­­Pypeline

What is ‘Pypeline’?

* Pypeline allows you to call Python code from within a running AnyLogic model by connecting to a local Python installation.
* With this approach, you can make use ofany installed Python libraries on your machine or custom code from an existing solution that you want to incorporate into your AnyLogic model.
* Due to the nature of its design, you can run any sort of parallel-running AnyLogic experiment (Parameter Variation, Optimization, etc.) without any extra code. In addition, JSON support is also included.
* It allows you to communicate between Java and Python for cases like:
  + Utilizing code that was previously written in Python without having to port to Java
  + Writing complex algorithms in Python that you can call in Java, optionally passing objects/data between the languages
  + When a certain library is only available in Python and you do not want to recreate it in Java
  + Using simulation as a testbed for testing trained artificial intelligence policies; useful for examining AI behavior in new situations, not just on historical data

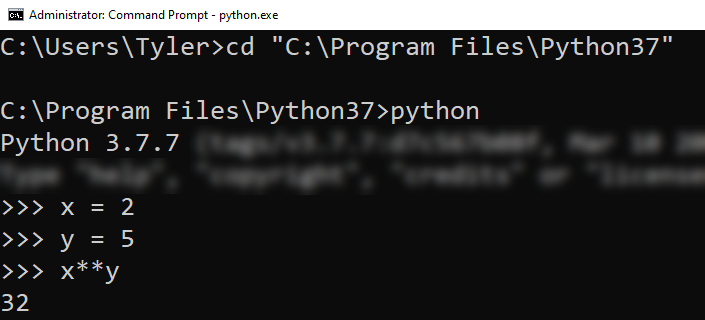
Warnings and caveats about use

* Pypeline is released as a complimentary library for AnyLogic users. It is not part of the main AnyLogic product, and The AnyLogic Company is not obligated to provide support for users or to provide future support/updates. It’s encouraged to make use of the community features available on GitHub (issues tab, forking, etc.) or elsewhere online.
* Using Pypeline is not a substitute for Java, which still remains as the only native scripting language of AnyLogic. You can (and should) build models in the AnyLogic GUI exactly as before, making full use of AnyLogic’s extensive native capabilities.
* Pypeline will also add some computational overhead to your model and therefore may not be the best option if computational efficiency is a priority in your models.
* As AnyLogic (Java) is in control of Python’s execution, the current version of Pypeline is ***not***suited for reinforcement learning or other machine learning training. However, we are working towards a version that lets you control exported AnyLogic models which would be applicable to reinforcement learning.

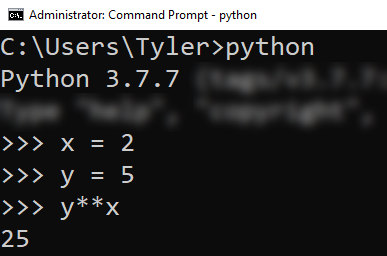
We have provided 3 demos and 4 example models with the library. Please review them to get some inspiration of possible use cases and to better understand the workflow.

## How the Java-Python connection works

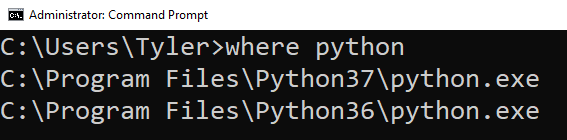
Traditionally, Python code is executed by either running pre-written files or interpretively (the code is evaluated on the spot) – and it’s the latter method that Pypeline takes advantage of. One way to run Python interpretively is by opening a command prompt, navigating to the installation directory, and running the Python executable (as seen in the screenshot below).

  
The `cd` command is used to change the active directory;   
Irrelevant text is blurred to avoid confusion

To avoid having to constantly navigate to the directory an executable file is in, all systems have a system “path”. The path consists of a series of folder that tells your system where to look for executables. If you have the Python installation directory on your system path, you can start the Python executable from anywhere (as seen in the screenshot below).

  
Irrelevant text is blurred to avoid confusion

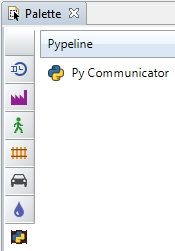
This lays the foundation for how Pypeline is able to interact with a live Python environment. Behind the scenes, it runs a command to search for a version of Python on your system path. For Windows, it uses the where command and for Mac/Linux it uses the find command. What it searches for depends on the configuration you setup in the Python Communicator object. If you select “auto”, it searches for “python”; selecting “python 2” or “python 3” will search for “python2” or “python3”, respectively. In case your desired version of Python is not on the system’s path, you can specify a custom location. More context is provided in the following section.

  
If multiple executables are found, only the top-most one is used

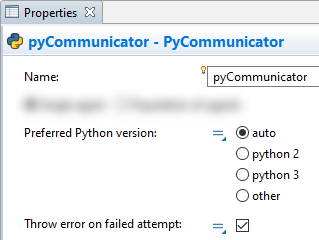
With the interpretive environment created, the code is run (via the provided functions) just like any other interpretive environment.

## Pypeline setup

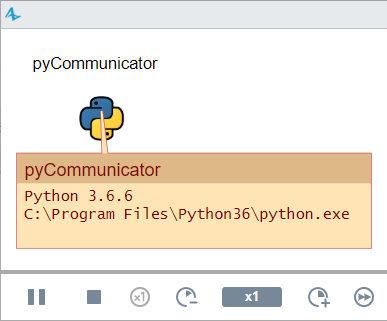
1. Before you can start using Pypeline, you’ll need to add the library to your AnyLogic environment. Place the jar file in a location where it won’t be moved from (or accidently deleted). Then, follow the steps described in [this AnyLogic help article](https://help.anylogic.com/topic/com.anylogic.help/html/libraries/Managing%20Libraries.html). Afterwards, you should see a new tab in your AnyLogic palette, as shown below:



1. Drag the “Py Communicator” object (henceforth referred to as “the Communicator”) into an existing model. There are two options to configure (see the screenshot below):
   1. Preferred Python version
      1. This informs the library which version of Python you want to use.
      2. Keeping the default option (“auto”) corresponds to whichever version of Python responds to the `python` command (depends on your system path)
      3. Selecting the “other” option allows you to specify the full path to a Python executable (Windows users: backslashes will need to be escaped)
   2. Throw error on failed attempt
      1. This will cause a runtime error to be thrown if any of your Python commands fail



1. Use the available functions (described in the next section) to execute Python code.
2. Run your model! If you click on the Communicator object to view its inspection window, the version of Python that is currently running will be listed. An example is shown in the image below:



## Pypeline usage

There are two primary functions of the Communicator that you will be using:

1. run(String…)
   1. This function is for statements that do *not* have any sort of return value (e.g., import statements, variable assignments)
2. runResults(String…)
   1. This function is for statements are *do* have an expected return value (e.g., function calls, calculations, retrieving the values of variables)

Each take an argument of one or more strings (indicated above by the class – String – and ellipsis to mean one or more). When multiple strings are passed at once, they are treated as being on their own line. This is useful for tasks like importing multiple libraries in one call or writing loops. Some generic examples are shown below:





Note: for multi-lined code, it does not matter how many spaces/tabs are used to indent; it only matters that the indentation is consistent (e.g., a double indentation should be twice the number of spaces/tabs as the first).

Each function returns an `Attempt` object. This object has two functions of its own:

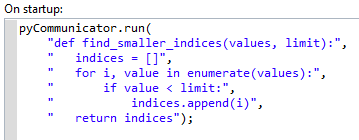
1. isSuccessful()
   1. This function returns a boolean representing whether the Python call executed without errors
2. getFeedback([Class])
   1. This function will return the result or response of the call to Python
   2. Calling without any arguments will return a String. If you know the data type of the response in advance, you can optionally pass the class
   3. If using the `run` command, there will only be any feedback if the command failed to execute
   4. When commands fail to execute, the error message will be contained in the feedback

Please see the demos and sample models for examples of these functions.

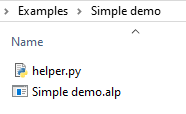
## Recommended workflow

While Pypeline allows you to seamlessly execute Python code inside AnyLogic, it should ideally be used in a way that most of your algorithms and function definitions are written outside of AnyLogic. The reasoning for this is for both organizational purposes and to be able to effectively debug your code outside of AnyLogic. Writing the bulk of your Python code in a standard Python IDE also allows you to utilize code completion and macros.

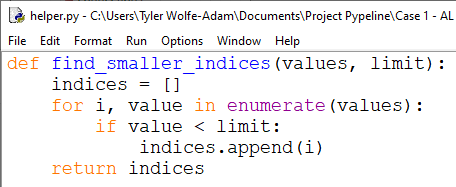
To elaborate with an example, let’s say you have a list of numbers and you want to get the indices less than a given value. For example, given the list: [1, 2, 9, 4] and the value 5, the function would return the list [0, 1, 3]. Creating this function directly inside AnyLogic using Pypeline would look like the following:



While this is valid and works okay for small examples, there is a more streamlined, flexible, and easier-to-manage alternative: any Python file in your model’s directory can be imported like any global library. For this example, I created a new file in my model’s folder called “helper.py” (as seen below):



Inside this file is the function I had to previously pass as strings:



In lieu of the function definition placed on my model’s startup, I can now directly import the function from the file, like so:

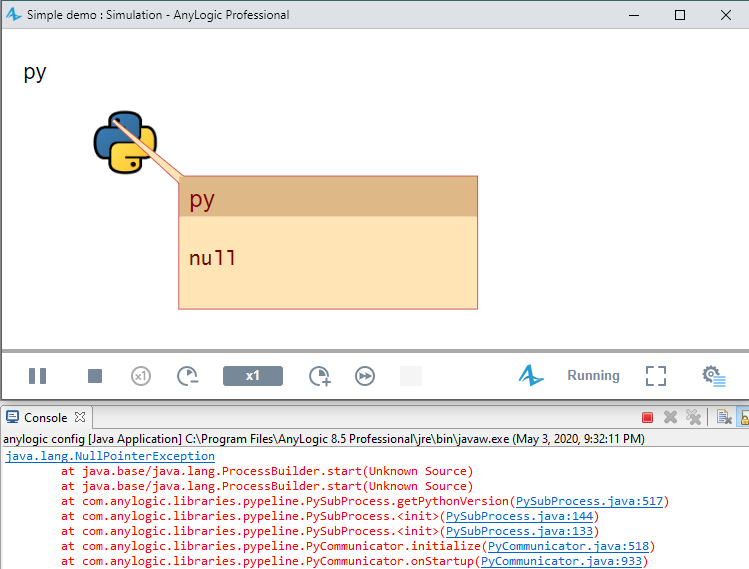


Now, this function is called in exactly the same way as with the string-defined version.

## Troubleshooting

As Pypeline relies on the existence of Python in your system path, and with each system being different, you may encounter different issues when trying to setup. This section aims to help with troubleshooting by describing potential problems, explaining why they occur, and solutions.

Problem: After running your AnyLogic model, you receive a “NullPointerException”. Clicking on the Communicator object shows “null”.

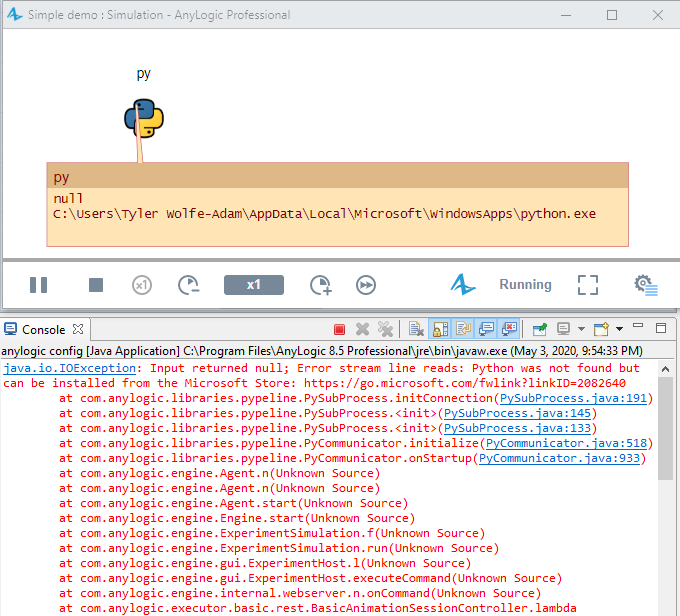


Explanation: No version of Python could be found.

Solution: Under the Communicator’s settings, for the Preferred Python version, if you have chosen “auto”, “python 2”, or “python 3”, this means that the Python executables do not sit on your system path. You have two options:

1. Add the Python executable to your desired version of Python to the system path. To do this, find the Python install directory (where the executable lives). Then, depending on your OS, follow any tutorial. For convenience, here are three for each major OS (not affiliated with the Pypeline project or AnyLogic):
   1. Windows: <https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/>
   2. Mac: <https://www.educative.io/edpresso/how-to-add-python-to-the-path-variable-in-mac>
   3. Linux: <https://docs.oracle.com/cd/E19062-01/sun.mgmt.ctr36/819-5418/gaznb/index.html>
2. Use the “Other” option and paste in the path to the Python executable

Problem: After running your AnyLogic model, you receive a “IOException” with a message about Python not being found. Clicking on the Communicator object shows “null” and a path to Python under a “WindowsApps” directory.

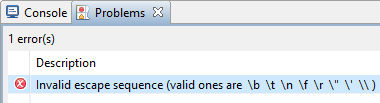


Explanation: Similar to the previous problem. In later versions of Windows 10, Microsoft included a version of Python inside the Windows store. If you go to a command prompt and try to enter in the `python` command, it will open the Windows store to where you can download it. It is not recommended to download from here (rather, you should install from python.org or use other Python environments, such as Spyder, Anaconda, etc.).

Solution: Please see the solution to the previous problem. Additionally, if you do not want to have the `python` or `python3` commands link to the Windows version, do the following:

1. Open the start menu and navigate to Windows Settings by clicking on the gear icon (above the power, in the lower left).
2. Navigate to Apps > App execution aliases
3. Disable the “App Installer” options for “python.exe” and “python3.exe”

Problem: When trying to run your AnyLogic model, you receive the error “Invalid escape sequence”.



Explanation: In Java, the backslash (\) is a special character that is used to “escape” other characters. One application is if you want a string that has a double quote character in it. For example:

String name = "John \"The man\" Doe";

Solution: A single backslash was used within a string with an invalid character following it. Most likely, you are on a Windows machine and have the preferred Python version set to “Other” with a custom path. Simply add an extra backslash to each backslash.

