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 of any 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 it 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

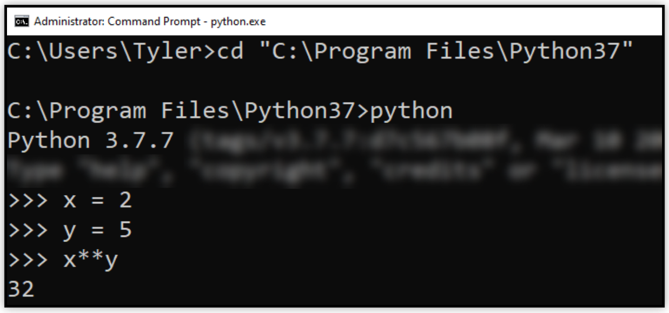
Warnings and caveats about use

- 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 (executing the code directly) – 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 Python installation directory, and running the Python executable (as seen in the screenshot below).

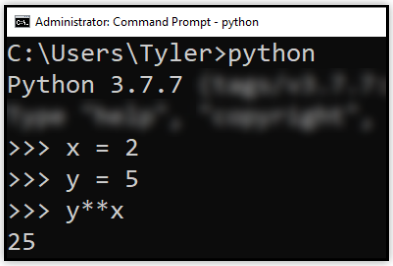


```
C:\Users\Tyler>cd "C:\Program Files\Python37"

C:\Program Files\Python37>python
Python 3.7.7
>>> x = 2
>>> y = 5
>>> x**y
32
```

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

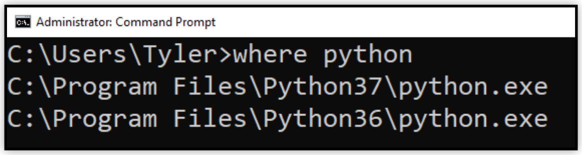
To avoid having to constantly navigate to the directory an executable file is in, all systems have a system "path". The path consists of a series of folders that tell your system where to look for executables. If you have the Python installation directory on your system path, you can start the Python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



```
C:\Users\Tyler>python
Python 3.7.7
>>> x = 2
>>> y = 5
>>> y**x
25
```

Irrelevant text is blurred

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 Pypeline Communicator object inside AnyLogic.



```
C:\Users\Tyler>where python
C:\Program Files\Python37\python.exe
C:\Program Files\Python36\python.exe
```

If multiple executables are found on the system path, only the top-most one is used;
you can resolve this by changing your system path or in the Pypeline Communicator's preferences

With the interpretive environment created, the code is run (via the provided functions, as part of the Pypeline Communicator object) just like any other interpretive environment.

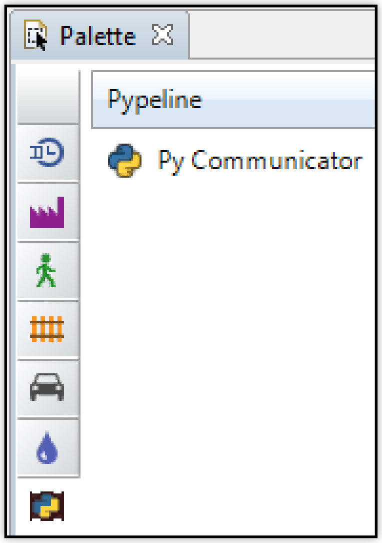
Pypeline setup

1. Before you can start using Pypeline, you'll need to add the library to your AnyLogic environment. Place the Pypeline jar file in a location where it won't be moved from (or accidently deleted).

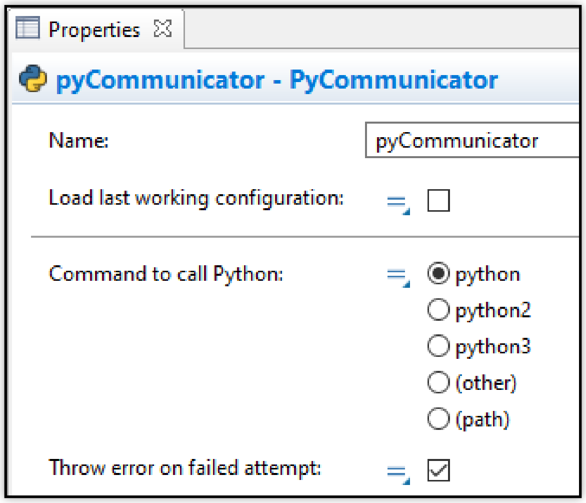
2. Inside AnyLogic, from within the Palette panel, select the "+" button on the bottom left. Then select "Manage libraries…", then "Add", then select the Pypeline jar file. Confirm all open windows.

    - Further details for adding a custom AnyLogic library are described in this AnyLogic help article.

3. Afterwards, you should see a new tab in your AnyLogic palette, as shown below:



4. Drag the "Py Communicator" object (henceforth referred to as "the Communicator") into an existing model. There are some options to configure (see the screenshot below):



Properties

pyCommunicator - PyCommunicator

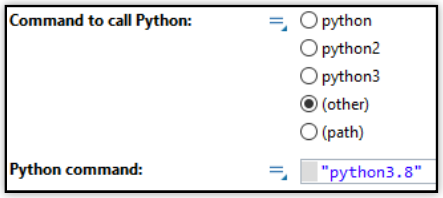
Name: pyCommunicator

Load last working configuration: ☐

Command to call Python: [✓ python] python2 python3 (other) (path)

Throw error on failed attempt: ☑

- Load last working configuration (default: false)
  - Selecting this option will override the remainder of the settings, using the last configuration that successfully connected to Python
  - This is useful when you have one version of Python that you always use, without having to reselect the options for it (as it works across all models)
  - If you have not previously run Pypeline, it will fall back to the default options
- Command to call Python (default: python)
  - This informs the library which command to use to call Python; for more information on how this works, see the above section: How the Java-Python connection works
  - Selecting "(other)" allows you to specify a custom alias/command to call Python; for example, see the screenshot below:



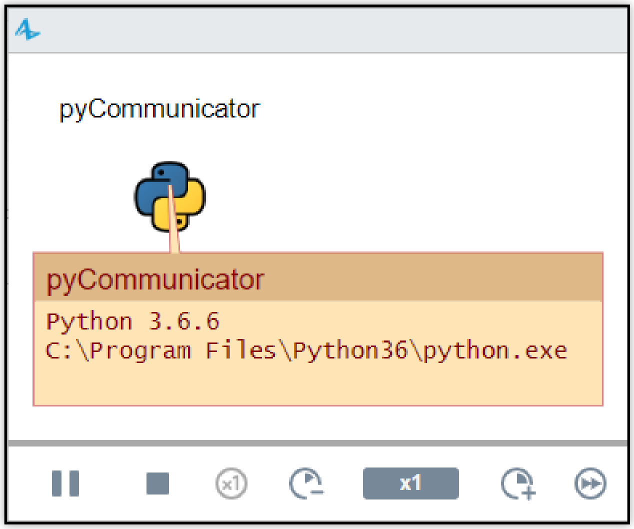
Command to call Python: python python2 python3 [✓ (other)] (path)

Python command: "python3.8"

  - Selecting the "(path)" option allows you to specify the full path to a Python executable (Windows users: backslashes will need to be escaped)
- Throw error on failed attempt (default: true)
  - This will cause a runtime error to be thrown if any of your Python commands fail

5. Use the available functions (described in the next section) to execute Python code.

6. Run your model! If you click on the Communicator object to view its inspection window, the version and executable path of the Python that is currently running will be listed. An example is shown in the image below:



```
Python 3.6.6
C:\Program Files\Python36\python.exe
```

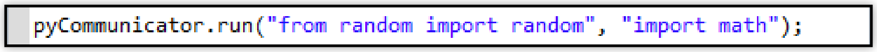
If you don't see this information, consult the section: Troubleshooting

Pypeline usage

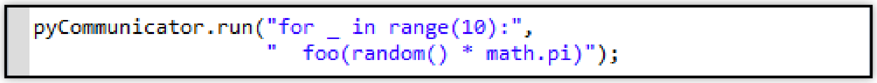
There are two primary functions of the Communicator that you will be using:

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

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 in the images below:



```
pyCommunicator.run("from random import random", "import math");
```



```
pyCommunicator.run("for _ in range(10):",
                   "  foo(random() * math.pi)");
```

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 a custom type that is part of Pypeline called `Attempt`. This object has two primary functions:

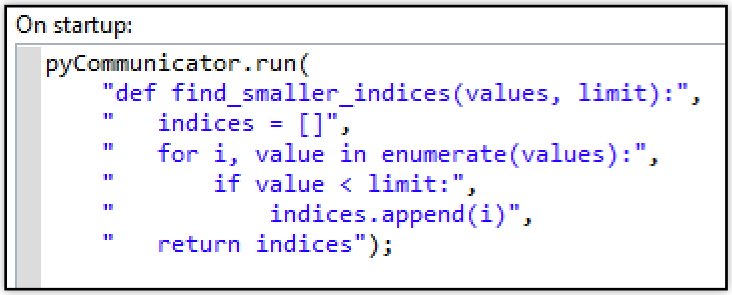
1. `isSuccessful()`
   - This function returns a boolean representing whether the Python call executed without errors
2. `getFeedback([Class])`
   - This function will return the result or response of the call to Python
   - Calling without any arguments will return a String. If you know the data type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

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

Recommended workflow

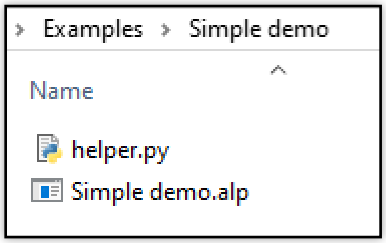
While Pypeline allows you to seamlessly execute Python code inside AnyLogic, it should ideally be used in a way that most of your algorithms and function definitions are written outside of AnyLogic. The reasoning for this is for both organizational purposes and to be able to effectively debug your code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

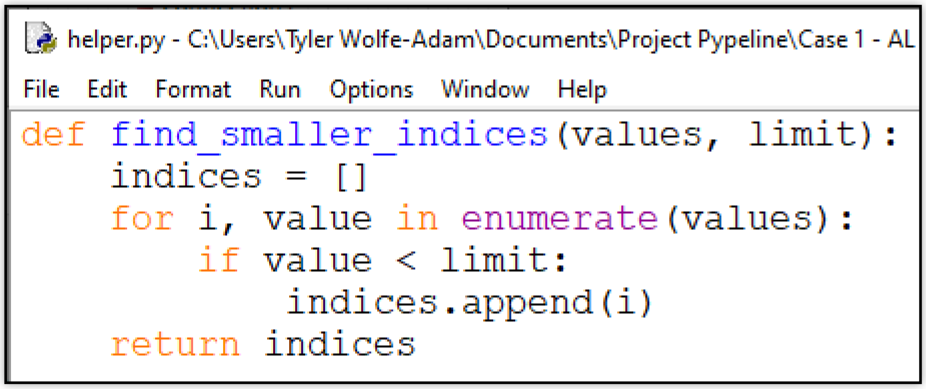

On startup:

```
pyCommunicator.run(
    "def find_smaller_indices(values, limit):",
    "   indices = []",
    "   for i, value in enumerate(values):",
    "       if value < limit:",
    "           indices.append(i)",
    "   return indices");
```

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):

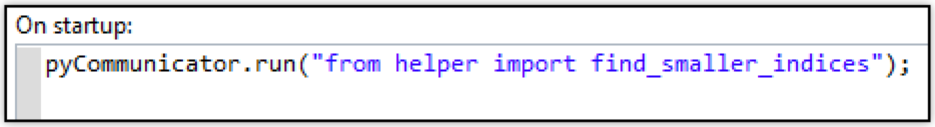


Inside this file is the function I had to previously pass as strings:


helper.py - C:\Users\Tyler Wolfe-Adam\Documents\Project Pypeline\Case 1 - AL
File Edit Format Run Options Window Help

```
def find_smaller_indices(values, limit):
    indices = []
    for i, value in enumerate(values):
        if value < limit:
            indices.append(i)
    return indices
```

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


On startup:

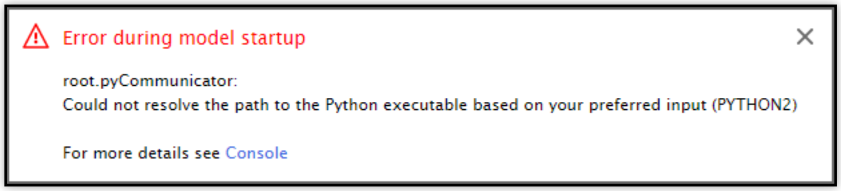
```
pyCommunicator.run("from helper import find_smaller_indices");
```

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 or during use. This section aims to help with troubleshooting by describing potential problems, explaining why they occur, and solutions.

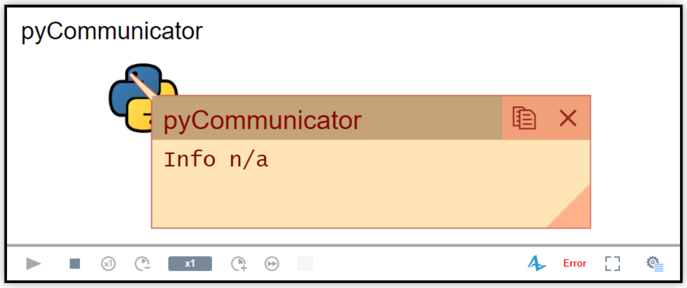
Problem: After running your AnyLogic model, you receive an error during model startup with a message about the path to the Python executable (an example can be seen in the first image below). Clicking on the Communicator object shows "null" or "Info n/a" (seen in the second image below).



```
Error during model startup

root.pyCommunicator:
Could not resolve the path to the Python executable based on your preferred input (PYTHON2)

For more details see Console
```



```
pyCommunicator
Info n/a
```

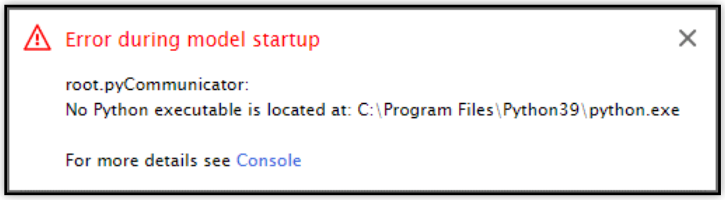
Explanation: No version of Python could be found based on how the Communicator's properties were configured. You will see this error if, under the Communicator's properties, for the command to call Python, you have chosen "python", "python2", "python3", or "(other)".

The error signifies that it tried calling the relevant command, but no version of Python responded. This is most likely due to not having an executable that sits on your system path.

Solution: You have two options:

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

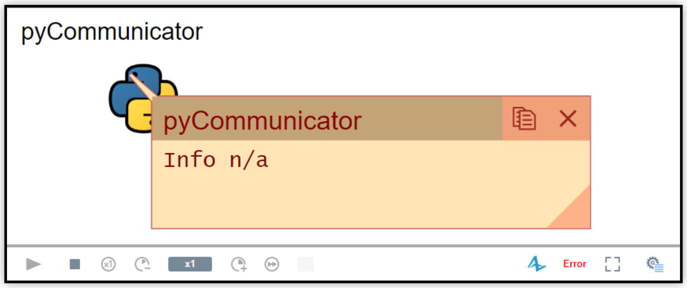
Problem: After running your AnyLogic model, you receive an error during model startup with a message about no Python executable being located at the specified path (an example can be seen in the first image below). Clicking on the Communicator object shows "null" or "Info n/a" (seen in the second image below).



Error during model startup

root.pyCommunicator:
No Python executable is located at: C:\Program Files\Python39\python.exe

For more details see Console



pyCommunicator

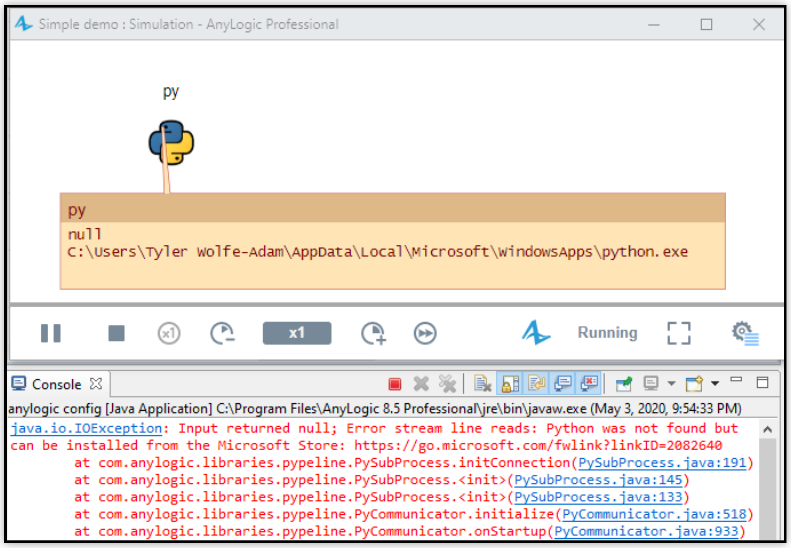
pyCommunicator

Info n/a

Explanation: No version of Python could be found based on how the Communicator's properties were configured. You will see this error if, under the Communicator's properties, for the command to call Python, you have chosen "(path)". This happens when the path that's been specified does not have a Python executable within it (if the path is a directory) or the file does not exist (if the path is to the executable).

Solution: For the custom path specified in the Communicator's properties, ensure that it is correct and points to a valid Python executable.

Problem: After running your AnyLogic model, you receive an “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.



```
py
null
C:\Users\Tyler Wolfe-Adam\AppData\Local\Microsoft\WindowsApps\python.exe
```

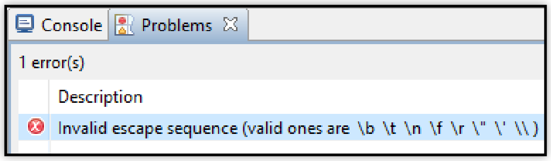
```
anylogic config [Java Application] C:\Program Files\AnyLogic 8.5 Professional\jre\bin\javaw.exe (May 3, 2020, 9:54:33 PM)
java.io.IOException: Input returned null; Error stream line reads: Python was not found but
can be installed from the Microsoft Store: https://go.microsoft.com/fwlink?linkID=2082640
	at com.anylogic.libraries.pypeline.PySubProcess.initConnection(PySubProcess.java:191)
	at com.anylogic.libraries.pypeline.PySubProcess.<init>(PySubProcess.java:145)
	at com.anylogic.libraries.pypeline.PySubProcess.<init>(PySubProcess.java:133)
	at com.anylogic.libraries.pypeline.PyCommunicator.initialize(PyCommunicator.java:518)
	at com.anylogic.libraries.pypeline.PyCommunicator.onStartup(PyCommunicator.java:933)
```

Explanation: This is similar to the first problem, in that it’s an issue with your system path/Communicator configurations, but is specific to Windows. 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, et cetera.

Solution: Please see the solution to the first 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".

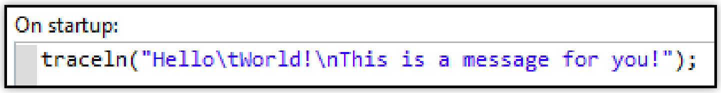


```
1 error(s)
Description
Invalid escape sequence (valid ones are \b \t \n \f \r \" \' \\ )
```

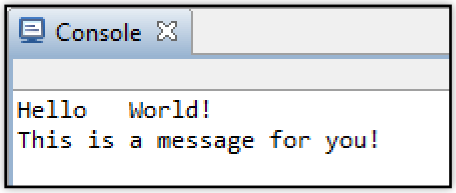
Explanation: In Java, the backslash (\) is a special character that is used to "escape" other characters in order to represent certain non-alphanumeric characters. One example of this is if you want a string that has a double quote character in it. For example:

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

This also applies to whitespace characters, such a tab (`\t`) or new line (`\n`). For example, running the code shown in the first image below produces the text shown in the second image below:



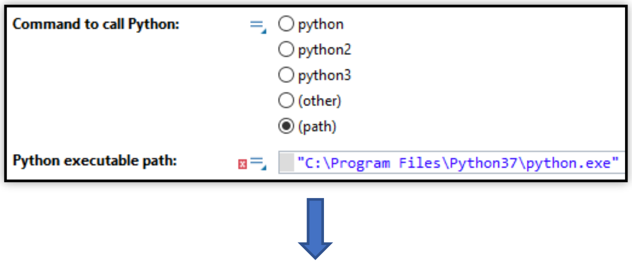
On startup:
```
traceln("Hello\tWorld!\nThis is a message for you!");
```



```
Hello	World!
This is a message for you!
```

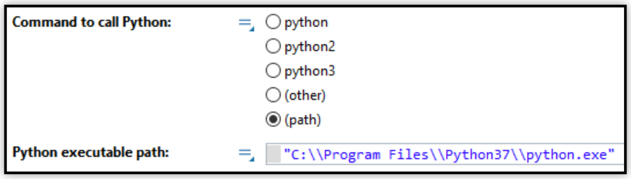
What this error message is referring to is that a single backslash was used within a string with an invalid character following it. If you have a string that starts with `"C:\Program Files"`, it throws the error because it doesn't recognize the character `"\P"`. In this case, it's trying to interpret what is meant as a literal-P as if it were one of the "special" characters.

Solution: Most likely, you are on a Windows machine and have the command to call Python set to "(path)" with a custom path specified. Simply add an extra backslash to each backslash.



Command to call Python: ○ python ○ python2 ○ python3 ○ (other) [✓ (path)]

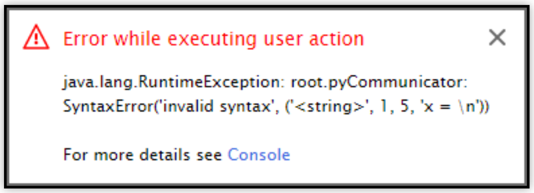
Python executable path: `"C:\Program Files\Python37\python.exe"`



Command to call Python: ○ python ○ python2 ○ python3 ○ (other) [✓ (path)]

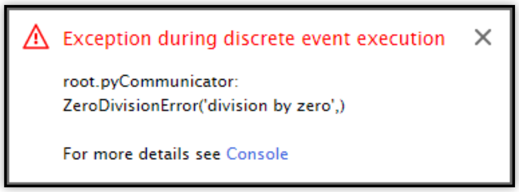
Python executable path: `"C:\\Program Files\\Python37\\python.exe"`

Problem: While running your model (at any point), you receive an exception or error with some other type of exception or error mentioned.


Error while executing user action

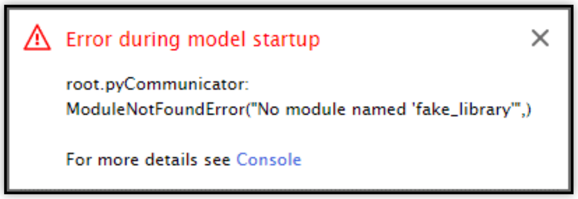
```
java.lang.RuntimeException: root.pyCommunicator:
SyntaxError('invalid syntax', ('<string>', 1, 5, 'x = \n'))
```

For more details see Console


Exception during discrete event execution

```
root.pyCommunicator:
ZeroDivisionError('division by zero',)
```

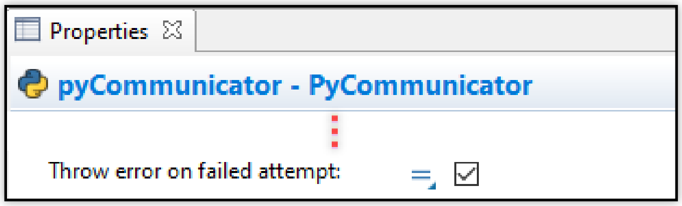
For more details see Console


Error during model startup

```
root.pyCommunicator:
ModuleNotFoundError("No module named 'fake_library'",)
```

For more details see Console

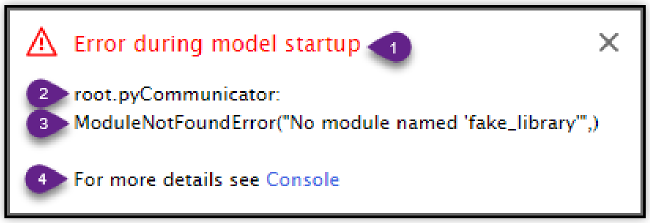
Explanation: This error appears because the option for "Throw error on failed Attempt" in the Communicator's properties is enabled (shown in the image below) and one of the commands to Python had failed.



pyCommunicator - PyCommunicator

Throw error on failed attempt: ✓

As these types of messages popup whenever any exception or error happens in AnyLogic (i.e., not related to Pypeline), it's good to know how to interpret these. Using the last example error shown above, take a look at the different parts of it. The image has been copied below and annotated, with the callouts referenced by the list below it:


Error during model startup

```
root.pyCommunicator:
ModuleNotFoundError("No module named 'fake_library'",)
```

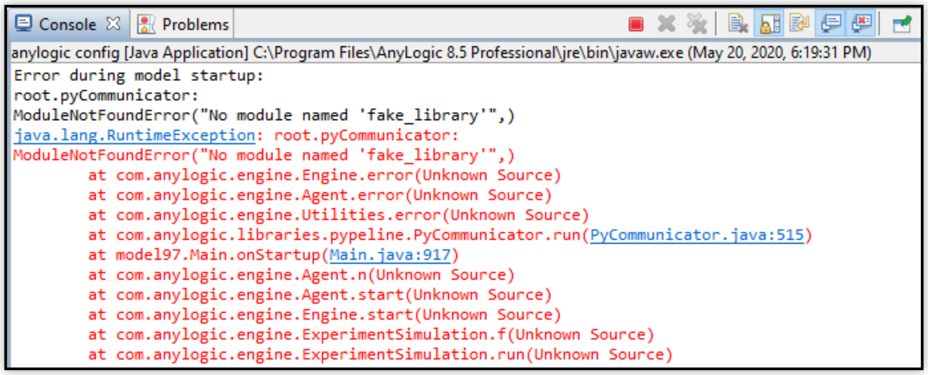
For more details see Console

1. This is a message from AnyLogic telling you the type of moment that the exception or error occurred at.

2. This is another message from AnyLogic telling you where the problem originated from. Specifically, it consists of the agent – in this case *root*, which refers to your top-level agent (typically Main) – and the name of the object – in this case *pyCommunicator*. You may also see the specific type of exception listed before this (as seen in the first example image above).

3. This is a message that provides more details about the exception or error regarding what went wrong. For Pypeline, this is the message received from Python as the feedback of a command being executed that fails. In other words, if the `isSuccessful()` command returns false, then then this text will be the output of the `getFeedback()` function.

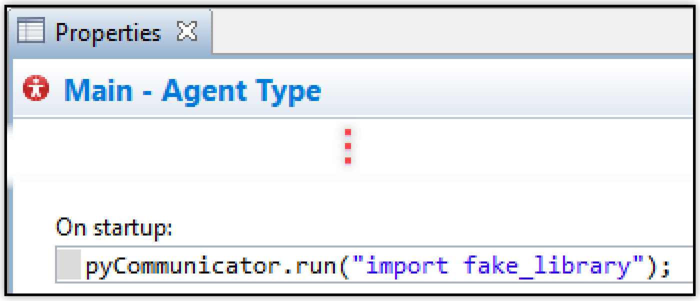
4. This is a reference to the AnyLogic console; for this example, the console is shown in the image below:



```
anylogic config [Java Application] C:\Program Files\AnyLogic 8.5 Professional\jre\bin\javaw.exe (May 20, 2020, 6:19:31 PM)
Error during model startup:
root.pyCommunicator:
ModuleNotFoundError("No module named 'fake_library'",)
java.lang.RuntimeException: root.pyCommunicator:
ModuleNotFoundError("No module named 'fake_library'",)
        at com.anylogic.engine.Engine.error(Unknown Source)
        at com.anylogic.engine.Agent.error(Unknown Source)
        at com.anylogic.engine.Utilities.error(Unknown Source)
        at com.anylogic.libraries.pypeline.PyCommunicator.run(PyCommunicator.java:515)
        at model97.Main.onStartup(Main.java:917)
        at com.anylogic.engine.Agent.n(Unknown Source)
        at com.anylogic.engine.Agent.start(Unknown Source)
        at com.anylogic.engine.Engine.start(Unknown Source)
        at com.anylogic.engine.ExperimentSimulation.f(Unknown Source)
        at com.anylogic.engine.ExperimentSimulation.run(Unknown Source)
```

While it can be daunting to try to understand what is going on, there is some useful information here to figure out where in the model that this error occurred. This block of red text is called a "stack trace" and describes the events leading up to the exception or error. Most of the lines are related to the AnyLogic engine, but one line that stands out is related to the `onStartup` function of the Main agent. This confirms annotation 1 shown above, which states the problem happened when the model started up.

If you were to take a look at the "*On startup*" field of Main, you would see the problematic code shown in the following image:



On startup:

```
pyCommunicator.run("import fake_library");
```

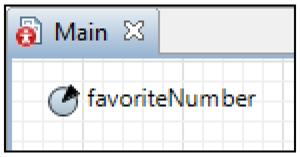
As the message from Python – `ModuleNotFoundError("No module named 'fake_library'",)` – states, this command failed because the library *fake_library* was not found.

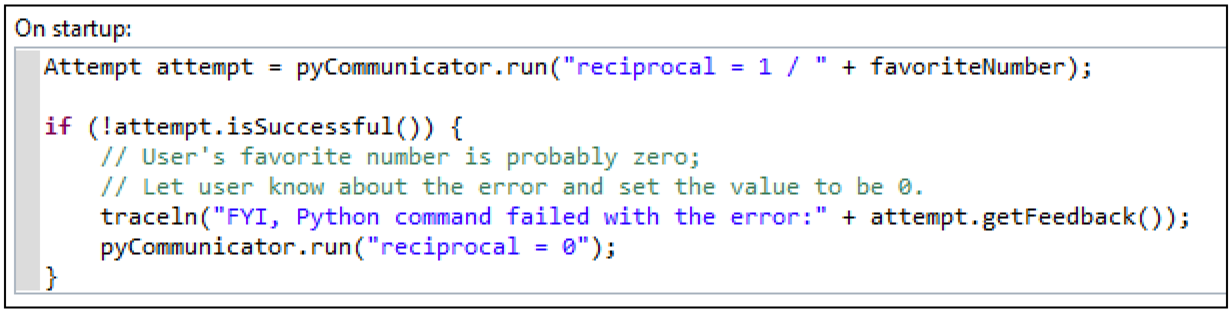
Solution: As this problem originates from your Python code, there is no one solution. However, if you do not want the entire model to come to a halt if an error occurs in Python and you instead want to handle it from AnyLogic, simply uncheck the option for "*Throw error on failed attempt*". One use-case for this is if you want to have fallback logic.

An example is shown in the images below. If the parameter *favoriteNumber* is set to zero, the first `run` function call will return a *ZeroDivisionError* in Python. In other words, the Communicator is telling Python to run the code: `reciprocal = 1 / 0` – code which will return the aforementioned error when executed.

Instead of halting the entire AnyLogic model when this happens, it is handled by additional code that executes after the attempt to the `run` function call. Specifically, it first checks to see whether the attempt was successful. If not, it then lets the user know this happened and then sets the variable to a fallback value (0).

If this is still unclear, create a new AnyLogic model and play around with it yourself. Additionally, the demo model called "Demo – Basic Functions" shows an example of this.



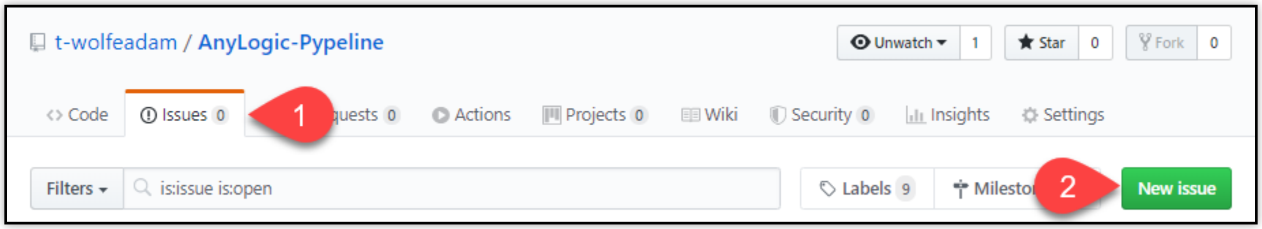


On startup:

```
Attempt attempt = pyCommunicator.run("reciprocal = 1 / " + favoriteNumber);

if (!attempt.isSuccessful()) {
    // User's favorite number is probably zero;
    // Let user know about the error and set the value to be 0.
    traceln("FYI, Python command failed with the error:" + attempt.getFeedback());
    pyCommunicator.run("reciprocal = 0");
}
```

If you have further problems, please file them in the "Issues" tab on the Pypeline GitHub repository.


t-wolfeadam / AnyLogic-Pypeline
Unwatch 1
Star 0
Fork 0
Code
Issues 0
1
quests 0
Actions
Projects 0
Wiki
Security 0
Insights
Settings
Filters
is:issue is:open
Labels 9
Milestor
2
New issue