pandoc-source-exec examples

Sebastian Höffner

March 2017

Preamble

Compile this document as follows:

```
pandoc --filter pandoc-source-exec \
       --highlight-style tango \
       -o example.pdf example.md
```

Usage

To execute code, add the class `exec` to your code:

````
```{ .python .exec }
print('Hello World')
```
````

This results in:

```
print('Hello World')
```

Output:

```
Hello World
```

You can also supply the interpreter keys in the `runas` argument:

````
```{ .python .exec runas=python2 }
print 'Hello World'
```
````

Or you can simply make up your own command:

````
```{ .exec cmd='/usr/bin/env python2 -c' }
print 'Hello World'
```
````

Examples

No execution

```
a = 3 + 5
print(a)
```

Simple execution

Using: `{ .python .exec }`

```
print('Hello World')
```

Output:

```
Hello World
```

Advanced execution

Known interpreter `{ .python .exec runas=python2 }`:

```
print 'Hello World'
```

Output:

```
Hello World
```

Custom interpreter `{ .exec cmd='/usr/bin/env python2 -c' }`:

```
print 'Hello World'
```

Output:

```
Hello World
```

Or ruby `{ .exec cmd='/usr/bin/env ruby -e' }`:

```
puts 'Hello World!'
```

Output:

```
Hello World!
```

Errors

`stderr` is piped to `stdout`, so that errors can also be shown.

Using: `{ .python .exec }`

```
print('Hello
```

Output:

```
  File "<string>", line 1
    print('Hello
                ^
SyntaxError: EOL while scanning string literal
```

File execution

Using: `{ .python .exec file='example.py' }`

File: `example.py`

```
import math

a = 3
b = 4
c = math.sqrt(a ** 2 + b ** 2)

print(c)
```

Output:

```
5.0
```

File without execution

Using: `{ .python file='example.py' }`

File: `example.py`

```
import math

a = 3
b = 4
c = math.sqrt(a ** 2 + b ** 2)

print(c)
```

Interactive execution

Using: `{ .python .exec .interactive }`

Interactive code will also be detected if the code block starts with `>>>`.

Note: This only works with python code so far, a custom command is not possible.

```
>>> a = 5 + 4
>>> 9 == a
True
>>> def h():
...     print('Hello World')
...
>>> h()
Hello World
```

API

The following keywords (classes denoted by a prefixed `.`, attributes with a following `=`) exist:

cmd= Allows to specify a custom interpreter command to execute the code. For example, to run ruby code one could use `cmd='ruby -e'`.

.exec Executes the following code cell according to the specified language. By default, it is only `echo`ed.

file= Replaces the code cell with content from the specified file. This searches recursively for files matching the pattern, so if you use `file=code.py` but your code is in fact in `src/code.py` it will still be found. Specify the full path to avoid ambiguities.

.interactive Executes the code as if it was inserted into an interactive session, returns results inline into the original code block. Only works for python code so far.

`runas=` Executes code with the specified executor, e.g. `python2` to distinguish it from `python` which defaults to `python3`. Can be overwritten by specifying `cmd=`.

`.hideimports` Hides import statements in output. Currently only supported for Python.

Supported languages

To be used with `runas=`, if it does not already match the language identifier:

- default
- perl
- php
- python
- python2
- python3
- ruby

default

```
default
```

Output:

```
default
```

perl

```
print 'perl';
```

Output:

```
perl
```

php

```
echo 'php';
```

Output:

```
php
```

python

```
print('python')
```

Output:

```
python
```

python2

```
print 'runas=python2'
```

Output:

```
runas=python2
```

python3

```
print('runas=python3')
```

Output:

```
runas=python3
```

ruby

```
puts 'ruby'
```

Output:

```
ruby
```

Removing imports

Removing imports affects only the final code rendering, not the execution.

````
```{ .python .exec .hideimports }
import statistics


print(statistics.mean([1, 2, 3])
```
````

```
print(statistics.mean([1, 2, 3]))
```

Output:

```
2
```