# ADAMS

**A**dvanced **D**ata mining **A**nd **M**achine learning **S**ystem

Module: adams-jython




Peter Reutemann

January 8, 2020

©2009-2015



THE UNIVERSITY OF
WAIKATO
*Te Whare Wānanga o Waikato*



Except where otherwise noted, this work is licensed under
**http://creativecommons.org/licenses/by-sa/4.0/**

# Contents

**1 Introduction** **7**

**2 Writing actors** **9**
2.1 Superclass and wrapper . . . . . 9
2.2 Implementation . . . . . 10
2.3 Parameters . . . . . 11

**3 Writing conversions** **13**
3.1 Superclass and wrapper . . . . . 13
3.2 Implementation . . . . . 14
3.3 Parameters . . . . . 14

**Bibliography** **15**

# List of Figures

# Chapter 1

# Introduction

Developing a new actor for ADAMS is not hard. But it can take too long, if you simply want to test something: writing the code, compiling, packaging and, finally, executing it. Dynamic languages that run in the Java Virtual Machine (JVM), like Jython ([2]), come in handy. Here, you simply have to write the code and then execute it. There is no need to compile or package anything.

Jython's syntax is quite different from the Java one. Here is an example, the following Java class:

```
import java.util.Vector;
public class FunkyVector extends Vector {
  public String toString() {
    String result = "Funky output: ";
    result += super.toString();
    return result;
  }
}
```

Looks like this as a Jython script:

```
import java.util.Vector as Vector
class FunkyVector(Vector):
    def toString(self):
        result = "Funky output: "
        result.join(Vector.toString(self))
        return result
```

For more details, you might want to check out the Jython [1] and Python [2] tutorials.

[1] http://www.jython.org/docs/tutorial/
[2] http://docs.python.org/tutorial/

# Chapter 2

# Writing actors

Writing a Jython actor works similar to writing a regular ADAMS actor in Java.

## 2.1 Superclass and wrapper

First, you create an empty text file for you new actor[1]. Second, you choose what superclass you want to derive it from:

- **Standalone** – `adams.flow.standalone.AbstractScript`
- **Source** – `adams.flow.source.AbstractScript`
- **Transformer** – `adams.flow.transformer.AbstractScript`
- **Sink** – `adams.flow.sink.AbstractScript`

This also determines, which ADAMS wrapper actor you need to use for executing your external script:

- **Standalone** – `adams.flow.standalone.Jython`
- **Source** – `adams.flow.source.Jython`
- **Transformer** – `adams.flow.transformer.Jython`
- **Sink** – `adams.flow.sink.Jython`

You simply use your script file as the `scriptFile` property and now you only have to write the actual code.

[1] You can also simply use the *inlineScript* option if you don't want to use a file

## 2.2 Implementation

As for writing your code, you merely have to implement all the abstract methods from your `AbstractScript` superclass. The following code shows a minimalistic *transformer*, which accepts and generates `Integer` objects. In the `doExecute()` method, as it stands, it does not do anything with the incoming data, it merely forwards a new `Token` with the data it received.

```
import adams.flow.core.Token as Token
import adams.flow.transformer.AbstractScript as AbstractScript
import java.lang.Class as Class

class SimpleTransformer(AbstractScript):
    def __init__(self):
        AbstractScript.__init__(self)

    def globalInfo(self):
        return "My simple transformer"

    def accepts(self):
        return [Class.forName("java.lang.Integer")]

    def generates(self):
        return [Class.forName("java.lang.Integer")]

    def doExecute(self):
        self.m_OutputToken = Token(self.m_InputToken.getPayload())
        return None
```

A slightly more complex version computes the square of the incoming integer:

```
...
def doExecute(self):
    input = self.m_InputToken.getPayload()
    self.m_OutputToken = Token(input * input)
    return None
...
```

## 2.3 Parameters

Of course, most of the actors that you will write, will require some form of parametrization. Instead of defining options in the script itself, the ADAMS wrapper actor takes on the role of providing parameters. Each of the `Jython` wrapper actors has a property called `scriptOptions` which takes a blank-separated list of key-value pairs ("key=value").

These options are available in the Jython script via the `getAdditionalOptions()` method, returning an `adams.flow.core.AdditionalOptions` container object. This container object offers retrieval of the options via their key:

- `getBoolean(String)` and `getBoolean(String,Boolean)`
- `getInteger(String)` and `getInteger(String,Integer)`
- `getDouble(String)` and `getDouble(String,Double)`
- `getString(String)` and `getString(String,String)`

The second method listed allows you to specify a default value, in case the option was not supplied.

Assuming that we require an additional option called `add`, we can use this parameter to add to our incoming integer value in order to generate output:

```
..
def doExecute(self):
    input = self.m_InputToken.getPayload()
    self.m_OutputToken = Token(input + self.getAdditionalOptions().getInteger("add", 1))
    return None
..
```

# Chapter 3

# Writing conversions

Just like with Jython actors, Jython conversions have follow the same principle, by being implemented as regular conversion schemes in ADAMS.

## 3.1 Superclass and wrapper

First, you create an empty text file for you new conversion[1]. Second, you use the following superclass to derive your Jython conversion from:

```
adams.data.conversion.AbstractScript
```

The external script is executed using the following wrapper conversion:

```
adams.data.conversion.Jython
```

You simply use your script file as the `scriptFile` property and now you only have to write the actual code.

[1] You can also simply use the *inlineScript* option if you don't want to use a file.

## 3.2 Implementation

As for writing your code, you merely have to implement all the abstract methods from your `AbstractScript` superclass. The following code shows a minimalistic conversion, which accepts and generates `Double` objects. In the `doConvert()` method, it merely divides the incoming doubles by 100.

```
import adams.data.conversion.AbstractScript as AbstractScript
import java.lang.Class as Class

class SimpleConversion(AbstractScript):
    def globalInfo(self):
        return "Just divides the incoming doubles by 100."

    def accepts(self):
        # very in-elegant, but works
        # http://www.prasannatech.net/2009/02/class-object-name-java-interface-jython
        return Class.forName("java.lang.Double")

    def generates(self):
        # very in-elegant, but works
        # http://www.prasannatech.net/2009/02/class-object-name-java-interface-jython
        return Class.forName("java.lang.Double")

    def doConvert(self):
        return self.m_Input / 100
```

## 3.3 Parameters

As in how to use parameters, see section 2.3 as it works the same way as for actors.

# Bibliography

[1] *ADAMS* – Advanced Data mining and Machine learning System
`https://adams.cms.waikato.ac.nz/`

[2] *Jython* – Python for the Java Platform
`http://www.jython.org/`