Features and Limitations

General requirements for the operation of Python nodes in Megaladata are the following:

- Python must be installed on the system with the same bitness as the Megaladata application/server.
- In Megaladata, preliminary configuration may be required. For details, see Administration and Component Parameters: Python.
- The supported Python versions are 3.5 and above.
- The latest tested Python version is 3.12.

For desktop editions, the execution of the Python node is allowed by default; for server editions it is disabled. If necessary, these settings can be changed in **Administration** > **Parameters** > **Component parameters: Python**.

Megaladata supports two modes of Python operation:

- within the Megaladata process (Windows only)
- in a separate process (on Windows and Linux)

Running Python nodes within the Megaladata process

Features and limitations:

- Python from the Anaconda distribution is not supported.
- Modules from the multiprocessing package are not supported.
- Creating a GUI from Python code is not the intended purpose of the Python component and in some cases can lead to errors. For example:
  - For `matplotlib` to work with the Qt backend and with PyQt in general, a `qt.conf` file with paths to Qt libraries/plugins is required. If they cannot be found, the Megaladata application will be forcibly terminated with an error message.
- Unfinished background threads can lead to an application crash.
- In case of critical errors, `python3x.dll` can cause the application to crash. It is recommended to use pre-debugged Python code.
- Only one Python node can be executed at a time.
- Execution of a Python node can only be stopped between interpreter instructions.
- A virtual environment can be used by setting the **Interpreter path** parameter for the Python component in **Component parameters: Python**. To do this, the interpreter path must point to the `python.exe` inside the virtual environment (see **Finding Python modules inside the Megaladata process** further on). For correct operation, the version of `python.exe` in the virtual environment and the version of `python3x.dll` running the node's code must match.

> **Note:** The in-process Python execution mode is not supported on Linux. Python nodes configured to run in-process will run in a separate process on Linux.

Running a Python node in a separate process

Features and limitations:

- When running executable Python code in a separate process, multiple Python nodes can be executed in parallel.
- The **Start wait timeout (ms)** option is not supported.

Running Python nodes in an isolated environment

On Linux, it is possible to run Python nodes in an isolated user environment. To do this, you need to:

- In the **Administration** pane, go to **Parameters** and enable the **Pass node environment variables** option under **Component parameters: Python**.
- In the **Interpreter Path** parameter, specify the full path to a bash script— `python_run.sh` or similar.
- Install Docker or Podman. In `python_run.sh`, there is a `DOCKER_OR_PODMAN` parameter responsible for the containerization method; the default value is `podman`.
- Prepare a base Docker image with Python and the necessary Python packages inside. In `python_run.sh`, the image name is specified in the `IMAGE_NAME` parameter; the default value is `python`.

The `python_run.sh` script has a `CONTAINER_OPTIONS` parameter where you can pass container settings, for example, the value of the `oom-score-adj` variable. By default, it is set to `oom-score-adj=1000`. This means that if the OOM Killer needs to terminate a process, the container will be the first to terminate.

Finding Python modules inside the Megaladata process

The Python node's code is executed in the `__main__` module. Thus, the condition `__name__ == '__main__'`, which is commonly used to run scripts, is met.

Importing modules from the file storage is supported. The path to the folder containing the interpreter is added as another path to `sys.path` —the list of paths where Megaladata searches for modules.

- If the package is saved, the relative path to the imported module starts with the package's location. The sys.path list (`sys.path[0]`) contains the path to the directory where the Python node's package is located.
- If the package is not saved, the relative path to the imported module starts with the user's directory. The sys.path list (`sys.path[0]`) contains the path to the user's directory.

If a file named `pyvenv.cfg` exists one directory above the **Interpreter path** directory, then `sys.prefix` and `sys.exec_prefix` are set to that directory, and the presence of `site-packages` is checked. The found `site-packages` is added to `sys.path`. If `pyvenv.cfg` contains the `include-system-site-packages` key and its value is not `true`, the base path to `site-packages` (relative to `sys.base_prefix`) is not included in `sys.path`.

In modules, the relative path starts from the location of the importing module.

▼ Example:

The following modules are located relative to the saved Megaladata package:

- module `./foo/__init__.py` :

```
__all__ = ["module"]
from .foo import cube
```

- module `./foo/foo.py` :

```
def cube(x):
    return x * x * x
```

- module `./foo/module/__init__.py` :

```
from .module import say_hello
```

- module `./foo/module/module.py` :

```
def say_hello():
    return "Hello"
```

In this case, importing and calling functions from these modules can be done in the Python node as follows:

```
from foo import cube
from foo.module import say_hello

print(say_hello())
print("3 ** 3 =", cube(3))
```

In the Preview mode , the value of the `__file__` attribute of the `__main__` module is `<preview>` ; when the node is executed, it is `<main>` .

▼ Example:

```
from builtin_data import OutputTable

if __file__ == "<preview>":
    print(__file__)
else:
    OutputTable.Append()
    OutputTable.Set(0, __file__) #__file__ == "<main>"
```

Read on: Variables