select **File** ⇨ **New Python File**.

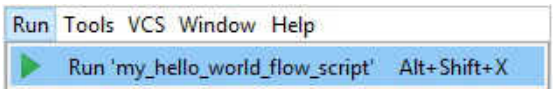
   Note: In this example, the job flow script is saved as my_hello_world_flow_script.py in the **client_scripts** folder of your personal federated area.

2. Enter your job flow script (program):

```
# ******************************************************************
#  NAME:      my_hello_world_flow_script.py                        *
#                                                                  *
#  PURPOSE:   My Python scripting client program (script) to create *
#             a simple job flow my_hello_world_flow containing the  *
#             3 tasks created in my personal federated area nodes   *
# ******************************************************************
from __future__ import absolute_import, division, print_function,\
    unicode_literals

from irm.context import IrmContext
from irm.config import IrmConfig

# import python.nodes as mytasks
from python.nodes import nodes as mytasks

import logging

# Enable Python logging at INFO level
logging.getLogger().setLevel(logging.INFO)
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s'
                           ' %(name)s - %(message)s')
```

```
# Step 1 Scripting Client Initialization - Build config and context
# Load configuration information that was stored by irmdev init
conf = IrmConfig()

# Use the configuration information to create context object
ic = IrmContext(conf)

# Step 2  Name and build the job flow definition
my_example_flow_ref = ic.build_jobflow("my_hello_world_flow")

# Step 3 Add my 3 tasks to the job flow definition
my_example_flow_ref.add(mytasks.my_hello(substitution_dict={
    "%in_words":"mydata.my_words.csv",
    "%out_greeting":"example.hello.csv",
}),name="my_hello")

my_example_flow_ref.add(mytasks.my_world(),name="my_world")

my_example_flow_ref.add(mytasks.my_sentence(substitution_dict={
    "%in_greeting":"example.hello.csv",
    "%in_subject":"example.world.csv",
    "%out_sentence":"example.hello_world.csv",
}),name="my_sentence")

# Step 4 Trigger to save job flow
my_example_flow_ref.save()

# Step 5 Create an instance and run it
my_example_flow_ref.run()
```

3. Select **Run** ⇨ **Run 'my_hello_world_flow_script'**.



Detailed View of Creating the Job Flow

Here is a detailed description of each step in the job flow script that you just created. The examples do not display all the options or arguments that are available for a given scripting client component. Default settings are used.

1. Initialize the scripting client.

```
# Load configuration information that was stored by irmdev init
conf = IrmConfig()
# Use the configuration information to create context object
ic = IrmContext(conf)
```

   Use **`IrmConfig()`** and **`IrmContext()`** to initialize the scripting client.

   `IrmConfig` loads the configuration information that was stored when you executed the **`irmdev init`** command. **`IrmContext`** then uses that configuration information to create a context object that will be used to interact with the Python scripting client throughout the remaining job flow script.

2. Name and build the job flow definition.

```
my_example_flow_ref = ic.build_jobflow("my_hello_world_flow")
```

Use the scripting client **`build_jobflow`** method to name and build a job flow definition.

In this example, the job flow definition is named my_hello_world_flow. This job flow definition is referenced by the object name my_example_flow_ref throughout the remaining job flow script.

3. Add the three tasks to the job flow definition.

```
my_example_flow_ref.add(mytasks.my_hello(substitution_dict={
    "%in_words":"mydata.my_words.csv",
    "%out_greeting":"example.hello.csv",
}),name="my_hello")

my_example_flow_ref.add(mytasks.my_world(), name="my_world")

my_example_flow_ref.add(mytasks.my_sentence(substitution_dict={
    "%in_greeting":"example.hello.csv",
    "%in_subject":"example.world.csv",
    "%out_sentence":"example.hello_world.csv",
}),name="my_sentence")
```

You can add your task macros to the job flow script in any order.

The syntax of each task follows the same basic structure, but it might vary according to how the inputs and outputs are named based on the task header.

For example, here is an explanation of the syntax that is used for the my_hello task in the job flow in this job flow script:

```
a. my_example_flow_ref.add(
   b. mytasks.my_hello(
          substitution_dict={
       c.   "%in_words":"mydata.my_words.csv",
            "%out_greeting":"example.hello.csv",
          }
   d. ), name="my_hello")
```

a. Use the scripting client **`add`** method to add a task to the job flow definition.

b. Specify the name of the task that you are adding to the job flow.

   Here, the my_hello task is in the **`python/nodes`** folder of the personal federated area, which was imported as mytasks earlier in this job flow script:

```
# import python.nodes as mytasks
from python.nodes import nodes as mytasks
```

c. Use the substitution_dict dictionary to specify the task inputs and outputs based on what is defined in the task file header.

   The my_hello task header uses parameter substitution to specify the task input and output files. The following table shows how the parameter name is paired with its value, which is the full three-level name of the file:

Parameter Name (from the Task Header)	User-Supplied Value
%in_words	mydata.my_words.csv
%in_words	example.hello.csv

There are several ways that you can specify the inputs and outputs in the task file header. These different specifications determine the syntax that you use in the job flow script with the scripting client **add** method:

Table 5.2 *Data Input and Output Specifications*

Task File Header Specification	Job Flow Script add Syntax

Parameter substitution without a default table name specified:
`## \param[in] %in_words`
	You must specify the name in the script using substitution_dict dictionary name value pairs:
	`my_example_flow_ref.add(`
	`  mytasks.my_hello(substitution_dict={`
	`  "%in_words":"mylib.myword.csv",`
	`})`

Parameter substitution with a default table name specified:
`## \param[in] %in_words=mylib.myword.csv`
	You can include the substitution_dict dictionary with the file name in the script, but it is not required.
	To accept the default table name, do not specify the parameter.
	An example of this syntax is shown in the Example 1 on page 45 job flow script where the my_world tasks is added to the job flow.
	`my_example_flow_ref.add(`
	`  mytasks.my_world(),`
	`name="my_world")`
	To override the default file name, specify the substitution_dict dictionary with the name of the override file:
	`my_example_flow_ref.add(`
	`  mytasks.my_hello(substitution_dict={`
	`  "%in_words":"newlib.override.csv",`
	`})`

No parameter substitution used: `## \param[in] mydata.mywords.csv`	The table name is fixed in the task header. Therefore, it cannot be changed or substituted in the job flow script.

d. (Optional) Specify a user-assigned task name.

It might be useful to assign a descriptive task name to identify tasks when you are using the same task multiple times with different inputs. If you do not specify a descriptive task name, the task name defaults to the name of the task.

4. Save the job flow definition to the SAS Infrastructure for Risk Management server.

```
my_example_flow_ref.save()
```

The scripting client **save** method saves the my_hello_world_flow job flow definition to the SAS Infrastructure for Risk Management server.

Once saved, you can use the job flow definition to create job flow instances in SAS Infrastructure for Risk Management.

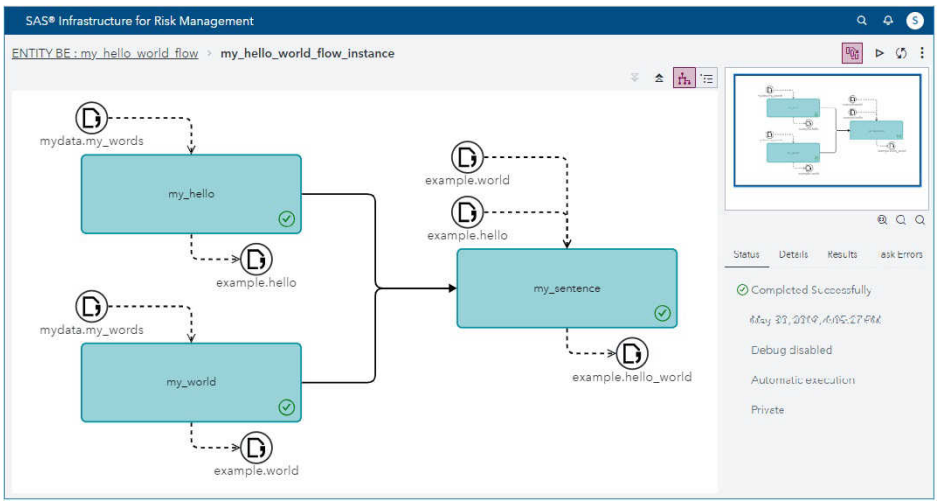
5. Create and execute a job flow instance.

```
my_example_flow_ref.run()
```

The scripting client **run** method runs your generated parallel program to create an instance of the job flow. It also provides a URL link that you can click (or copy to a web browser) to view the job flow instance in the SAS Infrastructure for Risk Management web application.

View the Job Flow Instance

After creating the job flow instance, you can view it from the SAS Infrastructure for Risk Management web application. Here is how the job flow instance appears in the SAS Infrastructure for Risk Management web application.



Example 2: Creating a Job Flow That Contains a Subflow

Here is an example of a job flow script that creates a job flow definition that performs the same function as the simple job flow in Example 1. However, in this job flow definition, the my_hello task and the my_world task are part of a subflow that is named words_subflow. The words_subflow is then added to a top-level, or parent, job flow.

High-Level View of Creating the Job Flow

1. In PyCharm, select **File ⇨ New Python File**

   Note: In this example, the job flow script is saved as my_hello_world_with_subflow_script.py in the **`client_scripts`** folder of the personal federated area.

2. Enter your job flow script (scripting client program). The job flow script is similar to Example 1. It creates a job flow containing the three example tasks, but this example does not add all three tasks to the top-level flow.

   In this example, you create a subflow (words_subflow) that contains two tasks (my_hello, my_world). That subflow is then added to the top-level, or parent, job flow along with the my_sentence task:

```
# ******************************************************************
#  NAME:      my_hello_world_with_subflow_script.py                *
#                                                                  *
#  PURPOSE:   My Python scripting client program (script) to create *
#             my_hello_world_sub_flow with a subflow and 1 task     *
# ******************************************************************
from __future__ import absolute_import, division, print_function,\
    unicode_literals

from irm.context import IrmContext
from irm.config import IrmConfig

# import python.nodes as mytasks
from python.nodes import nodes as mytasks

import logging

# Enable Python logging at INFO level
logging.getLogger().setLevel(logging.INFO)
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s'
                           ' %(name)s - %(message)s')

# Step 1 Scripting Client Initialization - Build config and context
# Load configuration information that was stored by irmdev init
conf = IrmConfig()

# Use the configuration information to create context object
ic = IrmContext(conf)

# Step 2  Name and build the TOP-LEVEL job flow definition
my_example_flow_ref = ic.build_jobflow("my_hello_world_sub_and_flow")

# Step 3  Name and build the WORDS_SUBFLOW subflow
my_subflow_ref = ic.build_jobflow("words_subflow")

# Step 4  Add the subflow to the TOP_LEVEL job flow
my_example_flow_ref.add_sub_flow(my_subflow_ref)

# Step 5  Add tasks (my_hello, my_world) to words_subflow SUBFLOW
```

```
my_subflow_ref.add(mytasks.my_hello(substitution_dict={
    "%in_words":"mydata.my_words.csv",
    "%out_greeting":"example.hello.csv",
}),name="my_hello")

my_subflow_ref.add(mytasks.my_world(substitution_dict={
    "%in_words":"mydata.my_words.csv",
    "%out_subject":"example.world.csv",
}),name="my_world")

# Step 6  Add task (my_sentence) to the TOP_LEVEL job flow
my_example_flow_ref.add(mytasks.my_sentence(substitution_dict={
    "%in_greeting":"example.hello.csv",
    "%in_subject":"example.world.csv",
    "%out_sentence":"example.hello_world.csv",
}),name="my_sentence")

# Step 7 Trigger to save job flow
my_example_flow_ref.save()

# Step 8 Create an instance and run it
my_example_flow_ref.run()
```

3. Select **Run** ⇨ **Run ‘my_hello_word_with_subflow_script’**.

Detailed View of Creating the Job Flow

Here is a detailed description of steps 3 – 6 in the job flow script that you just created. The remainder of the script is the same as that which is explained in the simple job flow script in Example 1.

1. (Step 3) Name and build words_subflow job flow definition.

   ```
   my_subflow_ref = ic.build_jobflow("words_subflow")
   ```

   In this example, the job flow definition of the new subflow is named words_subflow . This job flow definition is referenced by the object name my_subflow_ref throughout the remaining job flow script.

2. (Step 4) Add the subflow to the top-level job flow.

   ```
   my_example_flow_ref.add_sub_flow(my_subflow_ref)
   ```

   The top-level job flow (named my_hello_world_sub_and_flow with reference my_example_flow_ref) was created in Step 2 and the subflow (named words_subflow with reference my_subflow_ref) was created in Step 3.

   Use the scripting **`add_sub_flow`** method to add the subflow to the top-level flow. This step designates words_subflow as a subflow (a job flow contained within a job flow) of hello_world_sub_and_flow.

3. (Step 5) Add tasks to the subflow.

   ```
   my_subflow_ref.add(mytasks.my_hello(substitution_dict={
       "%in_words":"mydata.my_words.csv",
       "%out_greeting":"example.hello.csv",
   }),name="my_hello")

   my_subflow_ref.add(mytasks.my_world(substitution_dict={
       "%in_words":"mydata.my_words.csv",
   ```

```
    "%out_subject":"example.world.csv",
})),name="my_world")
```

The two tasks (my_hello and my_world) are added to the subflow (sentence_subflow with the reference name example_subflow_ref).

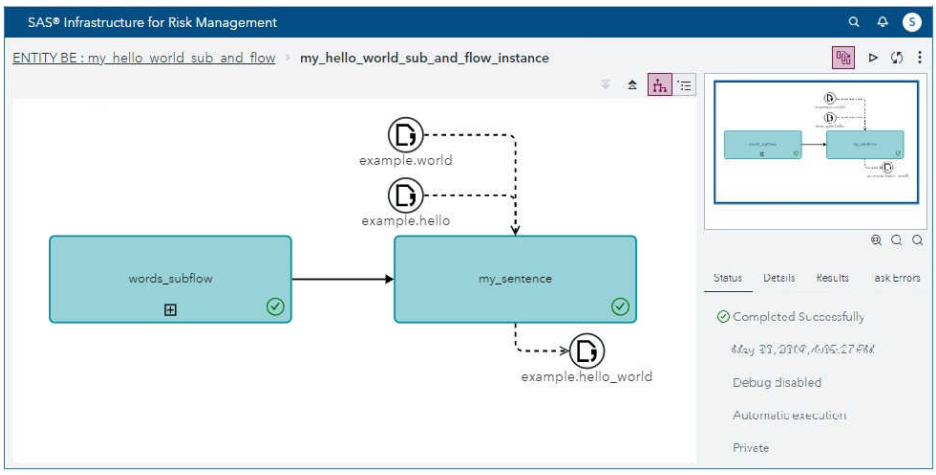
4. (Step 6) Add the remaining task (my_sentence) to the top-level job flow.

```
my_example_flow_ref.add(mytasks.my_sentence(substitution_dict={
    "%in_greeting":"example.hello.csv",
    "%in_subject":"example.world.csv",
    "%out_sentence":"example.hello_world.csv",
})),name="my_sentence")
```

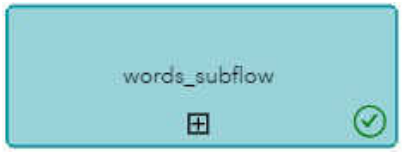
The single task (my_sentence) is added to the top-level job flow (my_hello_world_sub_and_flow job flow with the reference name my_example_flow_ref).

View the Job Flow Instance

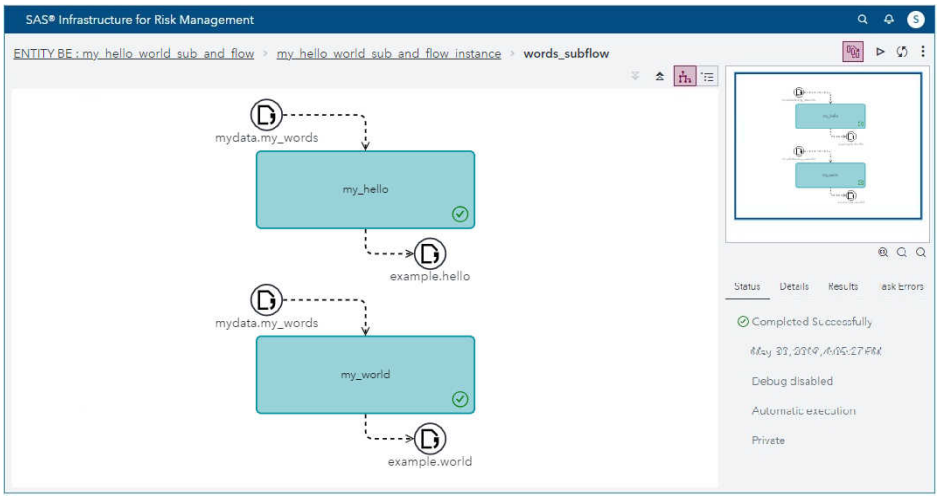
After creating the job flow instance, you can view it from the SAS Infrastructure for Risk Management web application. Here is how the job flow instance appears in the SAS Infrastructure for Risk Management web application.



The subflow (words_subflow) is denoted in the SAS Infrastructure for Risk Management web application by a plus (+) sign to indicate that it is not a single task, but a subflow.



To expand the subflow and view the tasks that it contains, double-click the subflow icon in the SAS Infrastructure for Risk Management web application:



Example 3: Creating a Job Flow That Simplifies the View of a Complex Job Flow

You can configure the inputs and outputs for subflows to display in the job flow diagram in the SAS Infrastructure for Risk Management web application. Here is an example of a job flow script that creates a job flow that performs the same function as the job flow with a subflow in Example 2. However, in this job flow definition, the job flow instance diagram in the SAS Infrastructure for Risk Management web application displays a visualization of the inputs and outputs that are associated with a subflow from the top-level (parent flow) view.

High-Level View of Creating the Job Flow

1. In PyCharm, select **File** ⇨ **New Python File**
2. Enter your job flow script (program), which consists of scripting client macros and your tasks. The program is the same as in Example 2, but with the addition of step 5b.

```
# ******************************************************************
#  NAME:      my_hello_world_with_visual_subflow_script.py          *
#                                                                   *
#  PURPOSE:   My Python scripting client program (script) to create *
#             my_hello_world_sub_flow with a subflow and 1 task     *
# ******************************************************************
from __future__ import absolute_import, division, print_function,\
    unicode_literals

from irm.context import IrmContext
from irm.config import IrmConfig

# import python.nodes as mytasks
from python.nodes import nodes as mytasks

import logging
```

```
# Enable Python logging at INFO level
logging.getLogger().setLevel(logging.INFO)
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s'
                           ' %(name)s - %(message)s')

# Step 1 Scripting Client Initialization - Build config and context
# Load configuration information that was stored by irmdev init
conf = IrmConfig()

# Use the configuration information to create context object
ic = IrmContext(conf)

# Step 2  Name and build the TOP-LEVEL job flow definition
my_example_flow_ref = ic.build_jobflow("my_hello_world_visual_flow")

# Step 3  Name and build the WORDS_SUBFLOW subflow
my_subflow_ref = ic.build_jobflow("words_subflow")

# Step 4  Add the subflow to the TOP_LEVEL job flow
my_example_flow_ref.add_sub_flow(my_subflow_ref)

# Step 5  Add tasks (my_hello, my_world) to words_subflow SUBFLOW
my_subflow_ref.add(mytasks.my_hello(substitution_dict={
    "%in_words":"mydata.my_words.csv",
    "%out_greeting":"example.hello.csv",
}),name="my_hello")

my_subflow_ref.add(mytasks.my_world(substitution_dict={
    "%in_words":"mydata.my_words.csv",
    "%out_subject":"example.world.csv",
}),name="my_world")

# Step 5b Add visual input and output to the words_subflow SUBFLOW
my_subflow_ref.add_visual_input_table_array(
    ["mydata.my_words.csv"])

my_subflow_ref.add_visual_output_table_array(
    ["example.hello.csv",
     "example.world.csv"])

# Step 6  Add task (my_sentence) to the TOP_LEVEL job flow
my_example_flow_ref.add(mytasks.my_sentence(substitution_dict={
    "%in_greeting":"example.hello.csv",
    "%in_subject":"example.world.csv",
    "%out_sentence":"example.hello_world.csv",
}),name="my_sentence")

# Step 7 Trigger to save job flow
my_example_flow_ref.save()

# Step 8 Create an instance and run it
my_example_flow_ref.run()
```

3. Select **Run ⇨ Run 'my_hello_word_with_subflow_script'**.

Detailed View of Creating the Job Flow

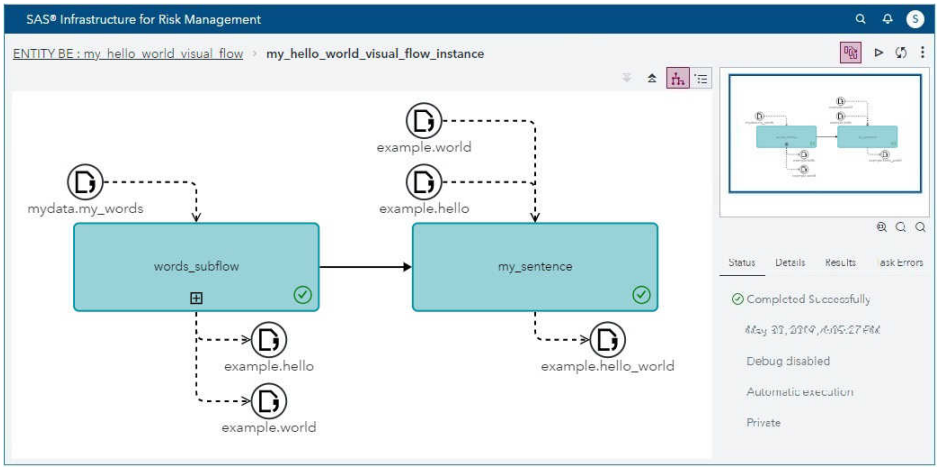
Here is a detailed description of the scripting client methods that you used in Step 5b to make the subflow's inputs and outputs visible from the top-level flow view.

Use the scripting client **`add_visual_input_table_array`** method to specify a list of inputs to display. Use the scripting client **`add_visual_output_table_array`** method to specify a list of outputs to display.

```
# Step 5b Add visual input and output to the words_subflow SUBFLOW
my_subflow_ref.add_visual_input_table_array(
    ["mydata.my_words.csv"])

my_subflow_ref.add_visual_output_table_array(
    ["example.hello.csv",
     "example.world.csv"])
```

In the scripting client **`add_visual_input_table_array`** and **`add_visual_output_table_array`** methods, you list the inputs and outputs that you want to display for the subflow task in the SAS Infrastructure for Risk Management web application. Each input or output in the array list is delimited by a comma (,).

Here is an example of how the job flow appears in the SAS Infrastructure for Risk Management web application after you create the job flow instance with the **`add visual data`** methods. These methods list the subflow inputs (mydata.my_words) and the subflow outputs (example.hello and example.world):



Chapter 6

Partitions

About Partitioning

Partitioning is an important functionality that SAS Infrastructure for Risk Management provides. Partitioning enables you to separate large data files into smaller units of data and dynamically distribute the calculation of these smaller units of data across multiple cores. Using partitions makes the calculation of large tables much faster and efficient.

SAS Infrastructure for Risk Management supports the following partition methods:

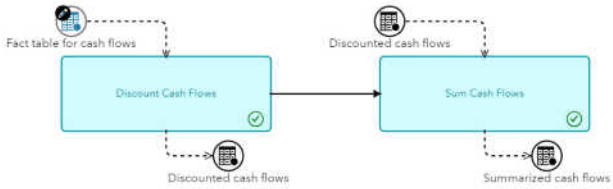
- By Number (ByN) — Data is partitioned by a user-defined number of partitions.
- By Variable (ByVar) — Data is partitioned by user-defined BY variables.

Using Partitions in a Job Flow

About Using Partitions in a Job Flow

The following sections illustrate how to use the ByN partition method and the ByVar partition method in a job flow.

Here is a job flow diagram view of a simple single-threaded job flow with two tasks — no partitioning involved. In this job flow, the first task discounts cash flows and the second task calculates the sum total of the discounted cash flow values.



The steps to create a partition job flow might differ between the two methods and the number of computational / analysis tasks in your job flow. Regardless of the partition method that you use, your job flow must complete the following four tasks:

1. Calculate the cardinality task
2. Partition the data task
3. Computational task.
4. Recombine task.

The partition task files used in the following sections are located in the sample Python federated area that is installed with SAS Infrastructure for Risk Management. In addition, two sample job flow scripts (simple_byn_partition_flow.py and simple_byvar_partition_flow.py) are also provided that you can use as a template for your job flow script.

To access the task files, navigate to **`\SASIRM\fa.sample.3.6.py\source\python\nodes`**. To access the sample job flow scripts, navigate to **`\SASIRM\fa.sample.3.6.py\client_scripts\sample_python`**.

Step 1: Calculate the Cardinality Task

About the Cardinality Task

This task calculates the cardinality of input data objects, which is the number of threads to create in partition tasks.

The cardinality task calculates the cardinality of input data based on a user-defined value that is specified in an input configuration data set. The output of this task is a cardinality table. The cardinality table contains one row and one column (MAX_RANK_NO). SAS Infrastructure for Risk Management uses the value of MAX_RANK_NO to determine the number of partitions to dynamically create in the partition tasks.

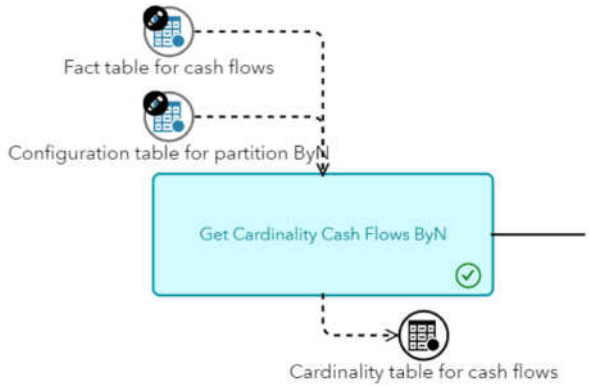
Here is an example of the output cardinality table:

VIEWTABLE: Mk_card.Card_filtered_zero_coupon_bond

	MAX_RANK_NO
1	32

ByN Partition Method

With ByN partitioning, the calculate the cardinality task calculates the cardinality of input data objects based on a user-defined number that is specified in an input configuration data set. If the number of partitions is not specified, the task sets the cardinality to 1.



Here are the inputs and outputs of the irm_get_cardinality_byn.py task:

Table 6.1 *Input Data Objects*

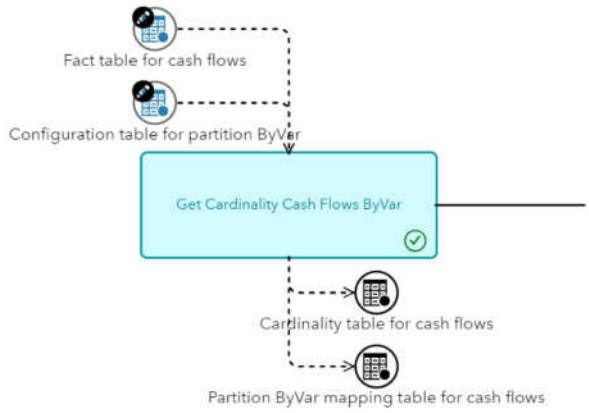
Parameters	Description
`[in] %IN_BYN_CONF=IRM_CFG.byn_configuration.csv`	Configuration data set that contains the user-defined number of partitions.
`[in] %IN_DS_PART_BYN=STAGING.cashflows.csv`	Data set to be partitioned.

Table 6.2 *Output Data Object*

Parameter	Description
`[out] %OUT_CARD_DS_BYN=MK_CARD.card_cashflows_byn.csv`	Cardinality table.

ByVar Partition Method

With the ByVar partition method, this task calculates the cardinality of input data objects based on user-defined BY variables that are specified in an input configuration file. If a BY variable is not specified, or no valid BY variable is provided, the task sets the cardinality to 1 and creates an empty ByVar mapping table.



Here are the inputs and outputs of the irm_get_cardinality_byvar.py task:

Table 6.3 *Input Data Objects*

Parameters	Description
`[in] %IN_BYVAR_CONF=IRM_CFG.byvar_configuration.csv`	Configuration data set that contains the user-defined BY variable or variables.
`[in] %IN_DS_PART_BYVAR=STAGING.cashflows.csv`	Data set to be partitioned.

Table 6.4 *Output Data Objects*

Parameters	Description
`[out] %OUT_CARD_DS_BYVAR=MK_CARD.card_filtered_cashflows.csv`	Cardinality table.
`[out] %OUT_CARD_MAP=MK_CARD.part_cashflow_mapping.csv`	BY variable(s) mapping table.

Step 2: Partition the Data Task

About the Partition Task

When the SAS Infrastructure for Risk Management server identifies a partition task, it reads the output cardinality table to create MAX_RANK_NO threads. In every thread, the partition task slices the data set using the ByN method or using the ByVar method.

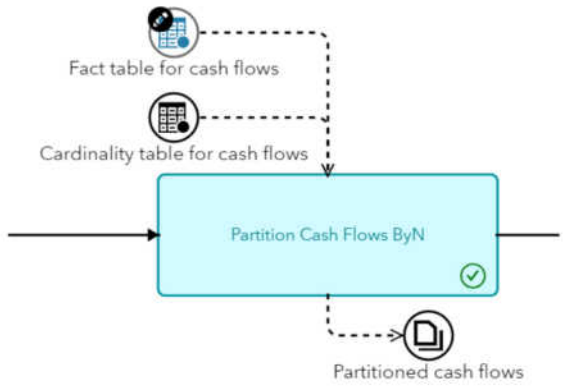
At minimum, the partition task will have the following inputs and output:

- Input 1: the output cardinality table created by Step 1.
- Input 2: the table to partition.

- Input 3: the partition mapping table created by Step 1. (ByVar partition only)
- Output: the partitioned results data.

ByN Partition Method

For ByN partitioning, this task partitions the input data based on the ByN cardinality table.



Here are the inputs and outputs of the irm_partition_bn.py task:

Table 6.5 *Input Data Objects*

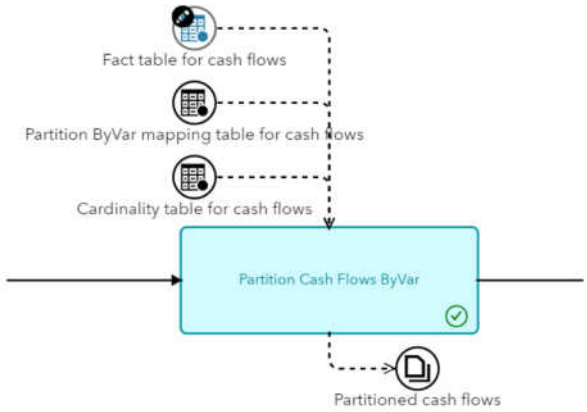
Parameters	Description
`[in] %DS_TO_PART=staging.cashflows.csv`	Data to be partitioned.
`[in] %CARDINALITY_DS=mk_card.card_cashflows_byn.csv`	Cardinality data that indicates how many partitions to create.

Table 6.6 *Output Data Object*

Parameter	Description
`[out] %PART_DS=p_stage.cashflows.csv Output data after partition.`	Partitioned data.

ByVar Partition Method

For ByVar partitioning, this task partitions the input data based on the ByVar cardinality table and mapping table.



Here are the inputs and outputs of the irm_partition_byvar.py task:

Table 6.7 *Input Data Objects*

Parameters	Description
`[in] %IN_PART_DS=STAGING.cashflows.csv`	Data to be partitioned.
`[in] %IN_CARD_MAP=MK_CARD.part_cashflow_mapping.csv`	ByVar mapping table that contains names and values of the By variables.
`[in] %IN_CARD_DATA=MK_CARD.card_filtered_cashflows.csv`	Cardinality data that indicates how many partitions to create.

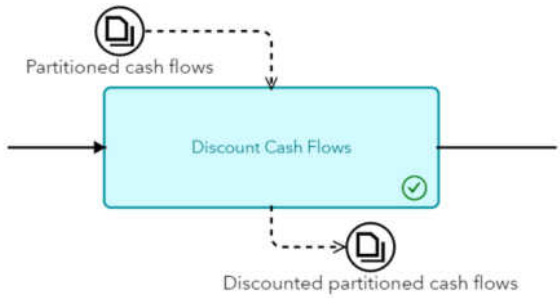
Table 6.8 *Output Data Object*

Parameter	Description
`[out] %PART_DS=p_stage.cashflows.csv`	Partitioned data.

Step 3: Compute the Partitioned Data Task

The computational task is a standard computing task except for the following:

- It must have partitioned input data and output data.
- When there is more than one input data objects, some can be non-partitioned. In this case, all instances of the computational tasks receive the same input.
- The partitioned outputs must have different library names.



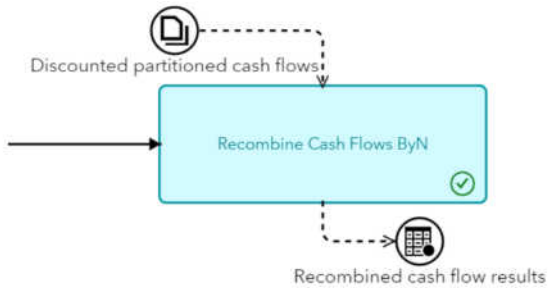
Step 4: Recombine the Results Task

About the Recombine Task

The recombine task recombines partitioned data into one non-partitioned data output. The recombine task is a single-threaded task.

ByN Partition Method

For ByN partitioning, this task recombines a set of ByN partitioned input data into one non-partitioned data output.



Here are the inputs and outputs of the irm_recombine_byn.py task:

Table 6.9 *Input Data Object*

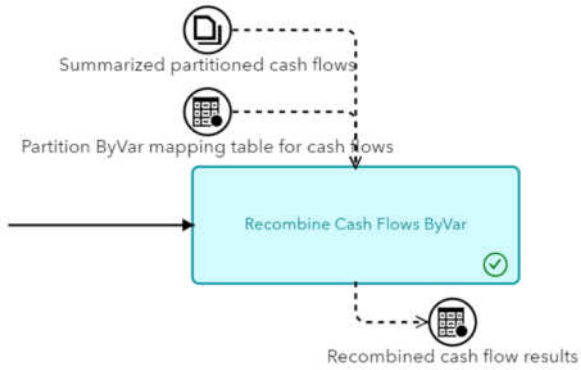
Parameter	Description
`[in] %TABLEIN=P_MK_CF.cashflows.csv`	Dictionary of identical structure data frames that represent partitioned input data.

Table 6.10 *Output Data Object*

Parameter	Description
`%TABLEOUT=R_MK_CF.cashflows_result.csv`	Data frame obtained by appending all input data frames. The output table has the same structure as an input partitioned data frame.

ByVar Partition Method

For ByVar partitioning, this task recombines a set of ByVar partitioned input data into one non-partitioned data output. During the recombine process, the ByVar variables are added back to the output data.



Here are the inputs and outputs of the irm_recombine_byvar.py task:

Table 6.11 *Input Data Objects*

Parameters	Description
`[in] %TABLEIN=P_MK_CF.cashflows.csv`	Dictionary of identical structure data frames that represent partitioned input data.
`[in] %IN_CARD_MAP_DS=MK_CARD.part_cashflow_mapping.csv`	ByVar mapping table that contains names and values of the By variables.

Table 6.12 *Output Data Object*

Parameter	Description
`[out] %TABLE_OUT_NM=R_MK_CF.cashflows_result.csv`	Data frame obtained by appending all input data frames. The output table has the same structure as an input partitioned data frame.

Chapter 7

Accessing SAS Infrastructure for Risk Management Information Using WebDAV

About Using WebDAV to Access Your SAS Infrastructure for Risk Management Information

SAS Infrastructure for Risk Management uses Web Distributed Authoring and Versioning (WebDAV) to provide an easy way for users to access the following types of SAS Infrastructure for Risk Management information:

- job flow definition files
- input and output files
- the execution status of job flow instances and sub-flow instances
- the execution status of tasks
- the navigation hierarchy of the data

Before using WebDAV to access SAS Infrastructure for Risk Management data, note the following restrictions:

- The scope of data that a user can access is controlled by permissions that are associated with the user's logon credentials.
- The content of a flow instance that has not been shared is accessible only to the owner of the flow. If the instance is shared, the content is accessible to users who have access to the business entities of the flow instance.
- The contents of the SAS Infrastructure for Risk Management WebDAV servlet are Read-Only and cannot be deleted.

SAS Infrastructure for Risk Management users can access the data store in a centralized location on a remote server using WebDAV by using either of the following methods:

- using a WebDAV URL in a browser
- using a WebDAV drive that is mapped to your computer

Using a WebDAV URL in a Browser

You can access your SAS Infrastructure for Risk Management information using a WebDAV URL in your browser.

Here is the general form of the WebDAV URL that you enter in your browser:

```
protocol//server_name:port/SASIRMServer/irmwebdav/userid
```

In the following example, you (sasdemo) create the hello_world job flow instance. (See Chapter 2, “Create Your First Parallel Program in 60 Seconds”.)

To view the job flow information using the WebDAV URL in the browser:

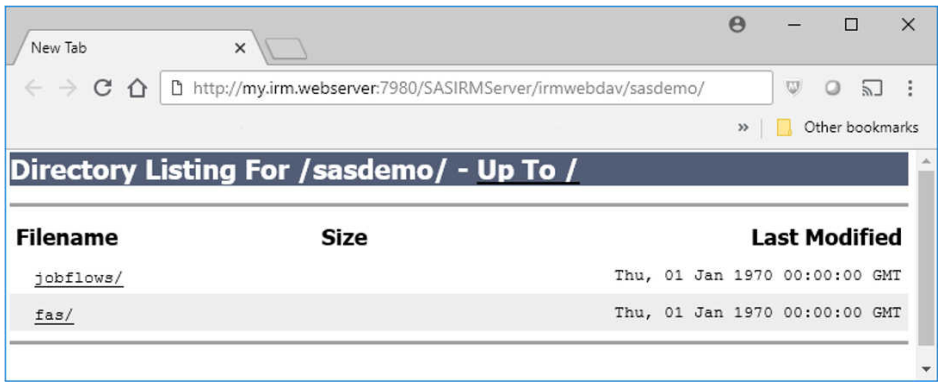
1. Enter the WebDAV URL in your browser.

   In this example, the URL for the SAS Infrastructure for Risk Management WebDAV server is:

   ```
   http://my.irm.webserver:7980/SASIRMServer/irmwebdav/sasdemo/
   ```

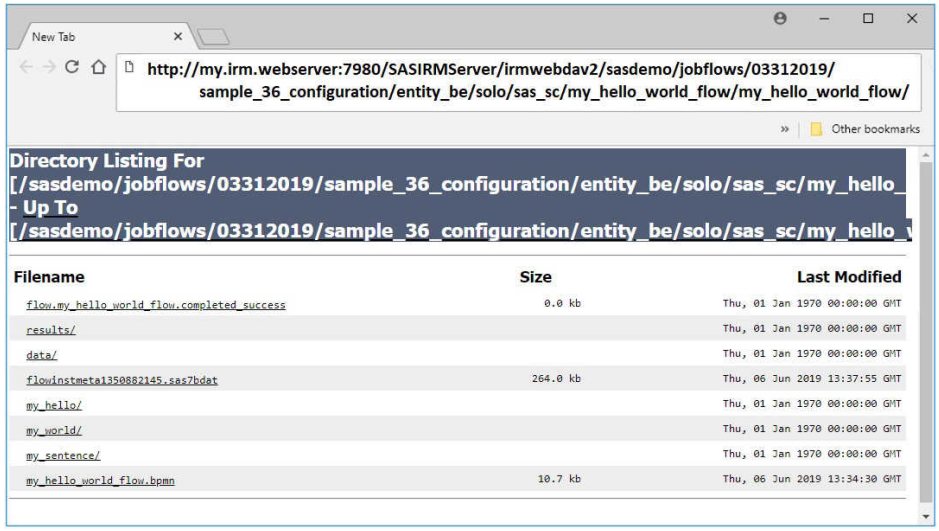
   Note: You will be prompted to log on to the SAS Infrastructure for Risk Management server.

   Here is how the WebDAV directories appear once you have logged on to the server:

2. As needed, drill down to find information. Click **Up To** to drill up the directory tree.

   If any instances have been created in your personal federated area, you can view the logs, status, data, and other files by drilling down through the job flow directory. For example, to see the hello_world job flow instance log and other information, follow the path: **jobflows/03312019/sample_36_configuration/entity_be/solo/sas_sc/hello_world_flow/hello_world**.



Mapping a WebDAV Drive to Your Computer

You can access your SAS Infrastructure for Risk Management information using a network drive on your computer that is mapped to the WebDAV server.

To access the data from a WebDAV drive that is mapped to your computer:

1. Map the SAS Infrastructure for Risk Management WebDAV servlet drive to your computer. For information about mapping the WebDAV servlet drive to your computer, refer to the documentation for your operating system.

   Here is the general form of the network drive folder name:

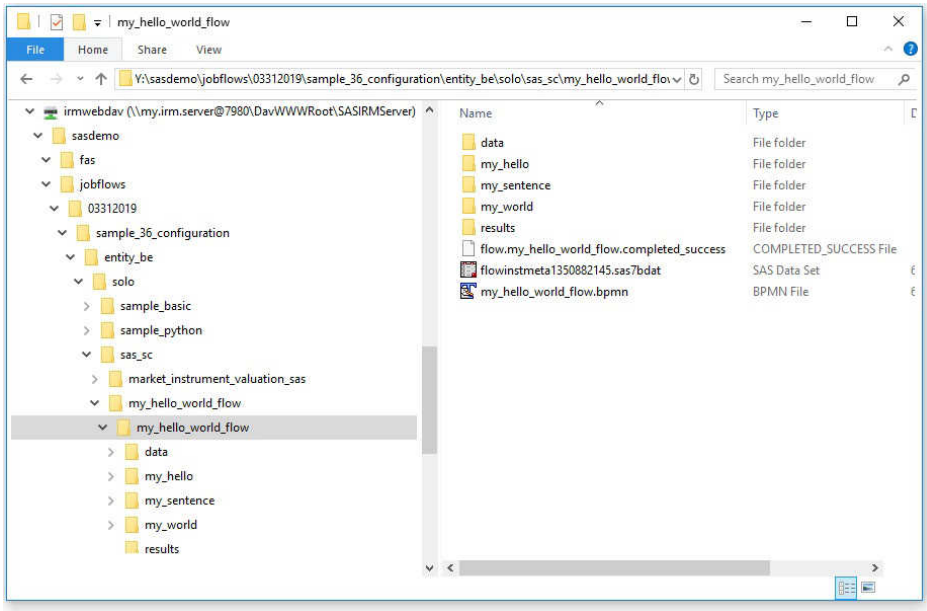
   `protocol//server_name:port/SASIRMServer/irmwebdav`

   Note: You will be prompted to log on to the SAS Infrastructure for Risk Management server.

2. Access the drive on your local system and click your user ID to navigate to the data that you want to access.

In the following example, you (sasdemo) create the hello_world job flow instance. (See Chapter 2, "Create Your First Parallel Program in 60 Seconds".) You map a network drive to the WebDAV server. The network drive folder name used in this example is **`http://my.irm.webserver:7980/SASIRMServer/irmwebdav/`**.

After you map the network drive, you can view the job flow instance logs, tasks, data, and other information:



In addition to viewing the files, you can access them directly for reporting or other tasks.

Note: If your SAS Infrastructure for Risk Management server is running HTTP and you are running Windows 10 on your local machine, you might have to modify the Windows Registry to allow Basic authentication to WebDAV over an HTTP connection before you can map a network drive to the SAS Infrastructure for Risk Management WebDAV servlet . For more information, refer to the Microsoft Windows documentation.

Chapter 8

Additional Programming Features

Deleting a Job Flow Definition

To delete a job flow definition from your personal federated area, use the **`delete_jobflow_definition`** method.

For details about the **`delete_jobflow_definition`** method, see "IrmContext Methods".

Using Generic Libraries to Bypass Live ETL

SAS Infrastructure for Risk Management supports generic library mapping definitions. Generic library mapping definitions enable access to third-party data that is located outside of a SAS Infrastructure for Risk Management federated area. For example, this data might be located in a relational database management system, Hadoop, CAS, and so on.

Generic libraries provide an easy method to bypass live ETL or to upload data objects into job flow instances. For detailed information about generic library mapping definitions, see "Generic Library Mapping" in *SAS Infrastructure for Risk Management: Administrator's Guide*.

Although the use of a generic library is convenient and useful during development, they are not recommended for deployments on production systems. Generic libraries break an essential SAS Infrastructure for Risk Management design premise, which is that SAS Infrastructure for Risk Management controls the landing area (%LA), the input area (%IA), and the persistent area (%PA).

This design premise is essential in guaranteeing Atomicity, Consistency, Isolation, Durability (ACID) transactions during the execution of a job flow instance. Therefore, if you use generic libraries for the design of your federated area, you must provide manual

or automated workflows to work around the lack of transactional integrity for your job flows.

Here is a partial list of the challenges that you or your users could experience if you use generic libraries in your job flows:

- Correctness
  - You must consider that the input data at the beginning of the execution of the flow is not guaranteed to be immutable for the duration of the execution. For example, if MK_OPT.positions.sas7bdat is the input data set to two tasks, the two tasks could read different values for the content of MK_OPT.positions.sas7bdat when they execute. With generic libraries, SAS Infrastructure for Risk Management cannot provide isolation or consistency.
  - For downloads, there is no guarantee that the data being downloaded is the result of the computation of the job flow instance. With generic libraries, SAS Infrastructure for Risk Management cannot guarantee durability.
  - To manually provide isolation during flow execution, consider that care must be exercised to avoid sharing data objects across flow instances.
  - When you use generic libraries, data conversions are automatically handled by SAS/ACCESS, which maps conversions to a string and a number. With this behavior in mind, it is impossible to write a general-purpose task that filters any table in a relational database management system (RDBMS).
  - For library engines that do not support concurrent access (for example, file systems), flow executions could fail when the data is produced by SAS Infrastructure for Risk Management (for example, output to a file) or accessed outside of SAS Infrastructure for Risk Management (for example, a file as input to a job flow).

    For example, if you have a job flow definition that creates a report.txt file outside of the persistent area, two or more concurrent executions of an instance of that flow could fail with an I/O error.
  - With generic libraries, the SAS Infrastructure for Risk Management task scheduling algorithm is no longer guaranteed to trigger task executions at the correct time. In particular, SAS Infrastructure for Risk Management will not detect cycles that are triggered by a dependency in the third-party system.
  - The auditing for uploads and downloads might not be accurate since the auditing might occur in the third-party system.
- Security
  - Without SAS Infrastructure for Risk Management asserting control over data that is read or written, there is no guarantee that the data is filtered according to the permissions of the user.
  - A separate authentication domain might have to be created for each user to access SAS Infrastructure for Risk Management securely.
- Performance
  - Tasks with generic libraries cannot leverage data object pooling. A lack of data object pooling typically results in an order of magnitude of performance degradation.
  - The SAS Infrastructure for Risk Management scheduling algorithm can no longer optimize disk access for tables that are held in a third-party system. This is true for a single machine or a grid deployment.

- During flow execution, multiple copies of the input and output data might occur between SAS Infrastructure for Risk Management and the third-party system (in both directions). It is recommended that you do not write SAS Infrastructure for Risk Management job flows that keep data in a third-party system rather than in the persistent area. Flow execution of data in a third-party system is relatively slow and expensive.

In addition, there is no live ETL with generic libraries since they do not use the input area or the landing area. The result of no live ETL is that when you use generic libraries, SAS Infrastructure for Risk Management can no longer reliably determine whether your job flow instances are up-to-date. Consequently, your users must ignore version information in the SAS Infrastructure for Risk Management web application about whether a job flow instance is current.

Before you adopt generic libraries for your job flow and task designs, carefully review these limitations and make sure that they will not interfere with the operations of your solution or that you have appropriate workarounds in place.

Configuring a One-Time Password for a Task

SAS Infrastructure for Risk Management provides system-reserved parameters that you can use to tag a task to indicate that a one-time password is required to execute the task. A one-time password might be needed to execute a task when the task requires credentials to access other applications outside of the SAS Infrastructure for Risk Management server.

To configure a one-time password for a task, include the following parameter in the task file header:

```
\param[in] %IRM_NUM_OF_ONE_TIME_PASSWORD=1
```

The parameter value specifies the number of one-time passwords that can be generated for the task.

After specifying the one-time password parameter in the task header, you reference the user ID and one or more generated passwords in the task code as &IRM_USER_ID and &IRM_USER_PASSWORD.

Here are examples of task code statements that reference the user ID and one or more one-time passwords:

- **%query_metadatalibs(userid=&IRM_USER_ID., password=&IRM_USER_PASSWORD.);**
- **%query_metadatalibs(userid=&IRM_USER_ID., password=&IRM_USER_PASSWORD_2.);**

Note: The one-time password feature is typically used with generic libraries during development and is not usually a part of deployments on production systems. Using generic libraries and one-time passwords violates the SAS Infrastructure for Risk Management job flow task's use of immutable inputs and outputs.

Reusing Subflows

You can use the IrmContext method add_sub_jobflow_by_reference to reuse existing subflows in a new job flow definition.

Suppose that you created the job flow (F_A) and a subflow (SUB_COMMON) in your personal federated area. In a new job flow script, you want to create a new job flow (F_B) that reuses the existing subflow (SUB_COMMON).

When you reuse existing job flow definitions and subflow definitions, the following rules apply:

- You can reuse only job flows that are created by the scripting client.
- Job flows can come from your personal federated area or a Read-only federated area.

For information about the syntax for reusing a subflow, see Appendix 1, "Scripting Client Reference".

Enabling a Python Task to Request a Service Ticket

A Python task might need to make REST calls to other applications. To make REST calls, a task needs access to a ticket-granting ticket (TGT), which contains the credentials of the user who is logged in. The task requests a service ticket (ST) from the TGT, and the ST provides access to the user's credentials. The ability to request a service ticket enables a user to log on to SAS Infrastructure for Risk Management using their credentials and then to log on to additional applications without being challenged for logon credentials. When a TGT is used, user credentials do not need to be stored in the SAS Metadata Server.

Note: A service ticket can be used for only one service.

To enable a task to request a service ticket, declare the following SAS Infrastructure for Risk Management **`%irm_tgt`** reserved substitution parameter as the task input in the task file header:

```
## \param[in] %irm_tgt=expiration-value
```

where *expiration-value* specifies how access to a TGT expires. Here are the two possible values:

- **`EXPIRE_WITH_TASK`**

  The TGT expires when the task execution has completed. This is the most secure option, because it ensures that a ticket cannot be used by another process at a later time.

- **`EXPIRE_WITH_TICKET`**

  The TGT expires when the ST expires or, if the execution completed with errors, immediately following the execution of the task.

Note: When defining parameter substitution, ensure that there are no spaces or tabs before or after the equal sign (=).

To make the TGT support more useful, SAS Infrastructure for Risk Management provides the following new macro variables and SAS autocall macro:

- **`IRM_MIDTIER_HOST`** — contains the host name of the SAS Infrastructure for Risk Management mid-tier server.
- **`IRM_MIDTIER_PORT`** — contains the port number that the SAS Infrastructure for Risk Management server is running on.
- irm_rest_service_ticket_from_tgt.sas — creates an ST from a TGT and URL.

Appendix 1

Scripting Client Reference

About the Python Scripting Client Reference Information

This reference information provides a complete list of the SAS Infrastructure for Risk Management Python scripting client commands, classes, and methods that you can use in job flow scripts to create Python programs.

In addition, this section provides instructions on how to access reference information for the SAS Infrastructure for Risk Management sample content. This documentation is provided in the SAS Infrastructure for Risk Management content.

Reference Documentation for Sample Python Flows

For additional information about the SAS Infrastructure for Risk Management Python scripting client, see the *SAS Infrastructure for Risk Management: Python Flows Reference Manual* that included in the SAS Infrastructure for Risk Management content.

This content is automatically installed when SAS Infrastructure for Risk Management is installed.

This reference documentation provides information about the sample Python flows and job flow scripts that are available, and how to use them.

To access this reference documentation, complete the following steps:

1. Log on to the SAS Infrastructure for Risk Management web application.
2. Open a job flow instance, right-click on a Python task and select **Show Help**.

   Details about that Python task are displayed.
3. To view the reference documentation, click **Main** in the menu bar.

Scripting Client Commands

The SAS Infrastructure for Risk Management development package for Python provides commands that you use to create and configure your scripting client development environment. You execute these commands from a command prompt on your local development machine.

Command	Description
irmdev init	Initializes your development environment and stores configuration information (for example, your SAS Infrastructure for Risk Management server host and port and user credentials). The information that you enter at the prompts when you execute this command is stored in hidden files (.irm and .authinfo) so that it does not appear in your code.
irmdev mount	Mounts your personal federated area on the SAS Infrastructure for Risk Management server to a host.
irmdev refresh	Refreshes your development environment, generates task API packages, applies any changes made to libnames.txt, and makes that data available. Run the **irmdev refresh** command anytime that task changes are made.
irmdev reset	Resets your development environment.
irmdev umount	Unmounts a local machine that is mounted to your personal federated area on the SAS Infrastructure for Risk Management server.

IrmConfig Overview

The IrmConfig class in theSAS Infrastructure for Risk Management Python scripting client causes your scripting client configuration to persist. It loads the stored configuration information that you entered when you ran the **irmdev init** command. The IrmConfig class provides Python setter methods that you can use to change the configuration at run time.

Syntax

```
IrmConfig()
```

Example:

```
your-IrmConfig-object = IrmConfig()
```

Description

Parameter: type, default	Description
auto_save: bool, default = False	Specifies to automatically save to the configuration file. Valid values are True (save) or False (do not save). The default is False. Example: IrmConfig(auto_save=False)

IrmConfig Methods

get_all

Gets all configuration key-value pairs.

Syntax

```
your-IrmConfig-object.get_all()
```

Returns

A dict containing configuration key-value pair.

get_value

Gets value from a configuration key.

Syntax

```
your-IrmConfig-object.get_value(key)
```

Description

Parameter:	Description
key	Key defined in IRM_prefix. Available keys: • IRM_USER_NAME • IRM_SERVER_PORT • IRM_SERVER_URL • IRM_SERVER_IS_HTTPS • IRM_FLOW_BASE_DATE • IRM_FLOW_CATEGORY_ID • IRM_FLOW_CONFIG_SET_ID • IRM_FLOW_ENTITY_ID • IRM_FLOW_ENTITY_ROLE_KEY

Returns

A configuration value.

save_to_config_file

Saves to a configuration file.

Syntax

```
your-IrmConfig-object.save_to_config_file()
```

set_flow_base_date

Sets the job flow base date.

Syntax

```
your-IrmConfig-object.set_flow_base_date(base_date)
```

Description

Parameter:	Description
base_date	Base date entered as **mmddyyyy**.

Returns

IrmConfig object.

set_flow_category_id

Sets the job flow category ID.

Syntax

```
your-IrmConfig-object.set_flow_category_id(category_id)
```

Description

Parameter:	Description
category_id	Job flow category ID.

Returns

IrmConfig object.

set_flow_config_set_id

Sets the job flow configuration set ID.

Syntax

```
your-IrmConfig-object.set_flow_config_set_id(config_set_id)
```

Description

Parameter:	Description
config_set_id	Job flow configuration set ID.

Returns

IrmConfig object.

set_flow_entity_id

Sets the job flow entity ID.

Syntax

```
your-IrmConfig-object.set_flow_entity_id(entity_id)
```

Description

Parameter:	Description
entity_id	Job flow entity ID.

Returns

IrmConfig object.

set_flow_entity_role_key

Sets the job flow entity role key.

Syntax

```
your-IrmConfig-object.set_flow_entity_role_key(entity_role_key)
```

Description

Parameter:	Description
entity_role_key	Job flow entity role key. Available keys: • IrmEntityRole.ANY • IrmEntityRole.SOLO • IrmEntityRole.GROUP • IrmEntityRole.NOTHING *Note:* You can import the IrmEntityRole class from the irm.config package.

Returns

IrmConfig object.

set_server_https_on

Enables the HTTPS protocol for the REST client.

Syntax

```
your-IrmConfig-object.set_server_https_on(is_on)
```

Description

Parameter:	Description
is_on :Boolean value	Boolean value is on.

Returns

IrmConfig object.

set_server_port

Sets the port number of the SAS Infrastructure for Risk Management server.

Syntax

```
your-IrmConfig-object.set_server_port(server_port)
```

Description

Parameter:	Description
server_port	Server port number. The default on Linux is 7980. The default on Windows is 80.

Returns

IrmConfig object.

set_server_url

Sets the URL to the SAS Infrastructure for Risk Management server.

Syntax

```
your-IrmConfig-object.set_server_url(server_url)
```

Description

Parameter:	Description
server_url	URL to the SAS Infrastructure for Risk Management server.

Returns

IrmConfig object.

set_user_password

Sets the password for the SAS Infrastructure for Risk Management user.

Syntax

```
your-IrmConfig-object.set_user_password(user_password)
```

Description

Parameter:	Description
user_password	Password for the SAS Infrastructure for Risk Management user name.

Returns

IrmConfig object.

set_value

Sets configuration key-value pairs.

Syntax

```
your-IrmConfig-object.set_value(key, value)
```

Description

Parameter:	Description
key	Configuration key defined in IRM_prefix. Available keys: • IRM_USER_NAME • IRM_SERVER_PORT • IRM_SERVER_URL • IRM_SERVER_IS_HTTPS • IRM_FLOW_BASE_DATE • IRM_FLOW_CATEGORY_ID • IRM_FLOW_CONFIG_SET_ID • IRM_FLOW_ENTITY_ID • IRM_FLOW_ENTITY_ROLE_KEY
value	Configuration value.

Returns

IrmConfig object.

IrmContext Overview

The context class serves as the entry point to the Python scripting client. Use **IrmContext** to create an object that is used to interact with the scripting client to create a parallel program. The IrmContext class is delivered in the irm.context package.

Syntax

```
IrmContext(irm_conf, flow_definition_name=None,
category_id=None, should_gen_api=False)
```

Example:

```
your-IrmContext-object = IrmContext(your-IrmContext-object)
```

Description

Parameter: type, default	Description
irm_conf: IrmConfig	Configuration object for the scripting client.
flow_definition_name: str, default None	Name of the job flow definition.
category_id: str, default None	ID of the job flow category. If the category ID is None, the category ID defined in the IrmConfig object is used.
should_gen_api: bool, default False	Default is false.

IrmContext Methods

add

Adds a task or sub job flow to a job flow to the job flow definition.

Syntax

```
your-IrmContext-object.add(flow_element, name=None,
task_type=TASK_EXECUTION_TYPE_COMPUTATIONAL,
cardinality_table=None)
```

Returns

If the flow_element is an IrmContext object, the calling IrmContext object is returned after adding the flow element subflow. If flow_element is a task object, the added task is returned.

Example:

```
your_flow_ref.add(mytaskname(substitution_dict={
        "%inoutparam":"mylib.mtable.ext"}),
        name="descriptive_name")
```

Parameters

Parameter: type, default	Description
flow_element: Task, IrmContext	Specifies whether a task or sub job flow is being added.

Parameter: type, default	Description
name: str, default None	Specifies the name of the take. If a name is not specified, the task default name is used.
task_type: str, default Task.TASK_EXECUTION_TYPE_COMPUTATIONAL	Specifies the task execution type. Valid values are Task.TASK_EXECUTION_TYPE_COMPUTATIONAL, Task.TASK_EXECUTION_TYPE_RECOMBINE, or Task.TASK_EXECUTION_TYPE_PARTITION. The default is Task.TASK_EXECUTION_TYPE_COMPUTATIONAL.
cardinality_table: str, default None	Specifies the name of the cardinality table. This parameter is required if the task_type is Task.TASK_EXECUTION_TYPE_PARTITION.

add_partition_task

Adds a partition task to a job flow.

Syntax

your-IrmContext-object.add_partition_task(task, cardinality_table, name=None)

Returns

The added partition task.

Example:

```
your_job_flow_ref.add_partition_task(mynodes.irm_partition_byn(substitution_dict={
        "%inoutparam":"mylib.mtable.ext"}),
        cardinality_table="mylib.cardtable.ext",
        name="descriptive_name")
```

Parameters

Parameter: type, default	Description
task: Task	Specifies the partitioned task to add.
cardinality_table: str	Specifies the name of the cardinality table
name: str, default None	Rename the default task from node package. Name for the task. If None, the default name is used.

add_recombine_task

Adds a recombine task to a job flow.

Syntax

your-IrmContext-object.add_recombine_task(task, name=None)

Returns

The recombined task.

Example:

```
For example:
  your_job_flow_ref.add_recombine_task(mynodes.irm_recombine_byn(substitution_dict={
        "%inoutparam":"mylib.mtable.ext"}),
        name="descriptive_name")
```

Parameters

Parameter: type, default	Description
task: Task	Specifies the recombine task to add.
name: str, default None	Rename the default task from node package. Name for the task. If None, the default name is used.

add_sub_flow

Adds a subflow to a job flow.

Syntax

your-IrmContext-object.add_sub_flow(sub_flow)

Returns

An IrmContext object for the job flow.

Example:

```
your_toplevel_flow_ref.add_sub_flow(your_subflow_ref)
```

Parameters

Parameter: type, default	Description
sub_flow: IrmContext	IrmContext object for the sub flow.

add_sub_jobflow_by_reference

Adds a subflow by reference to a job flow.

Syntax

your-IrmContext-object.add_sub_jobflow_by_reference(referenced_jobflow_name, referenced_category_id, is_in_subflow_folder=True)

Returns

New referenced job flow.

Example:

```
your_jobflow_ref.add_sub_jobflow_by_reference("your_subflow_name", "mysample")
```

Parameters

Parameter: type, default	Description
referenced_jobflow_name: str	Referenced job flow name.
referenced_category_id: str	Referenced job flow category ID.
is_in_subflow_folder: bool, default True	Boolean value to indicate whether the referenced job flow is in a subflow folder.

add_task

Adds a task to a job flow.

Syntax

your-IrmContext-object.add_task(task, name=None)

Returns

The added task.

Example:

```
For example:
  your_flow_ref.add_task(mytaskname(substitution_dict={
        "%inoutparam":"mylib.mtable.ext"}),
        name="descriptive_name")
```

Parameters

Parameter: type, default	Description
task: Task	Specifies the task to add.
name: str, default None	Renames the default task from node package. Name for the task. If None, the default name is used.

add_to_parent_flow

Adds a job flow to a parent flow.

Syntax

your-IrmContext-object.add_to_parent_flow(parent_flow)

Returns

An IrmContext object for the parent flow.

Example:

```
your_other_flow_ref.add_to_parent_flow(your_parent_flow_ref)
```

Parameters

Parameter: type, default	Description
parent_flow: IrmContext	IrmContext object for the parent flow.

add_visual_input_table_array

Adds an input table array to a job flow for visualization. The input table names are added to the top-level view of the job flow instance diagram displayed in the SAS Infrastructure for Risk Management web application.

Syntax

your-IrmContext-object.add_visual_input_table_array(visual_input_table_array=None)

Returns

This IrmContext object.

Example:

```
your_jobflow_ref.add_visual_input_table_array (
          ["my.input01.ext","my.input02.ext"])
```

Parameters

Parameter: type, default	Description
visual_input_table_array: list, default None	Name of the input table array for visualization purposes.

add_visual_output_table_array

Adds an output table array to a job flow for visualization. The output table names are added to the top-level view of the job flow instance diagram displayed in the SAS Infrastructure for Risk Management web application.

Syntax

your-IrmContext-object.add_visual_output_table_array(visual_output_table_array=None)

Returns

This IrmContext object.

Example:

```
your_jobflow_ref.add_visual_output_table_array (
          ["my.output01.ext","my.output02.ext"])
```

Parameters

Parameter: type, default	Description
visual_output_table_array: list, default None	Name of the output table array for visualization purposes.

build_jobflow

Builds a new job flow object from the current context. Use **build_jobflow** to name and build a job flow definition.

Syntax

your-IrmContext-object.build_jobflow(flow_definition_name, category_id=None)

Returns

A new IrmContext object for the job flow.

Example:

```
your_flow_ref = your-IrmContext-object.build_jobflow("my_flow", "example")
```

Parameters

Parameter: type, default	Description
flow_definition_name: str	Name of the job flow definition.
category_id: str, default None	Specifies the category_id for the job flow definition. If a value is not specified, None (the default) is used.

create_instance

Creates a job flow instance.

Syntax

your-IrmContext-object.create_instance(enable_debug=False, enable_auto_run=True)

Returns

An instance key and a response code. If executed successfully, 200 is returned as the response code.

Example:

```
your_flow_ref.create_instance()
```

Parameters

Parameter: type, default	Description
enable_debug: bool, default False	Boolean flag that enables or disables verbose debug messages.
enable_auto_run: bool, default True	Boolean flag that indicates whether the job flow instance should be automatically executed.

delete_jobflow_definition

Removes a job flow definition from your personal federated area. (The job flow definition must be created by the scripting client.)

Syntax

your-IrmContext.delete_jobflow_definition(jobflow_name, category_id=None)

Returns

A tuple containing the job flow definition ID and a response code. If executed successfully, 200 is returned as the response code.

Example:

```
jobflow_definition_id, response_code = ic.delete_jobflow_definition("my_jobflow_001", "s
if response_code != 200:
   return "Fail to delete job flow definition"
```

Parameters

Parameter: type, default	Description
jobflow_name: str	Specifies the name of the job flow definition.
category_id: str, default None	Specifies the category_id for the job flow definition. If a value is not specified, None (the default) is used.

execute_instance

Executes a job flow instance.

Syntax

your-IrmContext.execute_instance(instance_key=None)

Returns

A response code for execution. If executed successfully, 200 is returned as the response code.

Example:

```
ic.execute_instance("123456789")
```

Parameters

Parameter: type, default	Description
instance_key: str	Specifies the job flow definition key. A value is required for this parameter.

generate_task_doc

Generates HTML documentation for the tasks in your personal federated area. The generated task documentation (content taken from the task file header \brief and \details tags) HTML files are written to the **doc** folder of your personal federated area. To access this documentation, right-click a task in the SAS Infrastructure for Risk Management web application, and select **Show help**.

Syntax

your-IrmContext-object.generate_task_doc()

Returns

A message and a response code.

Example:

```
your_flow_ref.generate_task_doc()
```

get_all_jobflow_definitions

Retrieves all job flow definitions.

Syntax

your-IrmContext.get_all_jobflow_definitions()

Returns

A tuple containing a string with all job flow definitions and a response code. If executed successfully, 200 is returned as the response code.

Example:

```
ic.get_all_jobflow_definitions()
```

get_instance_infos

IrmContext method that retrieves the job flow instance information.

Syntax

your-IrmContext-object.get_instance_infos(instance_key)

Returns

A tuple containing the job flow instance info and a response code. If executed successfully, 200 is returned as the response code.

Example:

```
your_flow_ref.get_instance_infos(123456789)
```

Parameters

Parameter: type, default	Description
instance_key: int	job flow instance key.

get_json_stream

Retrieves the job flow JSON stream.

Syntax

your-IrmContext-object.get_json_stream()

Returns

A job flow in JSON format.

Example:

```
your_flow_ref.get_json_stream()
```

get_task_infos

Retrieves information about the available tasks from federated areas.

Syntax

your-IrmContext.get_task_infos(jobflow_name, category_id=None)

Returns

A tuple containing a string with the available tasks and a response code. If executed successfully, 200 is returned as the response code.

Example:

```
ic.get_task_infos()
```

Note: Run the **`get_task_infos`** method with DEBUG level logging to display a URL link in the console that enables you to view the task information list.

Parameters

Parameter: type, default	Description
instance_key: int	Job flow instance key.

refresh_pfa

Refreshes the personal federated area and updates the task APIs.

Syntax

`your-IrmContext.refresh_pfa()`

Example:

```
ic.refresh_pfa()
```

run

Executes a job flow definition and creates an instance on the SAS Infrastructure for Risk Management server.

Syntax

`your-IrmContext-object.run(enable_debug=False, instance_name=None)`

Returns

A tuple containing the instance key and a response code. If executed successfully, 200 is returned as the response code.

Example:

```
your_flow_ref.run()
```

Parameters

Parameter: type, default	Description
enable_debug: bool, default False	Boolean flag that enables or disables verbose debug messages.
instance_name: str, default None	Name of the job flow instance. If a name is not specified, None (default) becomes the name of the job flow instance.

save

Saves a job flow definition to the SAS Infrastructure for Risk Management server.

Syntax

`your-IrmContext-object.save()`

Returns

A tuple containing the job flow definition string and response code. If executed successfully, 200 is returned as the response code.

Example:

```
your_flow_ref.save()
```

Recommended Reading

- *SAS Infrastructure for Risk Management: Administrator's Guide*
- *SAS Infrastructure for Risk Management: User's Guide*

For a complete list of SAS publications, go to support.sas.com/en/books.html. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: support.sas.com/en/books.html