Script Markdown User Manual

Script Markdown Utility

This guide is NOT Confidential

This manual describes the *Script Markdown Utility*, its features, purpose and more. I've packed it with examples too, so hopefully after you read it, you'll know all you need to know about how to use it to create script markdown documents quickly, easily and most important, efficiently. **Enjoy!**

NOTE: Additional samples that might be worth reviewing are those in the unit test code located here: ***(../tests/in/*.md)***. Keep in mind that the unit test code is meant to stress test the parser as well as the limits and edges of the syntax of smd, so in some cases it might be confusing or even contradictory to what the user guide covers!

Revision: *0.4.2* (20200923 @ 07:18:09)

Ken Lowrie
512-867-5309
me@mydomain.com

Notes to Reviewer

Please send Ken Lowrie any and all feedback, preferably by marking up the PDF using embedded comments. If you edit the PDF text, do so inline using comment boxes, or if you edit the text directly, change the color and/or font size so I can easily find it. Within different versions of this proposal, **additions are marked like this** and ~~deletions are marked like this~~

Table of Contents - SMD User Guide

smd - The Script Markdown Processor

Welcome to the user manual for **smd**, the **Script Markdown** Processor that can take plain text files written in a specialized markdown syntax and turn them into rich HTML documents. This guide will take you through installation and setup of **smd**, and then show you how you can use it to create all sorts of interesting and cool HTML projects. So let's get going!

Setting up smd

There are two approaches to installing **SMD** on your system; either using **pip** or **pipenv**. This chapter will cover both methods, and you can choose which is better for you.

Please take note of the following caveats before continuing with the setup process.

Minimum Required Python Version

> **smd** requires Python 3.7.3 or later! If you are running an older version, you either have to upgrade or install a virtual environment with something newer in order to run **smd**.

Tk (tkinter) requirement for ismd

> If you will be using **ismd** (**Interactive Script Markdown**), the version of Python you will be using must have been built with **tkinter** (**Tk**). Versions from ActiveState will have this, but if you use some other version, for example, if you use **pyenv** to manage the version of Python on your system, you must set the compiler flags *prior* to doing the pyenv install.

If you did not have these flags set when you installed Python using **pyenv**, and you want to use **ismd**, you will have to *uninstall* the current version, and reinstall it (with the environment variables set). Here is a link https://tkdocs.com/tutorial/install.html that will provide the information you need to get this done, although take a look at this Stackoverflow Question and Answer for the steps needed to solve this issue.

Okay, let's get started.

Installing with pip

The easiest way to install **smd** on your system is to use the default package installer for Python or **pip**. From a terminal window on your machine, change to the root directory of your cloned repository and type:

```
$ pip install .
```

> note default content

To install in edit mode with pip

```
$ pip install -e .
```

> HINT: If you change your mind about how **smd** is installed, simply run ***pip install .*** or ***pip install -e .*** to change it. When pip detects that **smd** is already installed on your system, it will uninstall it first, and then install it in whatever mode you specify.

One drawback to installing **smd** using the **pip install** method is that the package and its dependencies will be installed globally, which could interfere with other packages installed on your system. Also, in order to minimize the number of dependencies that are installed using **pip**, three other packages that are needed by **ismd**, namely **watchdog**, **selenium** and **bottle**, are not automatically installed.

If you will not be using **ismd**, then there is nothing extra required. **smd** and **smdparse** both run without these additional packages. Go ahead and skip down to the smd vs markdown section to continue.

Additional pip install steps if using ismd

If you will be using **ismd**, then you must either install these packages manually, or alternatively, you can use the **pipenv** method described in the next section, which *will* install the extra packages automatically. To install them manually using **pip**, use these commands in your terminal window:

Install additional package dependencies using pip

```
$ pip install selenium
$ pip install bottle
$ pip install watchdog
```

After completing the pip installs for **selenium**, **bottle** and **watchdog**, the last thing you need to configure is a web driver for **selenium** for the browser you will be using. Skip down to Installing a webdriver for selenium for the details.

Installing with pipenv

---

pipenv is a much better way to install **smd**. It will create a virtual environment for running **smd** on your machine, preventing installation of the various site-packages in a global manner, the way **pip** does. This, in turn, isolates **smd** to a private environment for use, testing and/or evaluation.

> NOTE: If you do not have pipenv installed, navigate to https://docs.python-guide.org/dev/virtualenvs/ for information on how to do that.

The **Pipfile** and **Pipfile.lock** files are provided in the root directory of your cloned repository. Simply navigate to that directory on your local machine and type:

```
$ pipenv install
```

to create the runtime environment for launching smd on your machine. Once it finishes, type:

Starting ismd after pipenv install

```
$ pipenv shell

# To Launch ismd and display help
$ smd -h

# or Launch smdparse and display help
$ smdparse - h

# or Launch ismd and display help
$ ismd - h
```

This will launch **smd**, **smdparse** and/or **ismd** within the virtual environment that was created. Alternatively, you can use **pipenv run ...** to execute the code inside the virtual environment without starting the **pipenv shell**:

Using pipenv run to execute Python code in virtual environment

```
# Launch ismd
$ pipenv run smd -h

# or Launch smdparse
$ pipenv run smdparse - h

# or Launch ismd
$ pipenv run ismd - h
```

By default, **smd** is installed *edit* mode (**pipenv install -e .**). This points the installation of the **smd** package to the current cloned repository directory instead of copying it to your site-packages installation directory; useful if you plan on making changes and don't want to reinstall each time you make a change.

If you don't want **smd** installed in *edit* mode, you can ...

```
$ pipenv uninstall smd
```

... followed by ...

```
$ pipenv install .
```

... from the **root directory of your cloned repository** to change how it is installed.

Installing a webdriver for *selenium*

In order for **selenium** to be able to control your browser, it requires a special driver called a **webdriver**. You can read more about Selenium with Python here, and specifically about Selenium drivers here.

Each browser driver should provide any required setup steps on their site. For the most part, however, it's just a matter of copying the driver into the PATH, so that **selenium** can find it. So, navigate to one of the following links and get the driver for your browser.

If you are using Google Chrome, you will need to get the Chrome Webdriver version.

If you are using Mozilla Firefox, you need to get the Firefox Webdriver version.

If you are using Apple Safari, the driver ships with **macOS**. Navigate to enabling the Safari Webdriver where you can read about setting up and configuring the Safari driver.

> At the time of writing this documentation, the **Safari** webdriver has a feature called the **glass pane** which makes it not very useful for being an **ismd** monitor. I've been unable to figure out a way to disable the glass pane without stopping the automation, which prevents you from interacting with the browser window to do simple things such as scrolling down... DOH! Gee, thanks Apple. Good news though, both Chrome and Firefox work GREAT! So you might want to use one of them instead, at least when using the monitor feature of **ismd**. If and when I find a solution, I will fix the code and update the documentation.

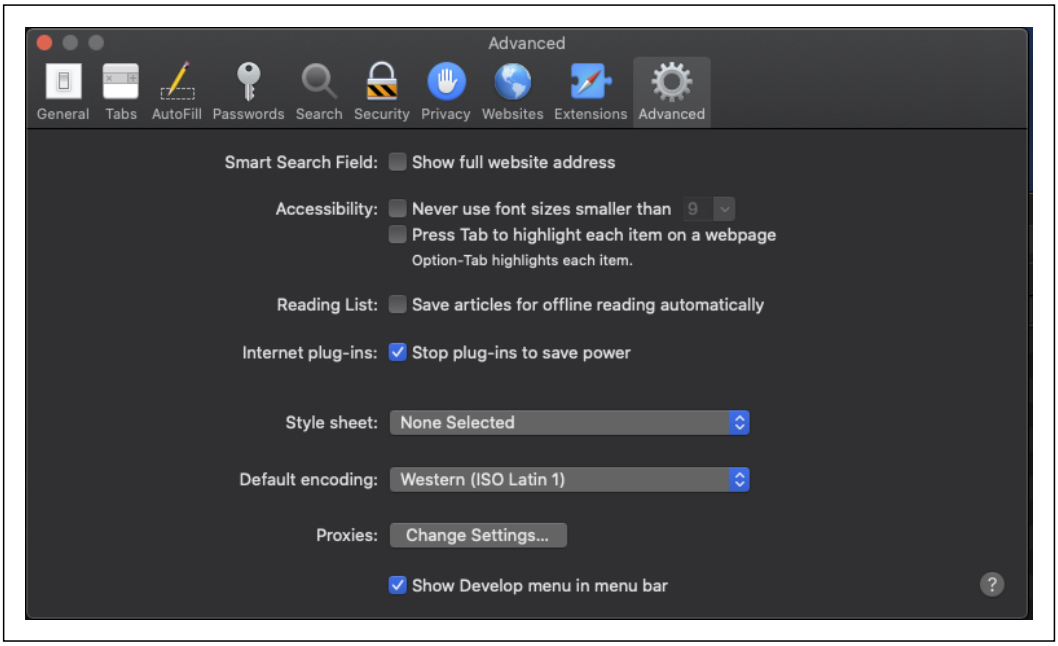
Here are a few notes that might help you get things working if you have never used **selenium** before.

First, on newer versions of **macOS**, the driver may require that you grant it the appropriate rights to run the driver on the platform, depending on whether or not the driver has been notarized. Usually, this can be accomplished by simply running the driver once. For both the Chrome Webdriver and the Firefox Webdriver versions, I simply ran the executable in a terminal window, and when I got the prompt from **macOS**, I allowed it to run. Fortunately, this is only required one time.

On Safari, you need to:

```
$ safaridriver --enable
```

You may also need to **Show Develop Menu**:



As well as **Develop | Allow Remote Automation**:

In order to get the Safari Webdriver to work. And by "*to work*", I mean "*show the first part of your markdown document only*". 😜.

Using smd with OBS Studio

---

smd can be used to create content (both static and dynamic) for use in **OBS Studio**. You will likely use either **smdparse** to generate a static file, or **ismd** to create an endpoint on the **localhost** that you can use with **OBS Studio**. Let's see how this can be done.

> **NOTE:** This section is jumping ahead just a bit, so if it seems confusing, don't worry. It does assume you are familiar with **OBS**, so if that isn't the case, then you may want to skip it for now.
>
> In addition, it assumes you have successfully installed and configured **smd** with either **pip** or **pipenv**, as described in the previous sections.

Here is how you can create a dynamic clock overlay for OBS Studio:

Creating content for OBS Studio

```
# If you installed smd with pipenv, start with:
$ cd <root_of_your_smd_repo>
$ pipenv shell

# Now move to the samples/obs/clock directory
(smd) $ cd samples/obs/clock

# this next command will create the virtual env and run ismd
(smd) $ ismd -nd -f clock.md -m endpoint hostgui

# ismd will parse clock.md, create an endpoint, and keep it live until you press CTRL-C (^C)
```

Alternatively, if the code is not changing (i.e. you've completed development/debugging), you can simply generate a static local file using **smdparse** and load that inside OBS. Assuming you are in the **samples/obs/clock** directory **AND** you are in a **pipenv** shell if you installed with **pipenv**:

```
(smd) $ smdparse -nd -f clock.md
```

Loading the *clock* HTML inside OBS

To access the **clock** sample inside OBS, add a **browser** to a **scene**, and use one of the following URLs (depending on whether you used **ismd** or **smdparse** to create the content):

URL syntax for accessing the clock sample

```
If you are running using the ismd endpoint then use:
localhost:8080/smd/clock/html/clock.html

If you used smdparse to generate the raw HTML then use:
file:///samples/obs/clock/html/clock.html
```

Once again, I realize we jumped ahead a bit with this sample on using **ismd** and **smdparse** since we have yet to cover them, but I figured showing a somewhat more real-world example would help illustrate the kind of things you can do with **smd**! One last thing to cover before we move on is how you can customize the **pipenv** **Pipfile** to add additional packages you want to use inside your **smd** virtual environment.

Creating the Pipenv/Pipenv.lock files the first time

If you want to create your own custom configuration for **pipenv**, you can do that with the following commands:

Creating the Pipfile and Pipfile.lock

```
$ pipenv install selenium
$ pipenv install watchdog
$ pipenv install bottle
$ cd <root_of_smd_cloned_repo>
$ pipenv install -e .

# Now add more pipenv install commands for your other packages
```

Technically, you don't need the first four steps above, since the **Pipenv** file already has everything you need for **smd**. You could just **pipenv install** your own packages while sitting in the root directory of your cloned repository, and they will be added to the existing configuration file. Subsequently, whenever you create a new virtual environment using **pipenv install** or **pipenv run**, it will add any additional packages automatically.

smd vs markdown - Repository README.md

Just below, I will import the README.md from the root directory of this repository. That file uses a more standard markdown syntax, and if you review the rendered output, you will see where certain things do not expand as the syntax is not supported by **smd**. The most obvious are the inline and reference links, although this is a rather simple example, not intended to show all differences between **markdown** and **smd**.

smd

Script Markdown Processor

Welcome to the Script Markdown Processor. This document should help you get the project up and running. If you have questions, or would like to provide feedback and/or to report a bug, feel free to contact the author, Ken Lowrie, at www.kenlowrie.com.

Attributions

This project was originally inspired by John Gruber's [Markdown](https://daringfireball.net/projects/markdown/), the [Python Markdown](https://github.com/Python-Markdown/markdown) Project, and Mark Boszko's [AVScriptFormat](https://github.com/bobtiki/AVScriptFormat) BBEdit plug-in. That resulted in [avscript](https://github.com/kenlowrie/avscript), the predecessor to **smd**.

Installing this app on your system

For instructions on how to install this package on your system, refer to the user guide located in the **docs/export** folder of the repository. Open the file **userguide.html** in your browser.

Python Version

SMD requires Python 3.7.3 or later, so if you are running anything older than that, you will either have to upgrade or install a virtual environment with something newer in order to use this package.

Why Script Markdown Processor?

I wanted a way to dynamically create browser content for use in OBS Studio, and this seemed like a good starting point to leverage other code I had. It grew from there...

Getting Started with the App

User documentation is available via the "docs/userdocs.md" file, which can be rendered on the fly using smd, smdparse or ismd. A static version has been exported into the **docs/export** folder for convenience.

Summary

This concludes the *README.md* for the Script Markdown (smd) Processor Python Package.

As you can see, the simple markdown including headings e.g. **#**, bold face **** and italics **** are parsed and render the same with **smd** as they do with standard **markdown**. As you will soon see, however, **smd** will do much, much more.

Table of Contents

smd Package

Before diving into the primary interfaces of the **smd** package, let's establish a little context first.

> This chapter assumes you have successfully installed **smd** using one of the methods documented in the previous chapter.

We will start by providing an overview of the repository layout, followed by a discussion of the primary command line utilities, and finally the startup & shutdown processes of the primary command line interfaces. This will establish enough background information about **smd** that the discussion on the command line parameters will make more sense.

Repository Layout

smd repository structure

```
root
   ├─▸ debug - Python scripts used for debugging smd, smdparse and ismd
   │     ├─▸ debug.py - Python script used for debugging smd
   │     ├─▸ debug-1.py - Python script used for debugging smdparse
   │     └─▸ debug-2.py - Python script used for debugging ismd
   │
   ├─▸ docs - user guide, etc.
   │     ├─▸ export - a rendered version of the user documentation in HTML format
   │     ├─▸ import - imports used by the documentation
   │     ├─▸ section - chapters that make up the user documentation
   │     └─▸ userdocs.md - the top level user documentation file
   │           └─▸ Parsed using smdparse -f userdocs.md -c -d export to create
docs/export/userdocs.html
   │
   ├─▸ samples - example smd files
   │     └─▸ obs - sample OBS Studio files
   │           ├─▸ clock - displays an updating digital clock via an OBS Browser link
   │           └─▸ border - displays an opaque border via an OBS Browser link
   │
   ├─▸ smd - Python [smb.b] package root
   │     ├─▸ smd.py - Script Markdown command line interface
   │     ├─▸ smdparse.py - Script Markdown Parser command line interface
   │     ├─▸ ismd.py - Interactive Script Markdown command line interface
   │     ├─▸ core - [smb.b] classes
   │     ├─▸ css - default CSS files
   │     └─▸ import - smd built-ins
   │           ├─▸ avs - smd AV script built-ins
   │           ├─▸ def_html.md - default HTML open tags
   │           ├─▸ def_head.md - default HEAD open tags
   │           ├─▸ def_body.md - default BODY open tags
   │           ├─▸ def_bodyclose.md - default BODY closing tags
   │           ├─▸ def_close.md - default HTML closing tags
   │           └─▸ builtins.md - main system builtins file
   │                 ├─▸ common.md - generic builtins
   │                 ├─▸ defaults.md - default values used in def_*.md files
   │                 ├─▸ code.md - @code namespace builtins
   │                 ├─▸ html.md - @html namespace builtins
   │                 └─▸ link.md - @link namespace builtins
   │
   ├─▸ src - Miscellaneous Python Source files
   │
   └─▸ tests - Unit tests for smd
         ├─▸ cmdline - Command Line Parameter unit tests
         ├─▸ in - individual unit test input files
         ├─▸ out - individual unit test master files (expected output)
         ├─▸ test.py - smd unittest command line interface (i.e. unit test runner)
         ├─▸ testdbg.sh - bash script that starts an interactive smd session on a unit
test
         └─▸ test2dbg.sh - starts an interactive smd session with both chrome and hostgui
windows
```

Primary components of smd

The primary components of **smd** consist of three (3) command line interfaces.

1. **smd** - The Script Markdown command line interface
2. **smdparse** - The Script Markdown Parser command line interface
3. **ismd** - The Interactive Script Markdown command line interface

> NOTE: If you used pipenv to install **smd**, be sure to either activate the virtual environment by typing **pipenv shell** or alternatively, type **pipenv run** in front of each **smd** command in the following examples.

smd command line interface

smd is the main command line interface, and the one that you can use to experiment with this language. Since you have successfully installed smd, go ahead and try it by entering the following command in a terminal window:

```
$ smd -nd
```

At this point, your terminal window is waiting for input. Type **Hello, world** and press the **Return** ↵ key on your keyboard.

```
$ smd -nd
Hello, world↵
Hello, world
^D
```

When you typed **Hello, world**, the parser simply echoed that input back to you. Congratulations, you have just created your first **smd** document!

To close **smd**, type **CTRL-D (^D)** (yes, macOS users, that's **CTRL-D** and not **CMD-D (⌘D)**). **CTRL-D** will send an EOF signal on the stdin input stream, which will cause **smd** to perform an orderly shutdown.

You can also terminate **smd** by pressing **CTRL-C (^C)**, however that results in the **SIG-INT** signal being sent, which will cause an unorderly shutdown (i.e. none of the closing tags will be written, etc.)

smdparse command line interface

smdparse is a higher level command line interface that sits atop **smd**. It imports the direct smd entry point, so it's using the exact same underlying code to parse your documents, however, it adds several useful options for automation. It also provides the foundation for the final major component, **ismd**.

Let's parse the **samples/hello/hello.md** sample file next using **smdparse**:

```
$ cd samples/hello
$ smdparse -f hello.md -c
$ open html/hello.html
```

The **open html/hello.html** command opened the generated **hello.html** file in your default browser so you can review it. Close the browser window when you are done.

ismd command line interface

ismd is an **interactive** version of **smd** that provides several different methods for viewing the output from **smd**. What's unique about it is that it monitors **all** of the underlying files that are processed by **smd** for changes, and when it detects a change, it automatically parses the document again, and updates the output monitors.

There are several different monitors currently supported by **ismd**: **chrome** (the default), which displays the output in a Chrome browser window; **firefox** and **safari**, which display the output in Firefox and Safari, respectively; **hostgui**, which displays the raw HTML output in a host window and finally **endpoint**, which creates an HTTP endpoint on the localhost that can be opened by any web browser.

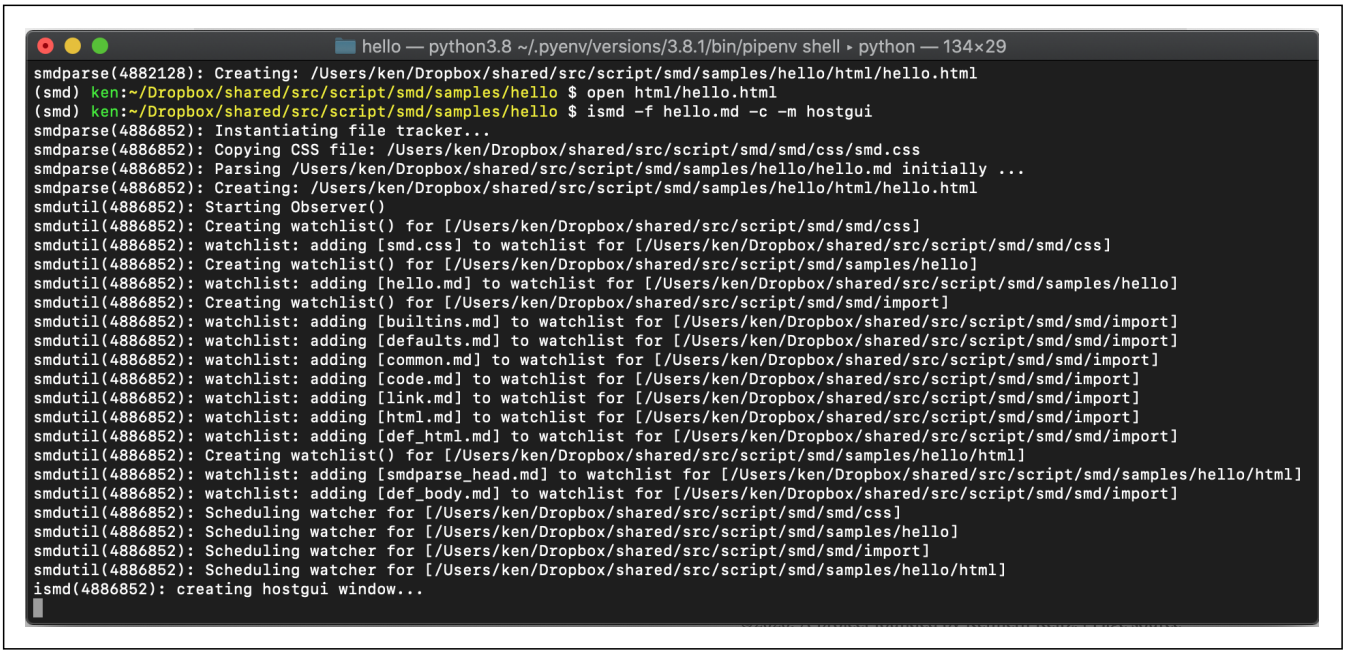
> NOTE: The **hostgui** monitor requires that your version of Python was compiled with the **Tk (tkinter)** support, and the browser monitors all require an additional webdriver before they will work. Be sure to review the Setup Chapter for the required information for using any of the **ismd** monitors.

Let's go ahead and parse the **samples/hello/hello.md** sample file next using **ismd** with the **hostgui** monitor:

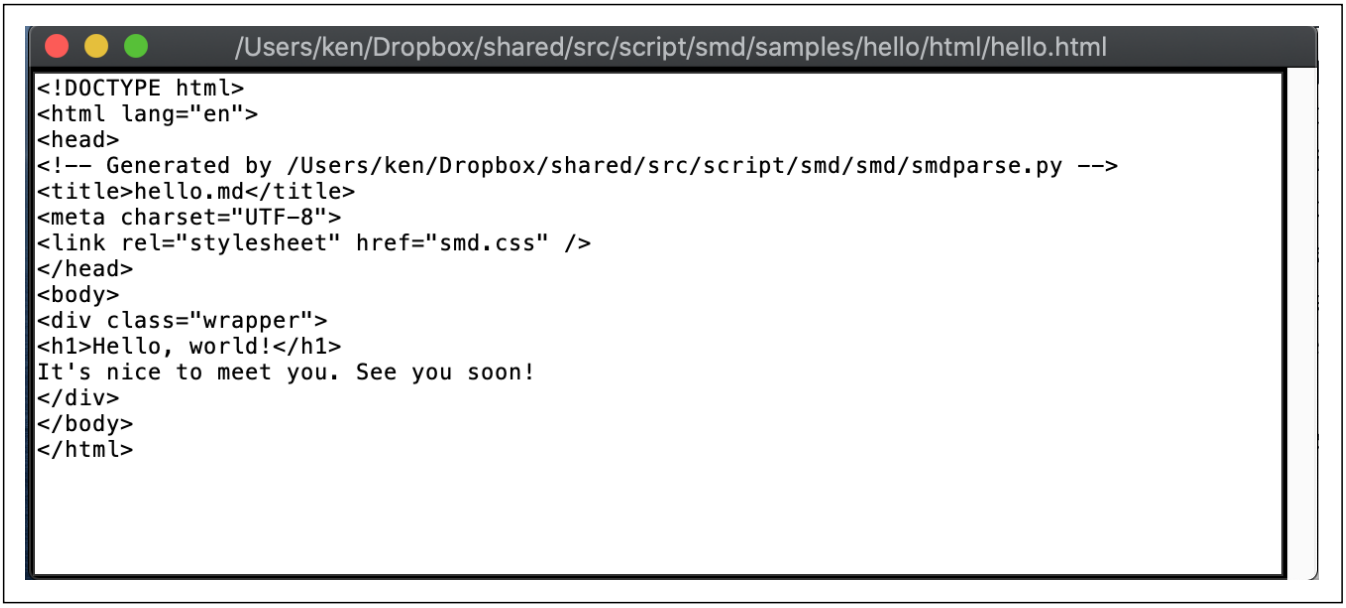
```
$ cd samples/hello
$ ismd -f hello.md -c -m hostgui
```

When you execute the **ismd** command, it will parse **hello.md**, then create a **Tk** window where it will display the contents of the parsed file. It will continue to monitor all files that are parsed to generate **hello.md**, and if any of them change, it will parse again and update the **hostgui** window. Experiment with that now, by changing the contents of **hello.md**, and saving it. You should see the window update with the new content each time you make a change and save the document. Here is what it looks like when you first run it:

The terminal window:



The **hostgui** window:



When you are done, be sure to type **CTRL-C** in the terminal window to terminate **ismd**.

Both **smdparse** and **ismd** and their command line options will be covered in more detail later in this chapter.

smd startup & shutdown overview

The following provides a high-level overview of the startup of **smd**. This is the basic flow when no command line parameters are passed to **smd**.

High level overview of startup

1. import builtins.md - which imports several other modules
    A. import defaults.md
    B. import common.md
    C. import code.md
    D. import link.md
    E. import html.md
2. import opening body markup
    A. import def_html.md
    B. import def_head.md

C. import def_body.md

// command prompt for interactive input unless -f filename or < filename (redirection from shell)
// <CTRL-D> ends the input ***(or EOF if -f filename or < filename (redirection from shell))***

3. import closing body markup
   A. import def_bodyclose.md
   B. import def_close.md

By reviewing the startup process, you can see that a number of the built in markdown files are preloaded when **smd** is started (assuming you did not pass any command line switches preventing that behavior). In addition, the default HTML markup files are loaded and dumped to the output stream, before it begins processing your input (either from the keyboard i.e. **stdin**, or from a file if **-f** or shell redirection is used). Once EOF is reached on the input stream, the remaining HTML markup files are read and dumped to the output stream, before it cleans up and performs an orderly shutdown.

Okay, now that we've provided enough context, let's take a look at the command line parameters for each of the major **smd** components.

smd command line parameters

The primary command line interface to **smd** is the Python module **smd.py**. This module contains the code to parse script markdown text files into HTML format. The command line parameters for **smd** are:

smd command line parameters

```
[-h]
[-f FILENAME]
[-ldb | -ndb]
[-lub | -nub]
[-nd] [-nohtml] [-nohead] [-nobody]
[-html HTML_NAME]
[-head HEAD_NAME]
[-body BODY_NAME]
[-bodyclose BODYCLOSE_NAME]
[-close CLOSE_NAME]
[-nu]
[-o [RAW_OUTPUT_FILE]]
```

General Parameters

Parameter	Description
-h *--help*	show help message and exit
-f FILENAME *--filename FILENAME*	the FILENAME that you want to parse. Default is **stdin** if not specified. You can also redirect **<** or pipe **|** the **stdin** stream when using **smd** in scripts.
-o [FILENAME] *--output-raw-data [FILENAME]*	write the raw data to output file. Default is: **None**. This is useful in certain debug situations to determine how far you are getting before encountering an issue. If you do not specify the FILENAME, it will default to **smd_rawdata.out**. This option writes out each input line read and returned to the main loop starting with the declaration of the **sys** variable and the importing of **builtins.md** then continuing with the processing of the input stream.
-nu *--no-user-files*	do not load any files from ***~/.smd***. Default is: **False**. Both the **builtins** and **document defaults** support the ability to override the system provided versions with user versions that are in the logged in user's home directory. This switch, when specified, prevents **smd** from loading any user-provided overrides.

Builtin Parameters

The *builtin parameters* control whether or not **smd** will process the default builtins stored in **builtins.md** during startup. There are potentially two sets of builtins that can be loaded, the system version located in **[sys.imports]/builtins.md** and the optional user version located in **[user.imports]/builtins.md**. The following command line parameters control how the system will process them.

Parameter	Description
-ldb *--load-default-builtins*	load default builtins during startup. Default is: **True**. This switch and it's counterpart **-ndb**, **--no-default-builtins** control whether or not **smd** will load **[sys.imports]/builtins.md** during startup. Specify one or the other, but not both.
-ndb *--no-default-builtins*	do not load default builtins during startup. Default is: **False**. See **-ldb**, **--load-default-builtins** for additional information.
-lub *--load-user-builtins*	load user builtins during startup. Default is: **True**. This switch and it's counterpart **-nub**, **--no-user-builtins** control whether or not **smd** will load **[user.imports]/builtins.md** during startup. Specify one or the other, but not both. Also note that **smd** will load the system builtins file first, and then the user builtins, which gives you the ability to override specific system builtins by simply redefining them during startup.
-nub *--no-user-builtins*	do not load user builtins during startup. Default is: **False**. See **-lub**, **--load-user-builtins** for additional information.

Document Default Parameters

Parameter	Description
-nd *--no-document-defaults*	do not load any document defaults during startup. Default is: **False**. This switch is a shorthand method of specifying **-nohtml**, **-nohead** and **-nobody** all at once. In addition, it is mutually exclusive to **-html**, **-head**, **-body**, **-bodyclose** and **-close**, so you cannot specify this along with any of those.
-nohtml *--no-default-html*	do not load default html during startup. Default is: **False**. This switch will prevent **smd** from loading and emitting the default document markup in **def_html.md** during startup and **def_close.md** during shutdown. Also note that precedence is given to **[user.imports]/def_html.md** if it exists, over the system provided **[sys.imports]/def_html.md**. By the way, this is true for all of the document defaults; that is, if a **user** provided version exists, it will have precedence over any **system** provided version.
-nohead *--no-default-head*	do not load default head during startup. Default is: **False** This switch will prevent **smd** from loading and emitting the default head markup in **def_head.md** during startup.
-nobody *--no-default-body*	do not load default body during startup. Default is: **False** This switch will prevent **smd** from loading and emitting the default body markup in **def_body.md** during startup and **def_bodyclose.md** during shutdown.
-html FILENAME *--set-html-name FILENAME*	set filename of document html markdown. Default is: **None**. This option allows you to specify the filename that will be used to load the default document markup that is normally contained in **def_html.md**.
-head FILENAME *--set-head-name FILENAME*	set filename of document head markdown. Default is: **None**. This option allows you to specify the filename that will be used to load the default head markup that is normally contained in **def_head.md**.
-body FILENAME *--set-body-name FILENAME*	set filename of document body markdown. Default is: **None**. This option allows you to specify the filename that will be used to load the default body markup that is normally contained in **def_body.md**.

-bodyclose FILENAME *--set-bodyclose-name FILENAME*	set filename of document body close markdown. Default is: **None**. This option allows you to specify the filename that will be used to load the default body close markup that is normally contained in **def_bodyclose.md**.
-close FILENAME *--set-close-name FILENAME*	set filename of document close markdown. Default is: **None**. This option allows you to specify the filename that will be used to load the default closing markup that is normally contained in **def_close.md**.

Common Use Cases

Typically, you will use **smd** to interactively learn the Script Markdown syntax and semantics. Although you could technically use it to parse entire documents, it does not contain support for handling the CSS file(s), so you would have to account for that separately. It is easier to use **smdparse** or **ismd** for that, since they both handle the CSS files, and are intended for use in automation and scripting tasks.

Here are a few common use cases for **smd**:

Common use cases for smd

```
// Start the parser in interactive mode, do not load the document defaults
$ smd -nd

// Parse the file samples/hello/hello.md
$ smd -f samples/hello/hello.md

// Same as the first example above i.e. smd -nd
$ smd -nohtml -nohead -nobody
```

smdparse command line parameters

The **smdparse** command line interface is in the Python module **smdparse.py**. This module contains the code to generate an HTML file from a text file written in script markdown (**smd**) format. **smdparse** utility exits after parsing the input file.

smdparse accepts the same parameters as **smd**, plus a few additional parameters useful for scripting and automation (the most common use case for **smdparse**):

smdparse command line parameters

```
    [-h]
    -f FILENAME
    [-c [CSSFILELIST [CSSFILELIST ...]]]
    [-d [PATH]]
    [-i [IMPORTFILELIST [IMPORTFILELIST ...]]]
    [-sph HEAD_FILE_NAME]
    [-dbg]
    [-notid]

    Plus all the options supported by smd
```

smdparse Common Parameters

This first group of options applies to both **smdparse** and **ismd**.

Parameter	Description
-h *--help*	show help message and exit
-f FILENAME *--filename FILENAME*	the FILENAME that you want to parse. Unlike **smd**, there is no default, this parameter must be specified.

-c [FILE1 [FILE2 ...]] *--cssfile [FILE1 [FILE2 ...]]*	the CSS file(s) you want used for the styling. If you specify **-c** by itself, the default CSS file **smd.css** will be used, and it will also be copied to the output directory **-d**. Otherwise, you can specify one or more CSS files, and each of them will be copied to the output directory, and a **link** tag will be added for each in the **<head>** section of the generated HTML output.
-d [PATH] *--path [PATH]*	the directory that you want the HTML file written to. If you either do not specify this option, or if you specify **-d** by itself, the default output directory will be **./html**, otherwise, the specified **PATH** will be used. Also, if the output directory does not exist, it will be created.
-i [FILE1 [FILE2 ...]] *--import [FILE1 [FILE2 ...]]*	list of file(s) to import after **builtins.md** loaded. If you specify **-i** without a filename, then *import_**--filename*** will be used. For example .... Default loaded. Default is **None**. If **-i/--import** is specified without a filename,
-sph FILENAME *--smdparse-head-name FILENAME*	the filename to use for the head HTML markdown. Default is **smdparse_head.md**

smdparse-Only Parameters

These options are specific to **smdparse**.

Parameter	Description
-dbg *--debug*	display additional debug information. Default is: **False**. This option will cause **smdparse** to dump the **ScriptParser** object instance variables as well as the **SystemDefaults** object instance variables, either or both of which you might find helpful when debugging script processing.
-notid *--no-tid-in-output*	do not display the thread id in the output messages. Default is: **False**. This option is used during the unit testing of **smdparse** to prevent the thread ID from being output in log messages, making it easier to validate the output against expected results. Most likely you won't use this option, unless you are unit testing changes made to **smdparse**.

Common Use Cases

Here are a few common use cases for **smdparse**.

```
Common use cases for smdparse


// Parse samples/hello/hello.md write to ./html directory
$ smdparse -f samples/hello/hello.md

// Parse samples/hello/hello.md, include default CSS file in output folder
$ smdparse -c -f samples/hello/hello.md

// Parse samples/hello/hello.md, include default CSS file, export to export folder
$ smdparse -c -f samples/hello/hello.md -d export
```

ismd command line parameters

---

ismd is an interactive version of **smd** that monitors all of the files that were processed when the specified input file is parsed, and whenever changes are made to any of them, it will re-parse the input file and update all of the output monitors.

By default, **ismd** creates a **Google Chrome** output monitor, so you can see the results of the parse displayed in the window, and when you update & save one of the files that went into creating it, it will automatically refresh the output window. This is extremely useful when you are developing new content, as you can see the results of your changes in real-time.

ismd accepts the same parameters as **smd**, along with the common parameters for **smdparse** reviewed in the previous section, plus these additional parameters:

ismd-specific command line parameters

```
[-m MONITOR [MONITOR ...]]
[-ipf]
```

Plus all the options supported by smd and the smdparse common options.

ismd-Only Parameters

These options are specific to **ismd**.

Parameter	Description
-m MONITOR [MONITOR ...] *--monitor MONITOR [MONITOR ...]*	Specify the monitor (one or more of the following: **chrome**, **safari**, **firefox**, **hostgui**, **endpoint**) you want used to display the output. The default is **chrome**. You can specify more than one monitor, which allows you to review the ouput in any/all of the monitors you are interested in deploying on. **hostgui** is a specialized monitor that displays the raw HTML in a platform-specific window, so that you can review the actual HTML being generated by **smd**. **endpoint** is another specialized monitor that will create an endpoint on **localhost** at the following URL: **localhost:8080/smd/*FILEBASE*/html/*FILEBASE*.html.** ***FILEBASE*** is the basename of the filename specified in the **-f** parameter. For example, if you specified **-f path/mydoc.md -m endpoint**, then ***FILEBASE*** will be **mydoc** in the URL string.
-ipf *--ignore-parse-fail*	Ignore failure of initial parse. Default is: False. By default, when **ismd** encounters a failure during the initial parse, it exits with an error code. In some cases, however, you might prefer to continue on and enter the monitor loop, knowing that you will fix whatever issue is causing the failure (assuming it's a scripting/markdown error and not an error in the Python code). In this case, specify the **-ipf** option, to ignore the error, and continue with the monitoring.

Common Use Cases

Here are a few common use cases for **ismd**.

Common use cases for ismd

```
// Parse samples/hello/hello.md and launch a Chrome monitor window
$ ismd -f samples/hello/hello.md

// Parse samples/hello/hello.md, include default CSS file, monitor using hostgui window
$ ismd -c -f samples/hello/hello.md -m hostgui

// Parse samples/hello/hello.md, include default CSS file, monitor with firefox and
create an endpoint
$ ismd -c -f samples/hello/hello.md -m firefox endpoint
```

ismd monitors changes and keeps all monitor windows updated until CTRL-C is pressed in the terminal window where you launch **ismd**.

Summary of Command Line Parameters Chapter

Okay, at this point, we've covered the setup of **smd** and the major components that make up the package and we also discussed the various command line options for those major components. It's time to get down to business, and learn how to use **smd** for generating HTML content from

your markdown files.

Table of Contents

Getting Started

In this chapter, we will discuss the basics of SMD: what it is, and some of the things you can do with it. It's a versatile script parser, that you can use to easily create HTML documents on the fly, and turn them into a static page, or serve them via an endpoint. Let's dive right in.

What is SMD?

SMD is a Python command line utility that takes plain text files loosely, *oh, so loosely*, based on Markdown as input, and generates an HTML document. A CSS file can be used to style the output, making it super easy to customize the final render to your liking.

In the beginning, it started as a utility for generating AV (Audio/Video) style scripts for filmmakers. Essentially:

> **Markdown** list item tags ***(*, -, +)*** were used to identify ***visuals*** (shots), and regular paragraphs were the ***audio/narration*** that go along with the visuals.

At least that's how it started out. In order to use it effectively, however, you needed to be using BBEDIT, which has built-in support for previewing markdown files, or really anything that uses Python to generate the **<body>** contents of an HTML document. It was also difficult to use because BBEDIT would only refresh the preview window when the primary document was updated, and as the scripts became more complex and modular, making a change in an imported script meant you had to change the primary document in order to see the changes.

It's grown quite a bit since those early days, and while some markdown span elements are still supported, generating AV Scripts is now done using specialized ***macros*** that are provided as part of the built-ins that come with SMD. The remainder of this document will provide an in-depth overview of the capabilities of SMD, but you are also encouraged to look thru all of the samples provided to get a better idea of what it can do. This manual, for example, is written using **smd**.

Inline Markdown

A few of the standard markdown span elements are supported, as are a couple of specialized span elements. These include (in order of precedence, i.e. the order they are parsed when scanning input lines):

1. Variables - These are smd namespace variables accessed with this type of markdown: **[variable]**
2. Strong - Double ** placed around content will apply the **** tag: ****wrap text in double asterisk for bold****
3. Emphasis - Single * placed around content will apply the **** tag: **wrap text in single asterisk for emphasis**
4. Insertion - Double + placed around content will apply the **<ins>** tag: **++wrap text in double plus signs for <ins> tag++**
5. Deletion - Double ~ placed around content will apply the **<del>** tag: ~~~~wrap text in double tilde for <del> tag~~~~

Here are a few examples:

When I write *text*, it becomes *text*, and when I write **text**, it becomes **text**.

You can stack them too, so that ***text*** becomes ***text***.

When you want to wrap text with <ins>, use the double plus (++), and it will render like this: **Stuff that's been added**.

Similarly, when you want to wrap text with <del>, use the double tilde (~~), and it will render like this: ~~Stuff that's been removed~~.

So far, this is similar to using regular markdown, so that part, at least, probably isn't new or exciting. When we start mixing in variables, however, things will get much more interesting. We will see that in the chapters that cover the namespaces: **var**, **html**, **link**, **image**, and **code**.

Table of Contents

Headings

Just like in standard Markdown, you can use the # symbol at the beginning of a line to designate an HTML **<h1>** element. ## symbols designate an **<h2>** element, and so on, up to ###### for **<h6>**. Here are examples of each.

Heading h1

Heading h2

Heading h3

Heading h4

Heading h5

Heading h6

And this is how they will look in the document when it's formatted:

Heading h1

Heading h2

Heading h3

Heading h4

Heading h5

Heading h6

You may want to style the headers to your liking in the smd.css file.

> **IMPORTANT! USING THE # TO CREATE HEADINGS *REQUIRES* THAT THE HASHTAG START IN COLUMN 1 OF THE LINE. IF YOU START IT ANYWHERE ELSE, IT WILL BE INTEREPRETED SIMPLY AS AN INLINE HASHTAG.**

For example:

this works

But **#this works**, clearly does not work.

Table of Contents

The builtins

This chapter will cover the builtins that are automatically loaded by **smd** (unless the *-ndb* and *-nub* command line switches are specified).

File [sys.imports/builtins.md]

All of the system provided builtins are located **[sys.imports]** directory. When **smd** is initializing, just before it loads the default document sections, it will **@import** the file **builtins.md**. If you examine this file, you will see it just imports several other files:

[sys.imports/builtins.md]

```
@import "[sys.imports]/defaults.md"
@import "[sys.imports]/common.md"
@import "[sys.imports]/code.md"
@import "[sys.imports]/link.md"
@import "[sys.imports]/html.md"
```

Each of these imports, in turn, defines things specific to the namespace they represent. Although they are technically variables, it might be easier to think of them as macros, because when they are encountered while parsing, they expand to something else. In some cases, it's just a direct text replacement, e.g. **[E.ast]** expands to *****, but in other cases, they might push several lines of other content, including **smd** commands onto the input stream!

If you directly examine the contents of these files, you will see the things that they define for you to use in your own markdown files. If you examine the contents of the user documentation, you will see how they are used to generate and format HTML content.

> Some of the content in these files will look rather foreign at first glance, especially those in **code.md**. However, once you understand the syntax for each of the namespaces, it will become much easier to decipher these system provided builtins.

System vs. User builtins

smd supports the concept of both system and user defined builtins. The system builtins are located in the **[sys.basepath]/import** directory. User builtins, on the other hand, are located in the **~/.smd/import** directory. On Linux based operating systems (of which macOS is a derivative), the tilde ~ character refers to the base of the currently logged in user directory: **/Users/<username>**.

Whenever the system is ready to load the builtins, it will look in both places for a file named **builtins.md**. It looks in the system directory first, and once finished processing those, it then looks in the user directory (if it exists). This allows you to **override** the system defaults with your own declarations.

The *sys* builtin

There is a special built-in in the @var namespace named *sys* that contains several attributes that are useful in your markdown files.

@var sys default attributes

basepath=The location of the **smd** package. This directory is where **smd, smdparse** & **ismd** are stored.
imports=The location of the import directory. This directory is located here: **[sys.basepath]/import**
root=The root of the **smd** installation. This directory is located here: **[sys.basepath]/..**
user_basepath=The location of the user smd directory for the currently logged in user.
user_imports=The location of the user smd import directory for the currently logged in user.
user_root=The home directory for the currently logged in user.

The most commonly used of the sys attributes is *imports*. Every time you want to load one of the system defined import files you will reference it. For example, **@import "[sys.imports]/divs.md"** would import the *divs.md* markdown script.

Table of Contents

helpers.md built-ins

helpers.md contains a set of built-ins that declare a set of commonly used tags, constants and other useful things to help with creating content.

> note default content

Let's start by taking a look at what's contained in **helpers.md**:

Contents of helpers.md builtins

```
@import "$/html.md"
@import "$/code.md"
@import "$/divs.md"
@html _="tab" _inherit="span" class="indent"
@var _="tab2" o="[tab.<][tab.<]" c="[tab.>][tab.>]" o3="[tab.<][tab.<][tab.<]"  c3="
[tab.>][tab.>][tab.>]"
@html _="us" _inherit="span" style="text-decoration:underline"
@html _="spanwc" _inherit="span" class="blue"
@html _="divx" _inherit="_div_extras_"
@html _="divx1" _inherit="divx" style="border-bottom:5px solid green;color:green"
@html _="divx2" _inherit="divx" style="border-bottom:3px solid black;color:green"
@html _="divx3" _inherit="divx" style="border-bottom:2px solid black"
@html _="li2" _tag="li" _inherit="li" style="font-size:1.3em;margin-left:2em"

@var e_us="{{html.us.<}}{{self.t}}{{html.us.>}}" t="{{self._help}}" _help="*{{self._}}
(t=\"text_to_underscore\")*"
@var hash1="# [code.repeat.run(t=\"-\", c=\"42\")]"
@var hash2="## [code.repeat.run(t=\"-\", c=\"42\")]"
@var hash3="### [code.repeat.run(t=\"-\", c=\"42\")]"
@var wrap_h="{{code.pushlines(t=\"@wrap html.divx\n{{self.t}}\n@parw 1\")}}"\
    hash1="{{code.pushlines(t=\"@wrap html.divx\n{{var.hash1}}\n@parw 1\")}}"\
    hash2="{{code.pushlines(t=\"@wrap html.divx\n{{var.hash2}}\n@parw 1\")}}"\
    hash3="{{code.pushlines(t=\"@wrap html.divx\n{{var.hash3}}\n@parw 1\")}}"\
    chapter="{{code.pushlines(t=\"@wrap html.divx1\n{{self.t}}\n@parw 1\")}}"\
    section="{{code.pushlines(t=\"@wrap html.divx2\n{{self.t}}\n@parw 1\")}}"\
    subsect="{{code.pushlines(t=\"@wrap html.divx3\n{{self.t}}\n@parw 1\")}}"

// Make a couple of specialized Simple DIV **note** types where the font color is blue
or red
@html _="_bluenote_p_" _inherit="_note_p_" class="[html._note_p_.class] blue"
@html _="_rednote_p_" _inherit="_note_p_" class="[html._note_p_.class] red"
// create var.bluenote, inherit note attributes, and then just change the p class in
the _inline attrs
@var _="bluenote" _inherit="note"
[code.attr_replace_str(s_str="_note_p_" r_str="_bluenote_p_"
attr="var.bluenote.inline_nd")]
[code.attr_replace_str(s_str="_note_p_" r_str="_bluenote_p_"
attr="var.bluenote.nd_open_inline")]
@var _="rednote" _inherit="note"
[code.attr_replace_str(s_str="_note_p_" r_str="_rednote_p_"
attr="var.rednote.inline_nd")]
[code.attr_replace_str(s_str="_note_p_" r_str="_rednote_p_"
attr="var.rednote.nd_open_inline")]

@html _="bigmargin" _tag="div" style="margin-left:3.3em;margin-right:3.3em"
_open="@@{{self.<}}" _close="@@{{self.>}}"

@html _="bmgreybg" _inherit="bigmargin" style="[html.bigmargin.style];border:2px solid
black;background:lightgray"
```

It starts by importing the files that contain things that are being extended or are used to create the macros it is declaring. In most cases, these would already be imported, so they are ignored, but in the rare case where they haven't been, it will prevent a bunch of errors when it tries to parse the contents of this file.

A few useful **@html** variables are created, including **tab** and **us**. **tab** just indents content by wrapping it with a **** tag, and **us** can be used to underline content inline. It would be a little clumbsy to write **[html.us.<]**underline**[html.us.>]**, so a little later on the variable **e_us** is created, that makes it easier.

The **wrap_h** variable contains several useful attributes to simply creating headings. They are:

hash1 - create <h1> header of dashes
hash2 - create <h2> header of dashes
hash3 - create <h3> header of dashes

chapter - create a chapter heading with a thick green bottom border
section - create a section heading with a smaller green bottom border
subsect - create a sub-section heading with a thin black bottom border

These are used throughout the user manual to create the headings you see in the documentation. For example, if I write **[wrap_h.hash3]**, the parser will emit (and your browser will render):

If instead I write **[wrap_h.subsect(t="###E and EMOJI")]**, this is what I get:

E and EMOJI

E and **EMOJI** create namespace wrappers for commonly used HTML Entities and Emojis. Another helper builtin, **e_moji**, makes it easy to create two different sizes of emojis. For example, **[e_moji(e="mask")]** emits 😷 , and **[e_moji.big(e="mask")]** emits 😷.

The variables **e_tag** and **e_var** are used quite a bit in the documentation to encode the entities and markdown so that it can be shown inline instead of having either the parser or the browser process it.

Finally, a commonly used trick of extending an underlying macro is used to create two custom versions of the **note **, **bluenote** and **rednote**. This is done by creating a new variable using the *color*note name, inheriting from the underlying **note** variable, and then one of the attributes within that variable is modified to use the newly created color class in the **@html** namespace. Let's use all three of them (note, bluenote and rednote) to see how they render:

This is my inline note using the default **note** builtin.

note default content

note default content

Taking an underlying builtin and just **tweaking** it makes it easy to extend the builtins, without having to spend a lot of time recreating the wheel so to speak... 😜

The last things (at the time of writing this chapter) are the **bigmargin** and **bmgreybg @html** variables that are also used in the documentation to render content with bigger than the default margins. They can both be nested, which makes it easy to do things like this:

Here is some content with 3.3em right and left margins. See how it's been indented on the left?

Now let's nest them:

Here is some content with 3.3em right and left margins. Now it's got double margins or 6.6em.

Let's try the **bmgreybg** with the same thing:

Here is some content with 3.3em right and left margins. See how it's been indented on the left?

Now let's nest them:

Here is some content with 3.3em right and left margins. Now it's got double margins or 6.6em.

That's kind of interesting with the double boxes. Let's use the **bigmargin** first, then the **bmgreybg** inside:

Here is some content with 3.3em right and left margins. Now it's got double margins or 6.6em.

Now let's use the **bluenote** to render the middle text and **bigmargin** for the two nests:

> Here is some **blue** content with 3.3em right and left margins. Now it's got double margins or 6.6em.

Anyway, I think you get the point. There's a lot you can do with the markdown and building custom variables or macros or whatever you want to call them. Let's move on.

Table of Contents

Namespaces

In **smd**, one of the primary types of markdown (specially formatted text) that you can use is the **variable**. There are several types of variables that are supported, and they are referred to as namespaces within **smd**. They are:

@var - The primary namespace for creating variables
@html - A namespace for creating HTML tags
@link - A namespace built on @html, used for creating hyperlinks and anchors (bookmarks)
@image - A specialized namespace specifically for creating HTML Image **<img>** elements
@code - A specialized namespace for creating complex macros (written in Python)

Built in Help

There are a few things we need to cover which apply to all namespaces, and in most cases, the chapters are organized so that if you maintain the order, the concepts will be introduced as you go. However, all namespaces support the concept of built-in help, which is an optional attribute named **_help** that can be added to any variable (either at declaration time or later on via **@set**). I want to cover this topic first, since accessing the builtin help, especially for some of the more advanced builtins, will likely be crucial to you learning how to effectively use **smd**. Let's see how it works.

> Because we haven't yet discussed any of the namespaces, this might seem a little out of place, so feel free to jump ahead to the **@var** namespace chapter, and then return here afterwards. Otherwise, stick with us, it's short, and should assist with understanding of how things work in any of the namespaces.

Take the following markdown which declares a variable **var1**, which takes a parameter **p** and wraps it with **** and **** tags. It looks like this:

```
@var var1="***{{self.p}}***" p="sample text"
```

Once declared, we can write markdown as follows to place the *em* and **strong** tags around text: **[var1(p="my important text")]**. And it would render like this:

my important text

Now that's all good (and rather simple), but down the road, when I'm trying to remember how to use it, I don't want to have to track down the file where it is declared to examine it, or even dump the variable definition to decode it's usage. Enter the **_help** attribute. If we add _help to our previous declaration like so:

```
@var var1="***{{self.p}}***" p="sample text" _help="Usage: {{self._}}(p=\"text to wrap with ***\")"
```

Then, when I want to use it, I can type **[var1.?]**, and it will generate this:

```
Usage: var1(p="text to wrap with ***")
```

You can also use the **_help** attribute directly and achieve the same result, but the **?** is shorter and easier to type! So what about the **??** variant? How does that fit in?

?? is a specialized version of the retrieve help attribute that strips most of the HTML markup that is embedded in the help string, making it easy to read when you are running **smd** interactively i.e. *smd -nd*. If I use it in a document that is used to generate HTML, it won't look that good. For example, if I type **[var1.??]**, it will generate this:

```
Usage: var1(p="text to wrap with *")
```

In this case, it doesn't look too bad, only causing the three inline astericks *** to render as one red and one black, but let's see how dumping the help string for something more complex looks, and it'll be more obvious. Take the built-in macro **code.get_value**. If I type **[code.get_value.?]**, it will display this:

Results of code.get_value.?

get_value(v="variable_name" ret_type="0|1|2|3|9" escape="True|False" esc_smd="True|False")

v - variable / attribute name to get

ret_type - how to emit the value (0, 1, 2, 3 or 9)
0 - will return the value raw i.e. not marked down
1 - will return the value with the initial markdown pass, but without handling delayed expansion
2 - will return the after replacing [with [and] with]
3 - will return the after replacing **code.get_value.** with **ns.varname.**
9 - will return the marked down value (default)

escape - True to escape the HTML, False otherwise (default)

esc_smd - True to escape the SMD, False otherwise (default)

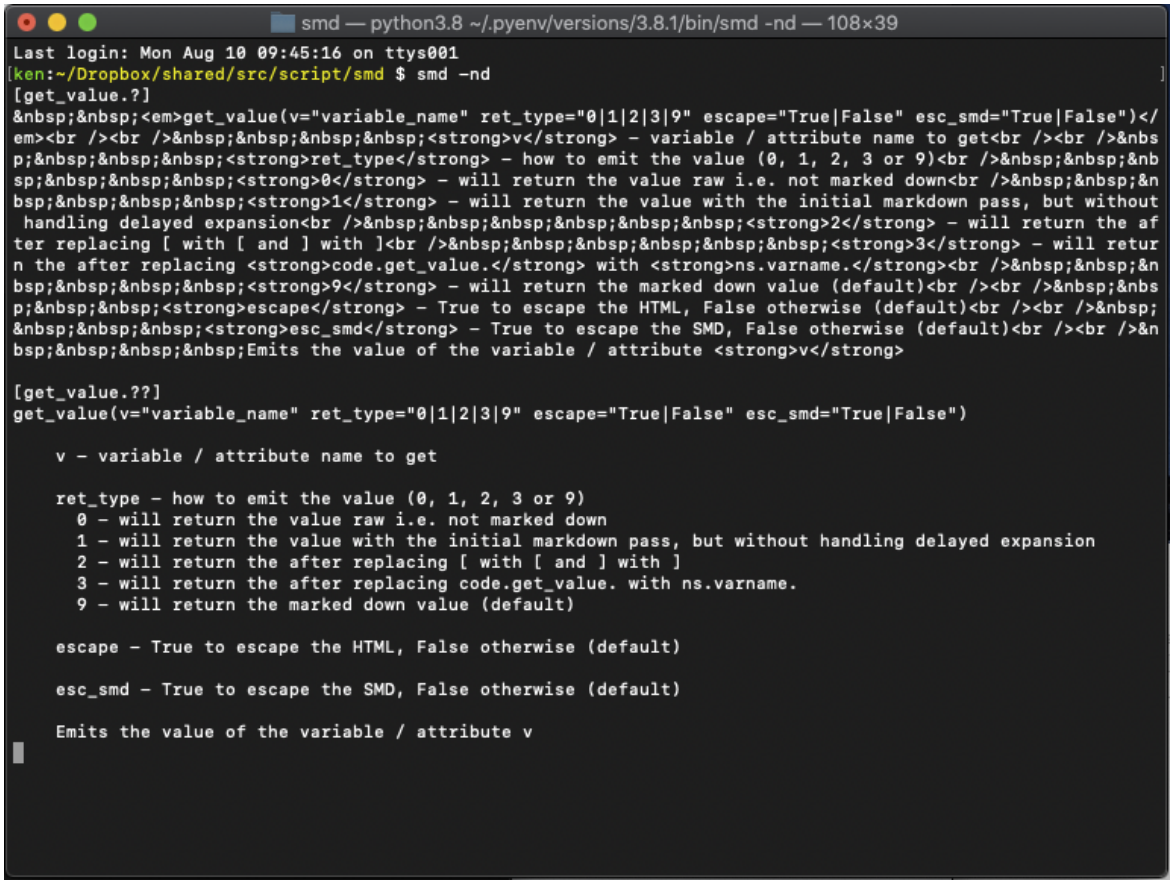
Emits the value of the variable / attribute **v**

and then if I type **[code.get_value.??]**, I get this:

Results of code.get_value.??

get_value(v="variable_name" ret_type="0|1|2|3|9" escape="True|False" esc_smd="True|False") v - variable / attribute name to get ret_type - how to emit the value (0, 1, 2, 3 or 9) 0 - will return the value raw i.e. not marked down 1 - will return the value with the initial markdown pass, but without handling delayed expansion 2 - will return the after replacing [with [and] with] 3 - will return the after replacing code.get_value. with ns.varname. 9 - will return the marked down value (default) escape - True to escape the HTML, False otherwise (default) esc_smd - True to escape the SMD, False otherwise (default) Emits the value of the variable / attribute v

So obviously, if you are expecting the content to be readable inside an HTML document, you want the first option. However, if you start **smd** interactively, and type the same two things, you will see the the **.?** version has all sorts of HTML tags embedded making it hard to read in the text console, but **.??** is nice and simple and easy to read. Here's a screen shot to prove it:



That should give you enough to use the built-in help system while you are learning **smd**. In the chapters that follow, we will take an indepth look at each namespace to learn it's specific syntax and semantics. With that, let's move right into the chapter on the **@var** namespace.

Table of Contents

The @var Namespace

The @var keyword is used to construct a more flexible type of variable for your documents. These variables are stored in the **@var** namespace, and can have names that are identical to variables in other namespaces, although it is normally recommended that you avoid doing that.

We will discuss this namespace first, because it is the basis for all namespaces, and most of the features, syntax and semantics apply to the others. Given that, it is a great starting point for our discussion on creating variables for use in **smd** documents.

@var Syntax

@var command syntax

@var name="value" [attr="value" [...]]

@var _id="name" [_format="value" [...]]

NOTE: either underscore '_' or '**_id**' can be used to name the variable.

@var variables are the only ones that can use the shorthand **name="value"** notation when defining a variable. All the other namespaces require that the **[_ or *_id*]** special attribute name be used to specify the variable name. Note that when the shorthand notation is used, it is the first **name="value"** pair that will be used to name the variable. Each of the following examples does exactly the same thing:

Declaring a variable

```
@var fu="bar"
@var _="fu" _format="bar"
@var _id="fu" _format="bar"
```

Essentially, declaring a variable in *any* of the namespaces is done in exactly this fashion. ***@ns*** followed by a series of attribute="value" pairs. Remember, only **@var** allows the shorthand **fu="bar"** format for naming the variable and setting the **_format** attribute value. All other namespaces require the alternate format. In addition, namespaces based on **@html** and **@code** normally do not specify **_format**, since that attribute is constructed on the fly during the declaration (**@html**) or is not used (**@code**). More on that later.

NOTE: If you redeclare a variable, it will be overwritten with the new declaration. If you want to add attributes to an existing variable, use **@set**

Declaring variables in other namespaces

```
@var _id="fu" _format="bar"
@html _id="fu"
@link _id="fu"
```

Notice that in the **@html** and **@link** declarations we didn't specify the _format attribute. You'll see why when we get to the chapters on those namespaces, but for now, just realize that if you did specify the _format="bar" on those, then they would result in the exact same behavior as the @var variable. That is, when you write **[var.fu]**, **[html.fu]**, **[link.fu]** in your markdown document, each would simply emit **bar**. Let's try that now.

Using _format on @html and @link

```
// declare 'fu' in @var, @html & @link
@var _id="fu" _format="bar"
@html _id="fu" _format="bar"
@link _id="fu" _format="bar"

// now print each one
[var.fu] emits bar
[html.fu] emits bar
[link.fu] emits bar

// Redefine html.fu and link.fu
@html _id="fu"
@link _id="fu"

// now print each one
[var.fu] emits bar
[html.fu] emits <fu></fu>
[link.fu] emits <a></a>
```

The reason that **[html.fu]** emits *<fu></fu>* will be explained in the **@html** chapter on variable declarations. For now, just go with it. 😜

One other caveat to the shorthand notation: if you specify also specify the **_format= attribute**, whatever you attempt to assign as part of the declaration will be ignored, and the value specified with _format will win. For example:

Using shorthand notation and specifiying _format

```
@var fu="bar" _format="123"

Then [fu] will emit 123 instead of bar
```

If you fail to specify the **_format** attribute when you declare an **@var** variable, and you do not use the short-hand notation to create it, then referencing the variable will result in all attributes being dumped. Let's see how that works. Take the following markdown:

```
@var _="myvar" attr1="value1" attr2="value2"
```

Then, if we write **[myvar]** the parser will emit:

```
attr1="value1"
attr2="value2"
```

There isn't much usefulness in this side affect since there is no way to control the formatting, however, you may find it handy as a way of checking the current definition of a variable without having to dump it. i.e. **[myvar]** is shorter than **[code.dump(ns="var" name="myvar")]** or even **@dump var="^myvar$"**.

Inheriting attributes

One last concept to discuss about declaring variables is the attribute **_inherit**. This attribute can be used at declaration time only, and it's purpose is to avoid having to redefine common attributes. This is used throughout the builtin files that come with **smd**, so you'll see it used quite often. Let's look at an example of how it is used.

Using _inherit to streamline declarations

```
// declare 'smdtag' variable
@var smdtag="@@{{self.il}}" il="{{encode_smd(t=\"{{self.il}}\")}}" b="**{{self.il}}**"
em="*{{self.il}}*" emb="***{{self.il}}***"

// now declare several variables based on smdtag
@var _id="smdvar" _inherit="smdtag" p="@var"
@var _id="smdhtml" _inherit="smdtag" p="@html"
@var _id="smdlink" _inherit="smdtag" p="@link"

// now we have three new variables that all contain similar attributes
[smdvar] = @var, [smdvar.b] = @var, [smdvar.em] = @var, [smdvar.emb] = @var
[smdhtml] = @html, [smdhtml.b] = @html, [smdhtml.em] = @html, [smdhtml.emb] = @html
[smdlink] = @link, [smdlink.b] = @link, [smdlink.em] = @link, [smdlink.emb] = @link
```

If you specify an attribute that is defined in the underlying inherited variable, it will override the underlying value. Otherwise, the new variable will contain all of the same attributes, plus any new ones added at the time of declaration (or later using **@set**).

@var Variable names

Variable names in the **@var** namespace, well actually, in **any** namespace are restricted to these requirements:

- Must begin with a letter or the underscore character
- Must contain only letters, numbers or underscores
- Are case sensitive. That is, abc and ABC are different variable names

Here are some examples of variable names:

Variable Name	Valid / Invalid
a	valid
a1	valid
a_1	valid
_a1	valid
A	valid
1a	invalid
a-b	invalid
1+	invalid
%a	invalid

Attribute names, on the other hand, are restricted to these requirements:

- Must contain only letters, numbers, underscores or dashes
- Are case sensitive. That is, abc and ABC are different variable names

Here are some examples of attribute names:

Attribute Name	Valid / Invalid
a	valid
a1	valid
a_1	valid
_a1	valid
A	valid
1a	valid
a-b	valid
1+	invalid
%a	invalid

As you can see, attribute names are not quite as restrictive in their naming as variable names, allowing dashes to be used, and also allowing names to begin with numbers.

@var Built-in Attributes

@var variables have a number of built-in attributes to extract common components. In fact, all namespaces share these built-in attributes, and in the case of variables based on the **@html** namespace, they are relied on much more, but nonetheless, they can be useful in any of the namespaces, if for no other reason to assist in debugging complex declarations.

Here is the full list of built-in attributes supported across all namespaces:

Attribute Name	Description
_	The name of the variable
_id	The name of the variable
?	Returns **_help** if present otherwise **No help available**
??	Same as ?, but strips most HTML tags. Useful when running **smd** interactively.
*_private_attrs_*	All of the private attributes (those that begin with underscore '_')
*_public_attrs_*	All of the public attributes (those that do not begin with underscore)
*_private_attrs_esc_*	Same as private attributes, but quotes are escaped.
*_public_attrs_esc_*	Same as public attributes, but quotes are escaped.
*_public_keys_*	The public attribute names
*_private_keys_*	The private attribute names
*_all_attrs_*	All attributes
*_all_attrs_esc_*	Same as all attributes, but quotes are escaped.
*_null_*	A null attribute - emits nothing, used to set default values of any public/private attribute

Examples

```
[var.fu._] = fu
[var.fu._id] = fu
[var.fu._all_attrs_] = _format="123"
[var.fu._private_keys_] =_format,
[var.fu.?] =No help available

[html.divx._all_attrs_] = _format="<{{self._tag}}{{self._public_attrs_}}>
</{{self._tag}}>" _tag="div" class="extras"
[html.divx._private_attrs_] = _format="<{{self._tag}}{{self._public_attrs_}}>
</{{self._tag}}>" _tag="div"
[html.divx._public_attrs_] = class="extras"
```

@var Miscellaneous Examples

Here are just a few more examples to help drive home your understanding of the declaration and usage of variables in the **@var** namespace.

Consider the following markdown:

```
@var _id="varname" attr1="value1" _format="format string"

[varname.attr1] will emit value1.
[varname._id] will emit varname.
[varname._format] will emit format string.
```

So that's pretty cool, but there's a bit more to the _format attribute. You can reference the attributes contained within the variable by using **{{self}}.attrname**. Given that, if we write the following:

```
@var _id="fullname" first="ken" last="lowrie" _format="{{self.first}} {{self.last}}"
```

```
@var _id="fullname" first="ken" last="lowrie" _format="{{self.first}} {{self.last}}"

[fullname] would emit ken lowrie.
```

Using that, you can build some pretty powerful tools for automating frequently used tasks in your documents. Take a look at the **film.md**, **image.md** and **avs.md** tests to get an idea of what you can do.

Before we leave this, take note of the difference between {{first}} and {{self.first}}. Both syntaxes are valid, but the first one references the normal variable first, while the second one (self.first) references the attribute named *first* defined within the *varname* variable. Also take note that you can reference the attributes of other **@var** variables as long as you qualify them. Let me show you a quick example of that. Take this:

```
@var _id="var1" first="ken" last="lowrie"

@var _id="var2" prefix="mr." _format="[self.prefix] [var1.first] [var1.last]"
```

Then, if you write this:

```
[var2] would emit mr. ken lowrie.
```

One last thing, remember how we discussed that sometimes you need a way to resolve ambiguities of variables across the different variable types? **image.** can be used to resolve a variable inside the **@image** namespace, and **var.** can be used to resolve a variable inside the **@var** namespace. Given that:

```
[var2] and [var.var2] resolve to the same thing.
```

Did you notice that in the first example, we used **{{** and **}}** around the variable/attribute name, and in the second, we used square brackes i.e. **[** and **]**? What's the difference (if there is one), and why use one over the other? Good question. Now is a good time to talk about the concept of attribute markdown expansion, and how it works.

Attribute Expansion

Next up is a topic you are going to see over and over in the coming chapters, and even more once you start examining the built-ins and this user guide. What I'm talking about is the concept of attribute expansion, in the context of **smd**'s markdown variables and attributes.

As you have seen, variables can have one or more attributes, and those attributes can be anything from plain text, to specialized markdown such as *, **, ++ and ~~, to the referencing of other attributes, both within the same definition, as well as in entirely other variables and yes, even other namespaces! Let's review a few examples to get started.

```
Attribute expansion - simple and inline markdown

@var var1="" attr1="plain text"
[var1.attr1]=plain text

@var var1="" attr1="*text with markdown*"
[var1.attr1]=text with markdown

@var var1="" attr1="<strong>text with html markup</strong>"
[var1.attr1]=text with html markup
```

With simple markdown, you are providing the values directly, and possibly adding the inline markdown elements to it for decoration.

Attributes can also reference values stored in other attributes within the same variable. This can be done using either square brackets **[ ]** or curly brackets **{{ }}**. Before looking at the examples, though, let's get acquainted with the semantic differences between the two variations.

Anytime you use square brackets during the declaration of new variables and/or attributes, they will be evaluated at the time the variable declaration is parsed. Curly braces, on the on the other hand, are not evaluated until the attribute value is referenced later in the document. This is a very important distinction, because sometimes, you want some parts of your declaration to evaluate during the parsing stage, and other times, you need them to evaluate later, when a given attribute is actually being referenced.

```
Attribute expansion - referencing attributes within the same variable

@var var1="" attr1="[self.attr1]" attr2="*from var1.attr2*"
[var1.attr1]=from var1.attr2

@var var1="" attr1="{{self.attr1}}" attr2="*from var1.attr2*"
[var1.attr1]=from var1.attr2
```

Couple of notes on the prior example. First, notice the use of **self.** as a shorthand notation for **var.var1**. This is a frequently used notation, because if you change the variable name, you don't have to change any of the **self.** references. The parser will take care of that for you.

Second, notice that both the square brackets and the curly braces produced the same results. This happens due to a side effect of referencing attributes during the declaration of a new variable. Since at the time of parsing, **var1** does not exist, the lookup for **var1.attr1** fails, and thus the markdown fails to expand. However, later, when we actually access the attribute, it will resolve, and thus both variations work the same way. Keep in mind that relying on this side effect isn't a great idea; it will often lead to pesky bugs in your markdown documents, so it is highly recommended that you use the appropriate syntax when it's called for, and avoid this side effect!

Okay, now let's add a new twist. Having the attributes in one declaration reference the attributes in another declaration. This will demonstrate the issue we were just talking about, that is, the semantic differences between square brackets and curly braces in **smd** markdown.

Attribute expansion - referencing attributes in a different variable

```
// Declare a second variable with an attribute we can reference from var1
@var extvar="" attr2="*from extvar.attr2*"

// Declare var1 and var2 then emit attr1 for each
@var var1="" attr1="[extvar.attr2]"
@var var2="" attr1="{{extvar.attr2}}"

[var1.attr1]=from extvar.attr2
[var2.attr1]=from extvar.attr2

// Change the value of extvar.attr2 and then reference attr1 again
@set _="extvar" attr2="***A new value!***"

// Review current values of var1.attr1 and var2.attr1

[var1.attr1]=from extvar.attr2
[var2.attr1]=A new value!
```

See the difference? In **var1.attr1**, it's value was set to **extvar.attr2** when it was defined, but **var2.attr1**, was told to wait to markdown its value until someone asks for it. This is what is normally referred to as *deferred expansion* in the user manual, and you will see it used all the time throughout the builtins.

Because of the deferred expansion, once we changed the value of **extvar.attr2** to something else, only **var2.attr1** reflected the new value. **var1.attr1** is still the same as what it was when it was declared. Usually, the curly braces are preferred, but there are some cases where using the square brackets are what you need. Before we leave this example, let's dump the current declarations of **var1**, **var2** and **extvar**, so you can see their current state.

Current definition of 'var1/2' and 'extvar' variables

extvar=
attr2=A new value!
_format=

var1=
attr1=from extvar.attr2
_format=

var2=
attr1={{extvar.attr2}}
_format=

So that leaves us with one more variation of delayed expansion support, the ***[!!]*** syntax. Essentially, it's a way to tell the parser's variable declaration API that you want to the variable or attribute to use square brackets, but you don't want to expand it at declaration time. In this case, the ***[!*** will be replaced with ***[*** and ***!]*** will be replaced with ***]***, just before the variable definition is written to memory. This will make more sense with an example. Let's use the exact one we just saw, but changing on the declaration of var1. Take a look:

Attribute expansion - referencing attributes using [!!] syntax

```
// Declare a second variable with an attribute we can reference from var1
@var extvar="" attr2="*from extvar.attr2*"

// Declare var1 and var2 then emit attr1 on each
@var var1="" attr1="[!extvar.attr2!]"
@var var2="" attr1="{{extvar.attr2}}"

[var1.attr1]=from extvar.attr2
[var2.attr1]=from extvar.attr2

// Change the value of extvar.attr2 and then reference attr1 again
@set _="extvar" attr2="***A new value!***"

// Review current values of var1.attr1 and var2.attr1

[var1.attr1]=A new value!
[var2.attr1]=A new value!
```

See how it worked this time? Let's look at the current declarations of the variables again:

Current definition of 'var1/2' and 'extvar' variables

extvar=
attr2=A new value!
_format=

var1=
attr1=[extvar.attr2]
_format=

var2=
attr1={{extvar.attr2}}
_format=

Now, in **var1.attr1**, it's value is set to **[extvar.attr2]**, which makes it behave like **var2.attr1**. In this case, we have applied *deferred expansion* using ***[!!]*** instead of the usual curly brackets **{{ }}**.

This can help you build some really cool automation in your documents. But you need one more thing before you get started. A way to change one or more attributes of an existing variable in any namespace. Enter **@set** and the **_null_** attribute. We've already been using them in our examples, so it may be old hat by now, but let's take a closer look at it just to be thorough.

The @set Keyword

Attributes in any variable in any namespace can be added or updated at any time. There are two (perhaps three depending on semantics) ways this can be done in all namespaces except **@code**. In @code, there is only one way to update an attribute value, and that is with **@set**. Let's see how it's done, regardless of the namespace.

Updating attribute values with @set

```
// First, let's declare a new variable called 'fu'
@var fu="bar" attr1="value1"
[fu.attr1] emits value1

// using @set, change the value of attr1 to 'value2'
@set _="fu" attr1="value2"
[fu.attr1] emits value2

// using the _null_ attribute, change the value of attr1 to 'value3'
[fu._null_(attr1="value3")]
[fu.attr1] emits value3
```

> The **_null_** attribute illustrates an important concept with the attributes of variables in **smd**. That is, any time an attribute value is changed by specifying a new value when a method is invoked, that value becomes the new value for that attribute. This is true with all namespaces except **@code**; in the **@code** namespace, attribute values overridden via a method invocation are temporary. Once the method returns, the original attribute values are restored.
>
> As previously mentioned, the only way to change the value of a variable in the **@code** namespace is by using **@set**. When this is done, the code behind the variable (Python source code) is recompiled, which updates the default value in the compiled code stored as part of the variable.

You can also add new attributes to an existing variable with **@set**. And, if you **@set** a variable that does not exist, **@set** will create it. Let's see some examples of adding attributes.

Adding attributes with @set

```
// add a new attribute to 'fu' using @set
@set _="fu" attr2="value42"
[fu.attr2] emits value42

// adding a new attribute doesn't affect existing attributes in 'fu'
[fu.attr1] still emits value3

// using the _null_ attribute, add attr3 to 'fu'
[fu._null_(attr3="many ways to add attributes")]
[fu.attr3] emits many ways to add attributes

// We can also add and update variables at the same time
[fu._null_(attr3="update 3rd attribute" attr4="add 4th attribute")]
[fu.attr3] emits update 3rd attribute and [fu.attr4] emits add 4th attribute

// We can also add and update variables at the same time using @set
@set _="fu" attr2="update attr2" attr6="6th attribute"
[fu.attr2] emits update attr2 and [fu.attr6] emits 6th attribute
```

Let's dump the variable 'fu' to see all the attributes and their values.

Current definition of 'fu' variable

fu=
attr1=value3
_format=bar
attr2=update attr2
attr3=update 3rd attribute
attr4=add 4th attribute
attr6=6th attribute

And finally, you can also specify the namespace two different ways with **@set**. Witness:

Specifying the namespace

```
// Specify the namespace with @set; don't leave it to chance!
@set _ns="var" _="fu" attr6="update attr6"
[fu.attr6] emits update attr6

// Alternate method to specify namespace
@set _="var.fu" attr6="update attr6 again!"
[fu.attr6] emits update attr6 again!
```

One last time let's dump the variable 'fu' to see all the attributes and their values.

Current definition of 'fu' variable

fu=
attr1=value3
_format=bar
attr2=update attr2
attr3=update 3rd attribute

attr4=add 4th attribute
attr6=update attr6 again!

Here are just a few more examples to help drive home your understanding of the declaration and usage of variables in the **@var** namespace.

Given the following markdown:

Create a new smd variable c2

```
@var ns="var"

@set _ns="[ns]" _="c2" \
      attr1="*value 1*" \
      attr2="**value 2**" \
      attr3="{{self.attr1}}--{{self.attr2}}" \
      attr4="[self.attr2]--[self.attr1]"
```

Then the following markdown sometime later will produce:

Display the values for each of the attributes in c2

[c2.attr1] = *value 1*
[c2.attr2] = **value 2**
[c2.attr3] = *value 1*--**value 2**
[c2.attr4] = **value 2**--*value 1*

Did you notice that we snuck in a new concept here? Line continuation can be accomplished by ending any line with the backslash **** character. When reading input lines, if the parser detects the backslash as the last character on the line, it will read the next line and append it to the current line, and then return the entire thing as though it were one long line. This helps you produce more readable markdown, especially when declaring variables or creating long textual content.

Well that wraps the section on **@var**! Most of the concepts we covered here apply to all namespaces, unless they say otherwise when we cover them. Let's move on to the next namespace, **@html**, arguably the reason we are all here, since HTML is the primary reason that **smd** exists!

Table of Contents

@html Namespace

The **@html** keyword is used to construct variables that represent HTML elements for your documents. These variables are stored in the **@html** namespace, and can have names that are identical to variables in other namespaces, although it is normally recommended that you avoid doing that.

Most of the features and concepts just discussed for **@var** variables also applies to **@html** variables. If you haven't read that chapter yet, you should, as this one assumes that you have read and understand those concepts, and will build upon them.

@html Syntax

@html command syntax

@html _id="name" [_tag="tagname" [...]]

All of the built-in attributes previously documented for **@var** are supported for **@html** variables, and a new one **_tag** has been introduced. _tag is used to set the HTML element name for the variable. If _tag is not specified, it will be set to the same value as _id. Let's define a few variables, and then examine them.

Defining an @html variable

```
@html _="blockquote" cite="url-goes-here"
@html _id="bquote" _tag="blockquote"
@html _id="bq" _inherit="blockquote"
```

In the preceding example, we've defined three **@html** variables: **blockquote**, **bquote** and **bq**. Let's use **[code.dump]** to take a look at their declarations.

Definition of *blockquote*, *bquote* & *bq* variables

blockquote=
cite=url-goes-here
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=blockquote

bq=
cite=url-goes-here
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=blockquote

bquote=
_tag=blockquote
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>

First, take a look at **blockquote**. **_tag**, although we didn't specify it, has been set to **blockquote**. In many cases, this is what you want, so not having to specify it is a nice shortcut when declaring variables. Also, notice how **_format** has been set to *<{{self._tag}}{{self._public_attrs_}}><{{self._tag}}>*.

Given that, if we write *[blockquote]* it will emit ***<blockquote cite="url-goes-here"></blockquote>***.

The first thing you'll notice is that there isn't a way to get content between the opening and closing tags, so on the surface, this doesn't look very useful. Enter the special attributes **<** and **>**, supported by all **@html**-based namespaces.

Now, if we write **[blockquote.<]** it will emit <blockquote cite="url-goes-here">. Conversely, if we write **[blockquote.>]**, it will emit </blockquote>. So now all we need is to specify a value for the **cite** attribute when we declare the open tag, add some content, and specify the closing tag. Let's see how it all comes together.

Adding content to the blockquote HTML variable

```
[blockquote.<(cite="https://google.com")]
Google has always had the "I'm Feeling Lucky" button just below it's search input box,
which can be used to automatically follow the first returned link in your search.
[blockquote.>]
```

This is what will be rendered by the browser:

> Google has always had the "I'm Feeling Lucky" button just below it's search input box, which can be used to automatically follow the first returned link in your search.

Keep in mind that the formatting of the **blockquote** tag is defined by your Browser, or the CSS specified for you document.

Let's return to looking at the **bquote** and **bq** variables. With **bquote**, notice it doesn't have the **cite** attribute, because it wasn't specified. Attributes can always be added to an **@html** variable on the fly, it isn't required that they be added when the variables are defined. And remember, most HTML elements support the global attributes (e.g. class, style, id, etc.), and if any of these are declared as public attributes on an **@html** variable, they will automatically be output whenever the opening tag of an element is emitted by the parser.

bq is identical to **blockquote**, because the **_inherit** attribute was specified at declaration time. Recall from the **@var** section, **_inherit** is used to add all attributes of the named variable to this new declaration. And, any of the underlying attributes can be overridden in the new declaration by simply specifying them again. For example, if we wrote:

@html _id="bq" _inherit="blockquote" cite="different-default-citation"

Then this is what the definition of **bq** looks like in memory:

New definition of *bq* variable

bq=
cite=different-default-citation
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=blockquote

@html Variable names

Variable names in the **@html** namespace are restricted to the same requires as the **@var** namespace.

@html Built-in Attributes

In addition to all of the built-in attributes supported by **@var**, **@html** variables have several built-ins that are unique to it:

Attribute Name	Description
_tag	The HTML element name of the variable
<	The opening tag for this variable
>	The closing tag for this variable
<+	The opening tag with embedded quotes escaped.

We've already seen these in action above, so no need to discuss further with the exception of the **<+**, which will output the opening tag with escaped quotes in the public variable definitions.

Examples

```
[html.blockquote] = <blockquote cite="https://google.com"></blockquote>
[html.blockquote.<] = <blockquote cite="https://google.com">
[html.blockquote.>] = </blockquote>
[html.blockquote.<+] = <blockquote cite=\"https://google.com\">
[html.blockquote._id] = blockquote
[html.blockquote._tag] = blockquote
[html.blockquote._public_keys_] = cite,
[html.blockquote._private_keys_] = _format, _tag,
```

@html Miscellaneous Examples

Here are just a few more examples to help drive home your understanding of the declaration and usage of variables in the **@html** namespace. Let's start by examining the contents of the **[sys.imports]/html.md** file:

Contents of html.md builtins

```
@html _id="html" _tag="html"

// <head> related tags
@html _id="head"
@html _="title" _format="{{self.<}}{{self._t}}{{self.>}}" _t="Default Title"
@html _="meta" _format="{{self.<}}"
@html _="meta-charset" _inherit="meta" charset="UTF-8"
@html _="meta-viewport" _inherit="meta" \
        name="viewport"\
        content="width=device-width, minimum-scale=1.0, maximum-scale=1.0, user-
scalable=0"
@html _="link" _format="{{self.<}}"
@html _="script"
@html _="css" _inherit="link" rel="stylesheet" href="your-stylesheet.css"
@html _="js" _inherit="script" src="your-javascript.js"

// <body> related tags
@html _="body"
@html _="code"
@html _="div"
@html _="em"
@html _="h" 1="<h1>{{self.t}}</h1>"\
            2="<h2>{{self.t}}</h2>"\
            3="<h3>{{self.t}}</h3>"\
            4="<h4>{{self.t}}</h4>"\
            5="<h5>{{self.t}}</h5>"\
            6="<h6>{{self.t}}</h6>"\
            _format="{{self._help}}"\
            _help="*{{self._}}.#(t=\"your heading\")*[bb]\
[sp.4]Emit an HTML Heading[bb]\
[sp.2]**Parameters**[b]\
[sp.4]t - text for heading[bb]\
[sp.2]**Methods**[b]\
[sp.4]# - integer between 1 and 6[bb]\
[sp.2]**Examples**[b]\
[sp.4]h.1(t=\"My [E.ast]h1[E.ast] heading\")"

@html _="li"
@html _="ol"
@html _="p"
@html _="pre"
@html _="span"
@html _="strong"
@html _="table"
@html _="td"
@html _="th"
@html _="tr"
@html _="ul"

// useful list styles
@html _="ulist" _inherit="ul" class="ulist"
@html _="ulistplain" _inherit="ul" class="ulist-plain"
@html _="olist" _inherit="ol" class="olist"
@html _="olist_template" _inherit="ol"\
        _format="@@{{self._inline}}"\
        _inline="<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>"
@html _="olistAlpha" _inherit="olist_template" class="olist-Alpha"
@html _="olistalpha" _inherit="olist_template" class="olist-alpha"
@html _="olistRoman" _inherit="olist_template" class="olist-Roman"
@html _="olistroman" _inherit="olist_template" class="olist-roman"
@html _="olistGreek" _inherit="olist_template" class="olist-Greek"
@html _="olistgreek" _inherit="olist_template" class="olist-greek"
```

You can see a combination of syntaxes used in declaring the builtins for **@html**. Most are very simple, just the name and tag (when necessary), others have a few have additional options. If you begin to examine other system builtin files, you will see more complex declarations. Remember **helpers.md** from the earlier chapter Helper built-ins?

Many times, you may find that taking a system provided builtin as a starting point, using **_inherit** to pick up all of its attributes, and then extending it by adding additional attributes or even changing the behavior by modifying an existing attribute is a great way to get exactly what you need. And this brings us to the end of **@html**. From here, spend some time experimenting with this namespace, and maybe try modifying existing or extending something to get a better feel for how they work.

One last thing before we take a look at the **@link** namespace. Do not use **self.<** or **self.<+** in any public attributes that are declared on **@html** variables. Doing so will cause an infinite loop in the **Markdown** class, because when it attempts to markdown **<** or **<+**, it will enumerate the public attributes, expanding any markdown in those attributes before rendering the content. However, in this case, the public attribute contains the **self.<** builtin, and it recurses to process that, only to find itself repeating the same thing. Long story short, it will throw an exception after twenty-five recursive calls to attempt to evaluate the expression. Here's a really short example that you can run in the interactive shell:

Sample infinite recursion using self.< in public attribute

```
$ smd -nd
@link _="fu" boom="{{self.<}}"
[fu.boom]
```

If you try the two statements above in an interactive **smd** session, you will quickly see the issue. You may want to add **@debug toggle="markdown"** just before the **[fu.boom]** so you scroll up to get a dump of markdown recursion loop (a trick discussed in the chapter on debugging.)

Table of Contents

@link Namespace

The **@link** namespace is built atop the **@html** namespace, and as such, it inherits all the same characteristics of things defined in the **@html** namespace. In this chapter, we will take a closer look at the **@link** namespace and what it has to offer.

Link Styles

There are two different styles of links built-in to **smd** which map to standard HTML links:

1. Hyperlinks - Links created using the HTML **a** tag with the **href** attribute. e.g. *content*
2. Bookmarks - Links created using the HTML **a** tag with the **id** attribute. **

As you can see, these links are both based on the HTML **a** attribute, and the distinction lies in how they are rendered within a document. The sections that follow describe how each link type is creating using the built-in factories specified in the ***sys.imports/link.md*** file.

Creating Hyperlinks

Hyperlinks are created using the **@link** namespace in **smd**, while specifying the attributes desired to describe the link. The **@link** namespace is a subclass of **@html**, and therefore it shares all the same characteristics, such as the < attribute to emit the open tag HTML and > to emit the close tag HTML. For more information on the **@html** namespace, refer to the @html chapter in the user manual.

You can create a variable in the **@link** namespace using the following syntax:

Creating variables in @link namespace

```
@link _="name" href="http://example.com"
```

If you then write **[link.name]**, the parser will emit ****. Such a simple example isn't very useful, however, since we need the ability to insert content between the anchor tag. This can be accomplished by using the special attributes < and >, as follows:

Emitting HTML links using special attributes

```
[link.name.<]content here[link.name.>]
```

Now, the parser emits: **content here**, which is a bit more useful, but seemingly rather complex to to use in practice. In light of this, two special built-ins are provided in order to ease the creation and usage of hyperlinks. The first is a link template,

[link._template_], which can be inherited during creation of new links, and the second is a factory, **[link.ln_factory]**, which can be used to easily create a hyperlink with minimal input, and which has several useful attributes that can be used during rendering. Let's take a look at each of these now.

Going back to our original example, if we add **_inherit="_template_"** to the definition e.g.:

Inheriting attributes from a template

```
@link _="name" href="http://example.com" _inherit="_template_"
```

We now have several new attributes available to use. For example, if we write **[link.name._asurl]**, the parser emits **http://example.com** which renders as http://example.com.

If we use just the variable name like we did before e.g. **[link.name]**, we get this: Usage: link.name(text="Text to Link"). As you can see, the default text for the link is actually a usage statement, requesting that we add the **_text=** parameter like so: **[link.name(_text="my web site")]**, which then yields: my web site.

If you want to use different content with the same link, then specifying the **_text=** parameter provides a convenient way to do that. For example, if I change the parameter to **_text="My Really Cool Website"**, I will get: My Really Cool Website.

There is one side effect to overriding the default values for attributes on variables in all namespaces except the **@code** namespace. When a default attribute is overridden in a call, it changes the default value of that attribute for subsequent calls!

For example, if I write **[link.name]** again, then I will get the last value specified for _text, e.g. My Really Cool Website.

> Keep that in mind as it affects parameter passing in all of the namespaces except **@code**.
>
> In **@code**, default values for parameters can only be overridden when the variable is defined with **@code** or updated with **@set**. Anything passed via a call will be persisted for that call, and then revert to the original value.

You can also set the initial value for **_text** when the link variable is initially defined. For example, if we write:

Specify initial value for _text during declaration

```
@link _="name" href="http://example.com" _inherit="_template_" _text="my default title"
```

And now if I write **[link.name]**, I will get the new default value for _text, i.e.: my default title, without having to specify **_text** on the initial usage of the link.

Using the link factory to create hyperlinks

Before we leave the section on hyperlinks, let's a have a look at a better shorthand for creating links, the built-in link factory **ln_factory**. This built-in allows you to easily create a new link with minimal parameters:

Using the link factory to easily create hyperlinks

```
[ln_factory(nm="sample" hr="http://example.com" t="my default title")]
```

And now, if I write **[sample]**, I will get: my default title. Notice how in this example, we left off the namespace **link.**. Recall that namespaces allow identifiers to be reused, and the value of a duplicated identifier in different namespaces is unique within each namespace. This ambiguity can lead to unexpected expansion since the if no namespace is supplied, the different namespaces are searched attempting to resolve the identifier. If the parser finds a match before it searches the intended namespace, then that is what it will use.

Given that, it's always a good idea to specify the namespace except in the most simple of documents, to avoid incorrect expansion down the road. Thus, if I write **[link.sample]**, I will get the value as before: my default title.

As you might expect, **ln_factory** inherits from **link._template_**, and as such, it inherits attributes including **_asurl**. For example, ***[link.sample._asurl]*** which expands to **http://example.com** and renders as: http://example.com.

One other feature of the **link._template_** are the **_qlink** and **_qtext** attributes. These allow you to specify alternative link text to the same **href**. For example, if I add **_qtext="alternative link text"** when the variable is declared, and then write **[link.sample._qlink]**, I will get: alternative link text.

You can also specify the **_qtext** using the special **_null_** syntax: **[link.sample._null_(_qtext="alt2 link text")]**. Doing so then causes **_qlink** to emit: alt2 link text.

And of course, you can create them on the fly by simply specifying the **_qtext** parameter when requesting the **_qlink** attribute value.

Specifying _qtext as parameter to _qlink

```
[link.sample._qlink(_qtext="yet another descriptive link text")]

// Emits the following on-the-fly
yet another descriptive link text
```

As in the **@html** namespace, any valid HTML attribute can be specified when creating link variables. Like **@html**, attributes that begin with an underscore _ are considered *private* and attributes that begin with a letter are considered *public*. When HTML tags are emitted as part of variable expansion, all public attributes are written as part of the opening tag. Let's see how this works.

If we add **title="my link title here"** when we define the link variable **link.sample** we were just using, and then emit the code using any of the emitters **(link.sample, link.sample._asurl, link.sample._qlink)**, the HTML code includes a **title=** attribute every time the anchor opening tag is written. Let's see how that looks, first, we'll declare it:

Adding public attributes to an @link variable

```
// You can add attributes with either @set or using the _null_ method

@set _="link.sample" title="link title is now set"
Now [link.sample] emits: my default title (hover to see the title)

[link.sample._null_(title="my link title here")]
Now [link.sample] emits: my default title (hover to see the title)
```

Let's take a closer look at what the parser emits and what the browser renders for each link method:

[link.sample] emits *my default title* which renders as: my default title

[link.sample._asurl] emits *http://example.com* which renders as: http://example.com

[link.sample._qlink] emits *yet another descriptive link text* which renders as: yet another descriptive link text

If you hover over any of the preceding links, the browser will show "**my link title here**" after a couple of seconds.

By using this same feature, you can add custom CSS styling via **style=** along with other valid HTML attributes to any *@link*, *@image* or *@html* namespace tag.

As one last example, we will add **style="font-size:2em"** to the link.sample variable, and then emit the same 3 links as before:

Let's write the following markdown: **[link.sample._null_(style="font-size:2em")]** first.

And now, **[link.sample]** emits *my default title* which renders as:

my default title

Similarly, **[link.sample._asurl]** and **[link.sample._qlink]** render as:

http://example.com

yet another descriptive link text

This should give you a pretty good idea of how you can use hyperlinks in your smd documents, and so this wraps up the section on creating them using the built-ins provided in ***sys.imports/link.md***.

Creating Bookmarks

Bookmarks are a special type of link used within an HTML document. There are two logical parts to a bookmark, the ***ID*** and the actual reference to the ID, which is done by writing ***#ID***. Within smd, bookmarks are implemented with the **@link** namespace, and are usually created using the Bookmark factory, **bm_factory**, which is a built-in link variable. **bm_factory** uses the **bm_template** variable in order to provide a simple abstraction for creating and using bookmarks within an HTML document.

> ***NOTE***: Because the **bm_template** isn't useful outside the context of the **bm_factory**, we are going to focus on the latter only, since it is what you will use to create and use bookmarks.

Similar to it's counterpart **ln_factory**, has the following syntax:

Using the bookmark factory to create in-document links

```
[bm_factory(nm="bookmark1" t="my bookmark text")]
```

The **bm_factory** does not need the **hr=** parameter, however, because the href is constructed using only the bookmark name, since it refers to a location within the current document.

Once the bookmark is created using the syntax above, hyperlinks within the current document are created by using the **.link** attribute, and the anchor location is emitted using simply the variable name. Let's look at an example.

Within your document, use the **[link.bookmark1]** to place the anchor at the appropriate location. In our case, I've placed the anchor just below the variable dump below, so that it's easy to test. To create a hyperlink to that location, use the syntax: **[link.bookmark1.link]**, as I have done right here: my bookmark text. <-- Click on that link and see that it takes you to the location below.

I'm going to dump the various **@link** variables we've been creating to create some space between the bookmark link I just emitted above and the location where the heading is that I dropped the anchor.

@link variable definitions and associated @html elements

bookmark1=
id={{self._}}
_format=@@ {{self._inline}}
_link_o=
link={{self._link_o}}{{self.text}}
text=my bookmark text
_inline=
_qlink={{self._link_o}}{{self._qtext}}
_qtext=SETMETOLINKTEXT
_tag=a

name=
_format={{self.<}}{{self._text}}{{self.>}}
_text=my default title
href=http://example.com
_asurl={{self.<}}{{code.esc_angles.run(url="{{self.href}}")}}{{self.>}}
_qlink={{self.<}}{{self._qtext}}{{self.>}}
_qtext=SETMETOLINKTEXT
_tag=a
_inherit=_template_

sample=
_format={{self.<}}{{self._text}}{{self.>}}
_text=my default title
href=http://example.com
_asurl={{self.<}}{{code.esc_angles.run(url="{{self.href}}")}}{{self.>}}
_qlink={{self.<}}{{self._qtext}}{{self.>}}
_qtext=yet another descriptive link text
_tag=a
title=my link title here
style=font-size:2em

This is where I dropped the anchor for bookmark1

Before we leave bookmarks, it is worth mentioning that this factory also supports the **_qlink** and **_qtext** attributes, just like standard links do. So, using our prior bookmark, here is the output when using them:

[link.bookmark1._qlink("my alternate bookmark name")] --> my alternate bookmark name

That's about all there is to using the smd built-ins for creating HTML bookmarks within a document.

Table of Contents

mailto links

You can create mailto: links in your document too, which enables users to click on a link to automatically compose an email addressed to the specified email address. **smd** will encode the entire mailto: link URL using a mix of decimal and hexadecimal HTML entities as a deterrent to spam bots that mine email addresses from HTML documents. Here's the syntax for a *mailto:* link:

mailto Link Syntax

@link _id="name" href="*mailto:*you@yourdomain.com"

// or use the link factory
[link.ln_factory(nm="sample" hr="mailto://example.com" t="my default title")]

Let's create two sample links using each method and see how they can be used in your markdown:

using the mailto: prefix on @link declarations

```
    @link _="email_me" href="mailto:me@domain.com"
    [ln_factory(nm="feedback" hr="mailto:user@mydomain.com?subject=feature%20feedback"
t="Send feedback")]
```

I've defined the previous examples inline in the user docs, so now we can use them by using either of these methods:

[link.email_me.<]email_me[link.email_me.>] emits this:

email_me

which the browser renders like this: email me.

Wait, what the ... happened to the href?

<a href="mailto:me@domain.com">

How come the **href** on the link is all jacked up? Actually, this is intentional! It encodes the email address in a way that makes it a bit more challenging for **spambots** and other web scraper tools to detect an email address. Not impossible, but definitely more challenging! In addition, every time the parser runs the output will be slightly different, yet the link and the behavior remain consistent and correct! Just a cool side benefit to how **smd** parses and encodes links that use a **mailto** prefix on the **href**.

Getting back to **mailto** links, let's take a look at the other example, the one created with the link factory. The variable name for it is **feedback**, and to use the link, all we need to write is **[feedback]**, which is much easier and cleaner than having to use the link open and close tags as before.

When we do that, the browser will display Send feedback, which is exactly what we need. If you'll recall from the discussion on the link factory before, you also have access to the **_qlink** attribute, so we can create custom alternate text on the fly.

For example, if we write **[feedback._qlink(_qtext="I would love to get your feedback!")]**, then this is what the browser would display: I would love to get your feedback!

And so we come to the end of the **@link** chapter. Built upon the **@html** namespace, **@link** offers some cool features for creating and using hyperlinks if your markdown documents. Next up, is the **@image** namespace.

Table of Contents

The @image Namespace

The **@image** keyword is used to declare variables that are used to include images in your document. Basically, **@image** is a convenient way to abstract the **<img>** HTML tag.

Despite being an abstraction of the HTML element **img**, **@image** is not based on the **@html** namespace. Given that, the underlying semantics do not support the **@html** built-in attributes such as **.<**, **.>** and **.<+**. The full syntax is:

@image Syntax

@image _id="imagename" src="pathtoimage" alt="" _private="val"

Here's how it works. Like all namespaces, *_id="imagename"* is what names the variable in the **@image** namespace. You will use that name to cause the **<img>** tag to be emitted in your document. Also recall that attributes that start with an underscore _ are considered private, and are not included in the generated **@image** HTML code. So, if you were to write:

Creating variables in @image namespace

```
@image _id="img1" src="path/foo.jpg" alt="my text for alt" class="myclass" _private="My
private note"
```

and then write:

[img1]

The code that would be inserted would be:

****

Note that _id wasn't included, nor was _private. However, I can reference them both using the syntax:

[img1._id] which emits *img1* and **[img1._private]** which emits My private note

This also works to reference the normal attributes. e.g. **[img1.class]** which emits myclass.

And as is the case for all namespaces, if there's ambiguity in the names used in different namespaces, you can add the namespace prefix to clarify. For example, ***image.*** in front of the name to force the correct namespace to resolve. For example, if I write the following two lines in my document:

```
@var img1="my variable img1 in @var namespace"
@image _id="img1" src="foo.png"
```

Then, when I write **[var.img1]** the parser will emit *my variable img1 in @var namespace* and when I write **[image.img1]** the parser emits **.

Let's go ahead and include an inline image to see **@image** in action. I will write:

```
@image _="shot1" src="[sys.root]/docs/import/shot1.jpg" alt="shot 1" style="width:30%"
```

When I write **[shot1]**, the parser emits ** and the browser renders:

Pretty straightforward, don't you think? Okay, now it's time to take a look at the system provided builtins for **@image**, so we can tap into some additional flexibility when rendering images with **smd**.

@image builtins

The **@image** namespace provides a limited set of builtins to get you started with using images in your markdown. These builtins are a good starting point for your own image styling, so let's begin with a quick overview of the builtins in **sys.imports/image.md**.

IMG_DEF contains four different default styles, each of which can be selected using the **IMG_STYLE** attribute shortcuts:

[IMG_STYLE.inline]
[IMG_STYLE.inline_border]
[IMG_STYLE.block]
[IMG_STYLE.block_border]

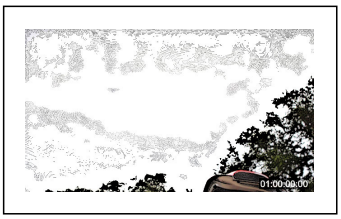
In addition, four fixed size attributes and one custom size attribute is provided via the **IMG_SIZE** declaration:

[IMG_SIZE.thumb]
[IMG_SIZE.small]
[IMG_SIZE.medium]
[IMG_SIZE.large]
[IMG_SIZE.custom(w="##%"))]

For a quick example of the size and styling shortcuts, let's write the following markdown in this document and see what we get:

```
[IMG_SIZE.thumb]
@image _="shot1" src="[sys.root]/docs/import/shot1.jpg" style="
{{IMG_STYLE.inline_border}}"
[shot1]
```

And here is what the browser renders:



Let's take a closer look at the builtins declared in **image.md** to assist with using images in your documents. We will start by looking at the built-in help for each builtin, and then we'll move on to using them in our document.

IMG_DEF Help:

IMG_DEF

Defines common image styling options.

Not normally accessed directly. Used by **IMG_SIZE** and **IMG_STYLE** as helpers.

Common Attribute Values
_i_width - width: CSS value
_b_size - border: CSS size value
_b_type - border: CSS type value

Common Styling Defaults
img_border_style - CSS styling for bordered image
img_inline_style - CSS styling for borderless image

Common Image Style Options
img_st_inline - CSS styling for inline image
img_st_inline_border - CSS styling for inline bordered image
img_st_block - CSS styling for block image
img_st_block_border - CSS styling for block bordered image.

IMG_SIZE Help:

IMG_SIZE

Provides methods for common image sizes.

Use them when you want to change the default image size used in the **IMG_STYLE** attributes.

Common Methods
thumb - Sets **IMG_DEF._i_width** to 20%
small - Sets **IMG_DEF._i_width** to 40%
medium - Sets **IMG_DEF._i_width** to 70%
large - Sets **IMG_DEF._i_width** to 90%
custom(w) - Sets **IMG_DEF._i_width** to **w**

Examples
[IMG_SIZE.small] - Sets **IMG_DEF._i_width** to **20%**
[IMG_SIZE.custom(w="50%")] - Sets **IMG_DEF._i_width** to **50%.**

IMG_STYLE Help:

IMG_STYLE

Several styling attributes for common image styles.

These attributes are used to obtain various CSS styles for an @image variable.

Common Attributes
inline - Returns **IMG_DEF.img_st_inline**
inline_border - Returns **IMG_DEF.img_st_inline_border**
block - Returns **IMG_DEF.img_st_block**
block_border - Returns **IMG_DEF.img_st_block_border**
Examples
[IMG_STYLE.inline] - Returns the CSS to style an inline image
[IMG_STYLE.block_border] - Returns the CSS to style a block image with a border.

The builtins you normally use in your documents are **IMG_SIZE** and **IMG_STYLE**. **IMG_SIZE** is used to set the size of the image that will be displayed in the document. Before we get too far, though, we need to actually declare an **@image** variable that we can experiment with. Let's begin with the following declaration:

Declare an image variable

```
// Set image size to large, declare myshot var and render

[IMG_SIZE.large]

@image _id="myshot" src="[sys.imports]/avs/needshot.png" style="
[IMG_STYLE.inline_border]"

[myshot]
```

And when we do that, here's what we get:

Okay, not too complicated thus far. The first thing to note is that the image size takes up almost the entire document window. This is because the width is currently set to **90%**. So let's go ahead and set the size to **[IMG_SIZE.small]**, and then render the image again:

Change image size to small and render again

```
// Set image size to small again and just render
[IMG_SIZE.small]
[myshot]
```

Wait, this looks the same! What's going on? Let's begin by taking a look at the declaration for **myshot**:

Declaration of image.myshot

```
myshot=
  src=/Users/ken/Dropbox/shared/src/script/smd/smd/import/avs/needshot.png
  style=margin-left:auto;margin-right:auto;width:90%;border:1px solid;padding:1em;
  _format=<{{self._tag}}{{self._public_attrs_}}/>
  _tag=img
```

Okay, looks like the issue is that **style** is hard-coded to the **inline_border** style. So, if we redeclare our @image variable, but this time use either "**[!IMG_STYLE.inline_border!]**" or "**{{IMG_STYLE.inline_border}}**", either of which will cause the parser to not evaluate the variable until it is used, that should allow us to override it. So basically, here's the markdown:

Declare an image variable

```
// Declare myshot, set image size to small, render
@image _id="myshot" src="[image_path]/needshot.png" style="[!IMG_STYLE.inline_border!]"
[IMG_SIZE.small]
[myshot]
```

And now if I write **[IMG_SIZE.thumb]** followed by **[myshot]** I will get:

Change image size to thumb and render again

```
// Set image size to thumb and just render
[IMG_SIZE.thumb]
[myshot]
```

Let's review the new definition of **myshot** to see how this change affected the variable definition:

Declaration of image.myshot

```
myshot=
  src=/Users/ken/Dropbox/shared/src/script/smd/docs/samples/image/needshot.png
  style=[IMG_STYLE.inline_border]
  _format=<{{self._tag}}{{self._public_attrs_}}/>
  _tag=img
```

You can see that **style=** is now set to **[IMG_STYLE.inline_border]** instead of being hardcoded. Using the [! !] around a variable name in an attribute declaration causes the parser to delay evaluation of the variable until it is actually used. Note that you would get the same effect by surrounding the variable/attribute name with curly braces, i.e. {{ }}.

You can see more of these builtins and the **@image** support in action by reviewing the samples that are included in the user guide.

Table of Contents

The @code Namespace

The **@code** keyword is used to construct a very specialized type of variable for your documents. Like all the other namespaces, these variables can have names that are identical to variables in other namespaces, although it is normally recommended that you avoid doing that.

@code variables are probably best described as macros, and although the other namespace variables could also be viewed this way, **@code** variables are much more powerful because they provide a way to construct emitters written in Python! **@code** macros are used heavily throughout the system builtins to achieve some of the more powerful features of **smd**.

> If you are not familiar with Python and Python programming, then you may want to skip this section; it isn't for you. You can still do quite a bit with **smd** without having to write extensions in Python.

@code Syntax

> @code command syntax
>
> **@code _id="name" src="value" type="eval | exec" [...]]**
>
> NOTE: either underscore '_' or '_id' can be used to name the variable.

@code variables require that the _ or ***_id*** special attribute name be used to specify the variable name. In addition to that requirement, **@code** variables also require two specific public attributes: ***src*** and ***type***. As you might expect, ***src*** contains the Python source code that will be executed, and ***type*** specifies either *eval* or *exec*, which are underlying Python concepts. Refer to the Python documentation for the specifics on *eval* vs. *exec* when writing Python code. Let's take a look at an example declaration:

Declaring an @code variable

```
@code _id="echo" src="print('{{self.text}}')" type="eval" text="*text to print*"
```

OR

```
@code _id="echo" src="print('$.text')" type="eval" text="*text to print*"
```

> **@code** declarations support the shorthand notation **$.attrname** which is identical to **{{self.attrname}}**. They can be used interchangably on @code declarations within the **src** attribute only.
>
> Keep in mind, however, that there is a subtle semantic difference between the two syntaxes. In addition to only being valid in the context of the **src** attribute value, if a **$.attrname** reference value contains delayed expansion markdown i.e. {{var.attr}}, it will simply be substituted, it will not be marked down.

Let's go ahead and write that declaration now using the first style for referencing attributes **{{self.text}}**, and then we'll type **[code.echo]** and it will emit: *text to print*. So far, so good, not too terribly complicated, right?

Okay, let's do it again, only this time, we'll pass specific text when we write the markdown, like so: **[code.echo(text="### My Inline Heading")]**, which will give us: ### My Inline Heading. Okay, that's weird, I was expecting an HTML level 3 heading. What happens if I write it again, only this time I'll put the macro at the start of a line:

My Inline Heading

Aha! Interesting. While that might seem unique to **@code** variables, it is not. You can write the same thing as an **@var** variable, and it will behave similarly. So if it behaves like **@var** variables, then I should be able to write **[code.echo]** again (at the start of the line, of course), and this time it should print the heading without me specifying it (according to the rule that parameters to markdown variables are sticky rule). Let's give that a shot:

text to print

Wait what? It defaulted back to the original declaration's value for **text**. And there's another difference about **@code** variables. Parameters passed to a markdown instance do not become the new default values for the attributes associated with that variable. So how do I override it without redeclaring the variable? With **@set**. Let's try that:

Changing @code attribute defaults

```
@set _="code.echo" text="### My Inline Heading"
[code.echo]
```

This causes the following to be emitted:

My Inline Heading

And that's how you change the default values for **@code** variable attributes.

Additional attributes on @code variables

@code variables have several unique attributes:

_code - used by **@code** to store the compiled Python code.

params - used by store the parameters to the current markdown instance
_last - stores the result of the last time the macro was run
run - used to invoke the macro. **[code.echo.run]** is the same as **[code.echo]**
src - holds the source code in string form.
type - stores the Python type used for evaluating/compiling the code.

Let's take a quick look at the definition for **code.echo**:

Definition for code.echo

echo=
src=print('{{self.text}}')
type=eval
text=### My Inline Heading
_code=*<code object>*
_last=### My Inline Heading
params={}

Accessing these attributes is done the same way as any other namespace. For example:

[code.echo.src] contains *print('{{self.text}}')* which emits *print('### My Inline Heading')*.

[code.echo._last] contains *### My Inline Heading* which emits *### My Inline Heading*.

_last just provides a convenient way to access the value output of the last time the code was run by the Python interpreter, avoiding having to run it again to get the same value. You might find this useful at times.

Let's update our **echo** variable by adding the following **help** attribute and changing the default for **text** to reference it (by the way, this is how most of the **@code** system built-ins are declared):

Adding a default help string to an @code variable

```
@set _="code.echo" text="{{self._help}}" _help="{{self._}}(\"*text to print*\")"

[code.echo._help] emits echo(text="text to print")
[code.echo.?] emits echo(text="text to print")
```

If you create your own **@code** macros, it's a good idea to create a default help string for it so that anyone trying to use it will see how it works. That someone might just be you, and you'll be glad you did!

Let's take a look at one more declaration:

@code _id="esc_angles" type="eval" src="print('{{self.url}}'.replace('<', '<').replace('>', '>'))" url="{{self._help}}" _help="Usage: code.esc_angles.run(url="text to escape"

And now we write the following markdown:

```
// Because url has not been set yet, it will default to the built-in help string

[code.esc_angles] emits   <em>esc_angles(url="<text to escape>")</em><br /><br
/>    <strong>url</strong> = string to operate on<br /><br />    Replace all
occurrences of <em><</em> with <em>&lt;</em> and <em>></em> with <em>&gt;</em> in
string <strong>url</strong>

[code.esc_angles(url="<https://google.com>")] emits <https://google.com>.
```

As it turns out, **code.esc_angles** is actually one of the built-in **@code** macros provided by **smd**.

Definition for code.esc_angles

esc_angles=
type=eval
src=print('{{self.url}}'.replace('<', '<').replace('>','>'))
url={{self._help}}
_code=*<code object>*
_last=<https://google.com>
params={}

Reviewing the builtin help for esc_angles

```
// Value of _help has been suppressed from the dump above. It follows:

_help=  <em>{{self._}}(url="&lt;text to escape&gt;")</em><br /><br
/>    <strong>url</strong> = string to operate on<br /><br
/>    Replace all occurrences of <em>&lt;</em> with <em>&amp;lt;
</em> and <em>&gt;</em> with <em>&amp;gt;</em> in string <strong>url</strong>

[esc_angles._help] emits
  esc_angles(url="<text to escape>")

    url = string to operate on

    Replace all occurrences of < with &lt; and > with &gt; in string url

[esc_angles.?] emits
  esc_angles(url="<text to escape>")

    url = string to operate on

    Replace all occurrences of < with &lt; and > with &gt; in string url
```

We will see the rest of built-in macros later in the chapter.

Inheriting attributes

@code variables do **not** support the **_inherit** attribute when declaring new variables.

@code Variable names

Variable names in the **@code** namespace are restricted to the same requirements as the other namespaces, e.g. Names.

@code Built-in Attributes

In addition to the specific attributes covered above, **@code** variables support the underlying built-in attributes of all namespaces. They can be useful to assist in debugging complex declarations. Refer to the Attributes section for all of the details of the built-in attributes.

@code Built-in Macros Help Section

@code provides a number of useful builtins to assist with creating more complex markdown documents. In this section, we'll take a closer look at what's available, and see how they can be used in your own documents.

Escaping and Encoding

Macro: code.esc_angles

esc_angles(url="<text to escape>")

url = string to operate on

Replace all occurrences of < with *<* and > with *>* in string **url**

Dump of code.esc_angles:

esc_angles=
type=eval
src=print('{{self.url}}'.replace('<', '<').replace('>','>'))
url={{self._help}}
_code=*<code object>*
_last=<https://google.com>
params={}

Macro: code.escape

escape(t="text to escape")

t = string to operate on

Same as code.esc_angles, but also replaces *&* with *&*

Dump of code.escape:

escape=
type=exec
src=from .utility import HtmlUtils;print(HtmlUtils.escape_html('{{self.t}}'))
t={{self._help}}
_code=*<code object>*
_last=
params={}

Macro: code.escape_var

escape_var(v="var_to_esc")

v = existing variable / attribute to escape

Same as code.escape, but takes an existing variable or attribute name to operate on.

Dump of code.escape_var:

escape_var=
type=exec
src=from .utility import HtmlUtils;print(HtmlUtils.escape_html_var('{{self.v}}'))
v=self._help
_code=*<code object>*
_last=
params={}

Macro: code.encodeURL

encodeURL(t="https://www.url.com?x=encode me")

t = a URL string to be encoded

Replaces all blanks in URL string e.g. " " with "*%20*"

NOTE: If **t** begins with **mailto:** the entire string is encoded using a mix of dec/hex HTML character entities

Dump of code.encodeURL:

encodeURL=
type=exec
src=from .utility import HtmlUtils;print(HtmlUtils.encodeURL('{{self.t}}'))
t={{self._help}}
_code=*<code object>*
_last=raw src=from .utility import HtmlUtils;print(HtmlUtils.encodeURL('{{self.t}}'))

params={}

Macro: code.encode_smd

encode_smd(t="smd markdown to encode")

t = an smd string that will be encoded for display

Use < > for *[ ]*, *2{* for *{{*, *2}* for *}}*, *2+* for ++ and *2~* for ~~.

The entire list of replacements is:
< becomes *[*
> becomes *]*
2{ becomes *{{*
2} becomes *}}*
* becomes ***
@ becomes *@*
2+ becomes *++*
2~ becomes *˜˜*

NOTE1: Do not use square brackets, curly braces, double plus or double tilde directly, because the parser will replace them with markdown.

NOTE2: You need to use *[E.lp]* for *(* and *[E.rp]* for *)*, because parenthesis are used by the markdown to identify parameters to the macros.

Dump of code.encode_smd:

encode_smd=
type=exec
src=from .utility import HtmlUtils;print(HtmlUtils.encode_smd('{{self.t}}'))
t={{self._help}}
_code=*<code object>*
_last=@code
params={}

Macro: code.encode_smd_var

encode_smd_var(v="var_to_encode")

v = existing variable / attribute to markdown as smd

Same as code.encode_smd, but takes an existing variable or attribute name to operate on

Dump of code.encode_smd_var:

encode_smd_var=
type=exec
src=from .utility import HtmlUtils;print(HtmlUtils.encode_smd_var('{{self.v}}'))
v=self._help
_code=*<code object>*
_last=raw src=from .utility import HtmlUtils;print(HtmlUtils.encode_smd_var('{{self.v}}'))
params={}

Miscellaneous Helpers

Macro: code.datetime_stamp

datetime_stamp(fmtstr="%Y%m%d @ %H:%M:%S")

fmtstr is a Python strftime format string. The default format is: %Y%m%d @ %H:%M:%S

NOTE: Unless you don't like the default format, you normally don't specify **fmtstr**

Dump of code.datetime_stamp:

datetime_stamp=
type=exec
src=from time import strftime;print(strftime("{{self.fmtstr}}"))
fmtstr=%Y%m%d @ %H:%M:%S
_code=*<code object>*
_last=20200923 @ 07:18:09
params={}

Macro: code.dump

dump(ns="namespace", name="var name regex" format="False" whitespace="False" help="False")

ns - the namespace that **name** is in
name - the variable/attribute name regex to dump
format - use formatting to aid readability

whitespace - include prefix whitespace (HTML entities) and breaks

help - dump the help attribute value (if present)

Dumps one or more variables that match **name**, formatting according to **format** and **whitespace**.

NOTE: The default values work best when running **smd** interactively.

Dump of code.dump:

dump=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.dump('{{self.ns}}', '{{self.name}}', {{self.format}}, {{self.whitespace}}, {{self.help}})
ns=code
name=dump$
format=True
whitespace=True
help=False
_code=*<code object>*
_last=
params={}

NOTE: The values for **format**, **whitespace** and **help** above are what is being used for generating the user documentation, and are not necessarily the defaults. Refer to the docs above for the actual defaults.

Macro: code.ln_alias

*ln_alias(**nm=**"existing_link_name", **attr=**"_new_PRIVATE_attr_name", **lt=**"new link text")*

nm - The name of the existing @link variable you are adding an alias to
attr - The name of a new PRIVATE attribute that will be added
lt - The new link text you want to use for the hyperlink

This macro adds a new alias for an existing link. That is, alternate text for the same underlying anchor.

NOTE: Be sure you specify a PRIVATE attribute i.e. begins with _ (underscore). If you do not, you will create an infinite recursion during markdown because the open tag will reference the new attribute!

Dump of code.ln_alias:

ln_alias=
type=exec
src=from .utility import CodeHelpers;print('@set _ns="link" _id="{0}" {1}="{3}{0}.<{4}{2}{3}{0}.> {4}"'.format('{{self.nm}}', '{{self.attr}}', '{{self.lt}}', CodeHelpers.b(0), CodeHelpers.b(1)))

nm=linkname
attr=_attr_name
lt=NEW_LINK_TEXT_HERE
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;print('@set _ns="link" _id="{0}" {1}="{3}{0}.<{4}{2}{3}{0}.>{4}"'.format('{{self.nm}}', '{{self.attr}}', '{{self.lt}}', CodeHelpers.b(0), CodeHelpers.b(1)))
params={}

Macro: code.repeat

repeat(t="chars to repeat" c="# to repeat")

t = character or string to repeat
c = number of times to repeat

Example: code.repeat(t="%" c="42") will print 42 "%" percent characters

Dump of code.repeat:

repeat=
type=eval
src=print('{}'.format('{{self.t}}'*{{self.c}}))
t={{self._help}}
c=1
_code=*<code object>*
_last=----------
params={}

Macro: code.wrap_stack

wrap_stack(w="< | > | # | tag.< | or tag.>" [encode="True | False"])

w = what you want to get from the wrap stack.
< - Will print the opening wrap tag
> - Will print the closing wrap tag
- Will print the current wrap tag index
tag.< - Will print the opening wrap tags in text form
tag.> - Will print the closing wrap tags in text form

encode = True if you want the HTML escaped, False for raw (default)

See the documentation on @wrap for details on how to use this macro

Dump of code.wrap_stack:

wrap_stack=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.wrap_stack('{{self.w}}', {{self.encode}})
w=*
encode=False
_code=*<code object>*
_last=html.li
params={}

Pushing lines onto input stream

Macro: code.equals

equals(v1="ns.var.attr1", v2="ns.var.attr2" true="ns.var.attr3" false="ns.var.attr4")

v1 - attribute 1 for comparison
v2 - attribute 2 for comparison
true - attribute to push if **v1** == **v2**
false - attribute to push if **v1** != **v2**

Compare attribute **v1** to attribute **v2** and push line onto input stream based on results.

This macro does a case-sensitive string comparison of **v1** and **v2**

Dump of code.equals:

equals=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.equals('{{self.v1}}', '{{self.v2}}', '{{self.true}}', '{{self.false}}')
v1=_ns.var.attr_
v2=_ns.txtvar.attr_
true=_ns.var.true_
false=_ns.var.false_
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.equals('{{self.v1}}', '{{self.v2}}', '{{self.true}}', '{{self.false}}')
params={}

Macro: code.pushline

pushline(t="line to push onto input stream")

t = string to push onto input stream

NOTE: This macro does NOT split the string on the newline \n character. It will give the illusion of doing so, but the newline will be treated as white space by a browser. To push multiple lines, use either code.pushlines or code.pushvar.

See also: code.pushlines, code.pushvar

Dump of code.pushline:

pushline=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.pushline('{{self.t}}')
t={{self._help}}
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.pushline('{{self.t}}')
params={}

Macro: code.pushlines

pushlines(t="lines to push onto input stream. separate lines with \n")

t = string to push onto input stream

NOTE: You can push multiple lines by inserting \n characters into the var/attr

See also: code.pushline, code.pushvar

Dump of code.pushlines:

pushlines=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.pushlines('{{self.t}}')
t={{self._help}}
_code=*<code object>*
_last=
params={}

Macro: code.pushlist

pushlist(attrlist="var.attr.list")

attrlist - an existing variable or attribute
If **attrlist** is a variable, then that variable *must* contain an attribute named **attrlist**.
If **attrlist** is an attribute, then that attribute *must* contain a list of other attributes in the same variable.

This macro pushes one or more lines specified by **attrlist** onto the input stream. The attributes must be in the same variable.

NOTE: See **var.vpl** for a template of what the **attrlist** variable should look like.

Dump of code.pushlist:

pushlist=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.pushlist('{{self.attrlist}}')
attrlist=var.macrohelp
_code=*<code object>*
_last=
params={}
nsvar=var
nsname=var1$|var2$|extvar$
title=Current definition of 'var1/2' and 'extvar' variables
name=esc_angles

Macro: code.pushvar

pushvar(t="variable whose value will be pushed onto input stream. separate lines with \n")

v = existing variable / attribute to push onto input stream

NOTE: You can push multiple lines by inserting \n characters into the var/attr
e.g. if var.x="line1\nline2" then pushvar(v="var.x") inserts two lines:

line1
line2

See also: code.pushline, code.pushlines

Dump of code.pushvar:

pushvar=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.pushvar('{{self.v}}')
v=self._help
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.pushvar('{{self.v}}')
params={}

Macro: code.replace

replace(var="search_str", val="attr_with_rep_val" str="ns.var.attr to operate on")

var - the string we are searching for
val - attribute containing replacement string
str - attribute containing string to operate on

Replaces all occurrences of **var** with **val** in **str** and then push **str** onto input stream.
NOTE: Does NOT modify the attribute **str** in memory; simply pushes modified string onto input stream.

Dump of code.replace:

replace=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.replace('{{self.var}}', '{{self.val}}', '{{self.str}}')

var=varname_to_replace
val=value_to_insert
str=string to operate on
_code=*<code object>*
_last=
params={}

Specialized get & set variable / attribute values

Macro: code.append

append(attr1="ns.var.attr", attr2="ns.var.attr with text to append")

attr1 - attribute to append text string to
attr2 - attribute containing the text to append to **attr1**

Appends text from **attr2** to existing attribute **attr1**.

Dump of code.append:

append=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.append('{{self.attr1}}', '{{self.attr2}}')
attr1=ns.var.attr
attr2=ns.txtvar.attr
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.append('{{self.attr1}}', '{{self.attr2}}')
params={}

Macro: code.attr_replace

attr_replace(s_str="search_str", r_var="attr_with_rep_val" attr="ns.var.attr to operate on" repl_nl="True | False")

s_str - the string we are searching for
r_var - attribute containing replacement string
attr - attribute containing string to operate on
repl_nl - replace escaped newline with newline (default:True)

Replaces all occurrences of **s_str** with **r_var** in **attr** and updates **attr**.
NOTE: Like **code.replace** except modifies **attr** directly and does NOT push anything onto the input stream.

Dump of code.attr_replace:

attr_replace=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.attr_replace('{{self.s_str}}', '{{self.r_var}}', '{{self.attr}}', {{self.repl_nl}})
s_str=string_to_replace
r_var=variable with new_value
attr=ns.var.attr
repl_nl=True
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.attr_replace('{{self.s_str}}', '{{self.r_var}}', '{{self.attr}}', {{self.repl_nl}})
params={}

Macro: code.attr_replace_str

attr_replace_str(s_str="search_str", r_str="repl_str" attr="ns.var.attr to operate on" repl_nl="True | False")

s_str - the string we are searching for
r_str - the replacement string
attr - attribute containing string to operate on
repl_nl - replace escaped newline with newline (default:True)

Replaces all occurrences of **s_str** with **r_str** in **attr** and updates **attr**.
NOTE: Like **code.attr_replace**, but accepts replacement string directly instead of indirectly via attribute.

Dump of code.attr_replace_str:

attr_replace_str=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.attr_replace_str('{{self.s_str}}', '{{self.r_str}}', '{{self.attr}}', {{self.repl_nl}})
s_str=string_to_replace
r_str=new_value
attr=ns.var.attr
repl_nl=True
_code=*<code object>*
_last=
params={}

Macro: code.attr_replace_value

attr_replace_value(attr="ns.var.attr to operate on" value="new value" repl_nl="True | False")

attr - attribute whose value we are to replace
value - replacement value
repl_nl - replace escaped newline with newline (default:True)

Replaces **attr** with **value**.

Dump of code.attr_replace_value:

attr_replace_value=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.attr_replace_value('{{self.attr}}', '{{self.value}}', {{self.repl_nl}})
attr=ns.var.attr
value=new_value
repl_nl=True
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.attr_replace_value('{{self.attr}}', '{{self.value}}', {{self.repl_nl}})
params={}

Macro: code.get

get(v="variable_name")

v - variable / attribute name to get

Emits the value of the variable / attribute **v**

Dump of code.get:

get=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.get_ns_var('{{self.v}}')
v=variable_name
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.get_ns_var('{{self.v}}')
params={}

Macro: code.get_value

get_value(v="variable_name" ret_type="0|1|2|3|9" escape="True|False" esc_smd="True|False")

v - variable / attribute name to get

ret_type - how to emit the value (0, 1, 2, 3 or 9)
0 - will return the value raw i.e. not marked down
1 - will return the value with the initial markdown pass, but without handling delayed expansion
2 - will return the after replacing [with [and] with]
3 - will return the after replacing **code.get_value.** with **ns.varname.**
9 - will return the marked down value (default)

escape - True to escape the HTML, False otherwise (default)

esc_smd - True to escape the SMD, False otherwise (default)

Emits the value of the variable / attribute **v**

Dump of code.get_value:

get_value=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.get_ns_var('$.v', $.ret_type, $.escape, $.esc_smd)
v=variable_name
ret_type=9
escape=False
esc_smd=False
_code=*<code object>*
last= {{self.}}(url="<text to escape>")

    url = string to operate on

    Replace all occurrences of < with < and > with > in string url
params={}

Macro: code.get_default

get_default(v="variable_name", default="default value")

v - The variable / attribute name to get
default - value to return if **v** is not defined

Returns the value of **v** unless undefined, in which case it returns **default**

Dump of code.get_default:

get_default=
type=exec
src=from .utility import CodeHelpers;CodeHelpers.default('$.v', '$.default')
v=var or attr name
default=default value
_code=*<code object>*
_last=raw src=from .utility import CodeHelpers;CodeHelpers.default('$.v', '$.default')
params={}

Contents of the code.md builtins file

The import file **[sys.imports]/code.md** is where the majority of the **@code** built-in macros are documented, so we will just embed the file here to review them.

Contents of code.md builtins

```
// @code namespace builtins

@code _id="esc_angles"\
      type="eval"\
      src="print('{{self.url}}'.replace('<', '&lt;').replace('>','&gt;'))"\
      url="{{self._help}}"\
      _help="[sp.2]*{{self._}}(url=\"[E.lt]text to escape[E.gt]\")*[bb]\
[sp.4]**url** = string to operate on[bb]\
[sp.4]Replace all occurrences of *[E.lt]* with *[E.amp]lt;* and *[E.gt]* with *
[E.amp]gt;* in string **url**"

@code _id="escape"\
      type="exec"\
      src="from .utility import HtmlUtils;print(HtmlUtils.escape_html('{{self.t}}'))"\
      t="{{self._help}}"\
      _help="[sp.2]*{{self._}}(t=\"text to escape\")*[bb]\
[sp.4]**t** = string to operate on[bb]\
[sp.4]Same as code.esc_angles, but also replaces *[E.amp]* with *[E.amp]amp;*"

@code _id="escape_var"\
      type="exec"\
      src="from .utility import
HtmlUtils;print(HtmlUtils.escape_html_var('{{self.v}}'))"\
      v="self._help"\
      _help="[sp.2]*{{self._}}(v=\"var_to_esc\")*[bb]\
[sp.4]**v** = existing variable / attribute to escape[bb]\
[sp.4]Same as code.escape, but takes an existing variable or attribute name to operate
on."

@code _id="encodeURL"\
      type="exec"\
      src="from .utility import HtmlUtils;print(HtmlUtils.encodeURL('{{self.t}}'))"\
      t="{{self._help}}"\
      _help="[sp.2]*{{self._}}(t=\"https://www.url.com?x=encode me\")*[bb]\
[sp.4]**t** = a URL string to be encoded[bb]\
[sp.4]Replaces all blanks in URL string e.g. \" \" with \"*%20*\"[bb]\
[sp.4]**NOTE:** If **t** begins with **mailto:** the entire string is encoded using a
mix of dec/hex HTML character entities"

@code _id="encode_smd"\
      type="exec"\
      src="from .utility import HtmlUtils;print(HtmlUtils.encode_smd('{{self.t}}'))"\
      t="{{self._help}}"\
      _help="[sp.2]*{{self._}}(t=\"smd markdown to encode\")*[bb]\
[sp.4]**t** = an smd string that will be encoded for display[bb]\
[sp.4]Use *[E.lt][sp][E.gt]* for *[E.lb][sp][E.rb]*, *2{* for *[E.lcb2]*, *2}* for *
[E.rcb2]*, *2[E.plus]* for *[E.ins]* and *2[E.tilde]* for *[E.del]*.[bb]\
[sp.4]The entire list of replacements is:[b]\
[sp.6]*[E.lt]* becomes *[E.amp]lsqb;*[b]\
[sp.6]*[E.gt]* becomes *[E.amp]rsqb;*[b]\
[sp.6]*2[E.lcb]* becomes *[E.amp]lcub;[E.amp]lcub;*[b]\
[sp.6]*2[E.rcb]* becomes *[E.amp]rcub;[E.amp]rcub;*[b]\
[sp.6]*[E.ast]* becomes *[E.amp]#42;*[b]\
[sp.6]*[E.at]* becomes *[E.amp]#64;*[b]\
[sp.6]*2[E.plus]* becomes *[E.amp]plus;[E.amp]plus;*[b]\
[sp.6]*2[E.tilde]* becomes *[E.amp]tilde;[E.amp]tilde;*[bb]\
[sp.4]**NOTE1:** Do not use square brackets, curly braces, double plus or double tilde
directly, because the parser will replace them with markdown.[bb]\
[sp.4]**NOTE2:** You need to use *[E.lb]E.lp[E.rb]* for *(* and *[E.lb]E.rp[E.rb]* for
*)*, because parenthesis are used by the markdown to identify parameters to the
macros."

@code _id="encode_smd_var"\
```

```
      type="exec"\
      src="from .utility import
HtmlUtils;print(HtmlUtils.encode_smd_var('{{self.v}}'))"\
      v="self._help"\
      _help="[sp.2]*{{self._}}(v=\"var_to_encode\")*[bb]\
[sp.4]**v** = existing variable / attribute to markdown as smd[bb]\
[sp.4]Same as code.encode_smd, but takes an existing variable or attribute name to
operate on"

@code _id="datetime_stamp"\
      type="exec"\
      src="from time import strftime;print(strftime(\"{{self.fmtstr}}\"))"\
      fmtstr="%Y%m%d @ %H:%M:%S"\
      _help="[sp.2]*{{self._}}(fmtstr=\"%Y%m%d @ %H:%M:%S\")*[bb]\
[sp.4]**fmtstr** is a Python strftime format string. The default format is: %Y%m%d @
%H:%M:%S[bb]\
[sp.4]**NOTE:** Unless you don't like the default format, you normally don't specify
**fmtstr**"

@code _id="repeat"\
      type="eval"\
      src="print('{}'.format('{{self.t}}'*{{self.c}}))"\
      t="{{self._help}}"\
      c="1"\
      _help="[sp.2]*{{self._}}(t=\"chars to repeat\" c=\"# to repeat\")*[bb]\
[sp.4]**t** = character or string to repeat[b]\
[sp.4]**c** = number of times to repeat[bb]\
[sp.4]Example: code.repeat(t=\"%\" c=\"42\") will print 42 \"%\" percent characters"

@code _id="get"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.get_ns_var('{{self.v}}')"\
      v="variable_name"\
      _help="[sp.2]*{{self._}}(v=\"variable_name\")*[bb]\
[sp.4]**v** - variable / attribute name to get[bb]\
[sp.4]Emits the value of the variable / attribute **v**"

@code _id="get_value"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.get_ns_var('$.v', $.ret_type,
$.escape, $.esc_smd)"\
      v="variable_name"\
      ret_type="9"\
      escape="False"\
      esc_smd="False"\
      _help="[sp.2]*{{self._}}(v=\"variable_name\" ret_type=\"0|1|2|3|9\"
escape=\"True|False\" esc_smd=\"True|False\")*[bb]\
[sp.4]**v** - variable / attribute name to get[bb]\
[sp.4]**ret_type** - how to emit the value (0, 1, 2, 3 or 9)[b]\
[sp.6]**0** - will return the value raw i.e. not marked down[b]\
[sp.6]**1** - will return the value with the initial markdown pass, but without
handling delayed expansion[b]\
[sp.6]**2** - will return the after replacing {{ with [ and }} with ][b]\
[sp.6]**3** - will return the after replacing **self.** with **ns.varname.**[b]\
[sp.6]**9** - will return the marked down value (default)[bb]\
[sp.4]**escape** - True to escape the HTML, False otherwise (default)[bb]\
[sp.4]**esc_smd** - True to escape the SMD, False otherwise (default)[bb]\
[sp.4]Emits the value of the variable / attribute **v**"

@code _id="get_default"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.default('$.v', '$.default')"\
      v="var or attr name"\
      default="default value"\
```

```
      _help="[sp.2]*{{self._}}(v=\"variable_name\", default=\"default value\")*[bb]\
[sp.4]**v** - The variable / attribute name to get[b]\
[sp.4]**default** - value to return if **v** is not defined[bb]\
[sp.4]Returns the value of **v** unless undefined, in which case it returns
**default**"

@code _id="pushline"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.pushline('{{self.t}}')"\
      t="{{self._help}}"\
      _help="[sp.2]*{{self._}}(t=\"line to push onto input stream\")*[bb]\
[sp.4]t = string to push onto input stream[bb]\
[sp.4]**NOTE:** This macro does NOT split the string on the newline \n character. It
will give[b]\
[sp.4]the illusion of doing so, but the newline will be treated as white space by a[b]\
[sp.4]browser. To push multiple lines, use either code.pushlines or code.pushvar.[bb]\
[sp.4]See also: code.pushlines, code.pushvar"

@code _id="pushlines"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.pushlines('{{self.t}}')"\
      t="{{self._help}}"\
      _help="[sp.2]*{{self._}}(t=\"lines to push onto input stream. separate lines with
\n\")*[bb]\
[sp.4]t = string to push onto input stream[bb]\
[sp.4]**NOTE:** You can push multiple lines by inserting \n characters into the
var/attr[bb]\
[sp.4]See also: code.pushline, code.pushvar"

@code _id="pushvar"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.pushvar('{{self.v}}')"\
      v="self._help"\
      _help="[sp.2]*{{self._}}(t=\"variable whose value will be pushed onto input
stream. separate lines with \n\")*[bb]\
[sp.4]v = existing variable / attribute to push onto input stream[bb]\
[sp.4]**NOTE:** You can push multiple lines by inserting \n characters into the
var/attr[b]\
[sp.4]e.g. if var.x=\"line1\nline2\" then pushvar(v=\"var.x\") inserts two lines:[bb]\
[sp.6]line1[b]\
[sp.6]line2[bb]\
[sp.4]See also: code.pushline, code.pushlines"

@code _id="ln_alias"\
      type="exec"\
      src="from .utility import CodeHelpers;print('@set _ns=\"link\" _id=\"{0}\" \
      {1}=\"{3}{0}.<{4}{2}{3}{0}.>{4}\"'.format('{{self.nm}}', \
      '{{self.attr}}', '{{self.lt}}', CodeHelpers.b(0), CodeHelpers.b(1)))"\
      nm="linkname" attr="_attr_name" lt="NEW_LINK_TEXT_HERE"\
      _help="[sp.2]*{{self._}}(**nm=**\"existing_link_name\",
**attr=**\"_new_PRIVATE_attr_name\", **lt=**\"new link text\")*[bb]\
[sp.4]**nm** - The name of the existing @link variable you are adding an alias to[b]\
[sp.4]**attr** - The name of a new PRIVATE attribute that will be added[b]\
[sp.4]**lt** - The new link text you want to use for the hyperlink[bb]\
[sp.4]This macro adds a new alias for an existing link. That is, alternate text for the
same underlying anchor.[bb]\
[sp.4]**NOTE:** Be sure you specify a PRIVATE attribute i.e. begins with ***_***
(underscore). If you do not, you[b]\
[sp.4]will create an infinite recursion during markdown because the open tag will
reference the new attribute!"

@code _id="append"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.append('{{self.attr1}}',
```

```
'{{self.attr2}}')"\
      attr1="ns.var.attr"\
      attr2="ns.txtvar.attr"\
      _help="[sp.2]*{{self._}}(attr1=\"ns.var.attr\", attr2=\"ns.var.attr with text to
append\")*[bb]\
[sp.4]**attr1** - attribute to append text string to[b]\
[sp.4]**attr2** - attribute containing the text to append to **attr1**[bb]\
[sp.4]Appends text from **attr2** to existing attribute **attr1**."

@code _id="equals"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.equals('{{self.v1}}',
'{{self.v2}}', '{{self.true}}', '{{self.false}}')"\
      v1="_ns.var.attr_"\
      v2="_ns.txtvar.attr_"\
      true="_ns.var.true_"\
      false="_ns.var.false_"\
      _help="[sp.2]*{{self._}}(v1=\"ns.var.attr1\", v2=\"ns.var.attr2\"
true=\"ns.var.attr3\" false=\"ns.var.attr4\")[bb]*\
[sp.4]**v1** - attribute 1 for comparison[b]\
[sp.4]**v2** - attribute 2 for comparison[b]\
[sp.4]**true** - attribute to push if **v1** == **v2**[b]\
[sp.4]**false** - attribute to push if **v1** != **v2**[bb]\
[sp.4]Compare attribute **v1** to attribute **v2** and push line onto input stream
based on results.[bb]\
[sp.4]This macro does a case-sensitive string comparison of **v1** and **v2**"

@code _id="wrap_stack"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.wrap_stack('{{self.w}}',
{{self.encode}})"\
      w="*"\
      encode="False"\
      _help="[sp.2]*{{self._}}(w=\"< | > | # | tag.< | or tag.>\" [encode=\"True |
False\"])*[bb]\
[sp.4]**w** = what you want to get from the wrap stack.[b]\
[sp.6][E.lt] - Will print the opening wrap tag[b]\
[sp.6][E.gt] - Will print the closing wrap tag[b]\
[sp.6]# - Will print the current wrap tag index[b]\
[sp.6]tag.[E.lt] - Will print the opening wrap tags in text form[b]\
[sp.6]tag.[E.gt] - Will print the closing wrap tags in text form[bb]\
[sp.4]**encode** = True if you want the HTML escaped, False for raw (default)[bb]\
[sp.4]See the documentation on @wrap for details on how to use this macro"

@code _id="replace"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.replace('{{self.var}}',
'{{self.val}}', '{{self.str}}')"\
      var="varname_to_replace"\
      val="value_to_insert"\
      str="string to operate on"\
      _help="[sp.2]*{{self._}}(var=\"search_str\", val=\"attr_with_rep_val\"
str=\"ns.var.attr to operate on\")*[bb]\
[sp.4]**var** - the string we are searching for[b]\
[sp.4]**val** - attribute containing replacement string[b]\
[sp.4]**str** - attribute containing string to operate on[bb]\
[sp.4]Replaces all occurrences of **var** with **val** in **str** and then push **str**
onto input stream.[b]\
[sp.4]**NOTE:** Does NOT modify the attribute **str** in memory; simply pushes modified
string onto input stream."

@code _id="attr_replace"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.attr_replace('{{self.s_str}}',
```

```
'{{self.r_var}}', '{{self.attr}}', {{self.repl_nl}})"\
      s_str="string_to_replace"\
      r_var="variable with new_value"\
      attr="ns.var.attr"\
      repl_nl="True"\
      _help="[sp.2]*{{self._}}(s_str=\"search_str\", r_var=\"attr_with_rep_val\"
attr=\"ns.var.attr to operate on\" repl_nl=\"True | False\")*[bb]\
[sp.4]**s_str** - the string we are searching for[b]\
[sp.4]**r_var** - attribute containing replacement string[b]\
[sp.4]**attr** - attribute containing string to operate on[b]\
[sp.4]**repl_nl** - replace escaped newline with newline (default:True)[bb]\
[sp.4]Replaces all occurrences of **s_str** with **r_var** in **attr** and updates
**attr**.[b]\
[sp.4]**NOTE:** Like **code.replace** except modifies **attr** directly and does NOT
push anything onto the input stream."

@code _id="attr_replace_value"\
      type="exec"\
      src="from .utility import
CodeHelpers;CodeHelpers.attr_replace_value('{{self.attr}}', '{{self.value}}',
{{self.repl_nl}})"\
      attr="ns.var.attr"\
      value="new_value"\
      repl_nl="True"\
      _help="[sp.2]*{{self._}}(attr=\"ns.var.attr to operate on\" value=\"new value\"
repl_nl=\"True | False\")*[bb]\
[sp.4]**attr** - attribute whose value we are to replace[b]\
[sp.4]**value** - replacement value[b]\
[sp.4]**repl_nl** - replace escaped newline with newline (default:True)[bb]\
[sp.4]Replaces **attr** with **value**."

@code _id="attr_replace_str"\
      type="exec"\
      src="from .utility import
CodeHelpers;CodeHelpers.attr_replace_str('{{self.s_str}}', '{{self.r_str}}',
'{{self.attr}}', {{self.repl_nl}})"\
      s_str="string_to_replace"\
      r_str="new_value"\
      attr="ns.var.attr"\
      repl_nl="True"\
      _help="[sp.2]*{{self._}}(s_str=\"search_str\", r_str=\"repl_str\"
attr=\"ns.var.attr to operate on\" repl_nl=\"True | False\")*[bb]\
[sp.4]**s_str** - the string we are searching for[b]\
[sp.4]**r_str** - the replacement string[b]\
[sp.4]**attr** - attribute containing string to operate on[b]\
[sp.4]**repl_nl** - replace escaped newline with newline (default:True)[bb]\
[sp.4]Replaces all occurrences of **s_str** with **r_str** in **attr** and updates
**attr**.[b]\
[sp.4]**NOTE:** Like **code.attr_replace**, but accepts replacement string directly
instead of indirectly via attribute."

@code _id="dump"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.dump('{{self.ns}}',
'{{self.name}}', {{self.format}}, {{self.whitespace}}, {{self.help}})"\
      ns="var"\
      name=".*"\
      format="False"\
      whitespace="False"\
      help="False"\
      _help="[sp.2]*{{self._}}(ns=\"namespace\", name=\"var name regex\"
format=\"False\" whitespace=\"False\" help=\"False\")*[bb]\
[sp.4]**ns** - the namespace that **name** is in[b]\
[sp.4]**name** - the variable/attribute name regex to dump[b]\
```

```
[sp.4]**format** - use formatting to aid readability[bb]\
[sp.4]**whitespace** - include prefix whitespace (HTML entities) and breaks [E.lt]br
/[E.gt][bb]\
[sp.4]**help** - dump the help attribute value (if present)[bb]\
[sp.4]Dumps one or more variables that match **name**, formatting according to
**format** and **whitespace**.[bb]\
[sp.4]**NOTE:** The default values work best when running **smd** interactively."

@var _id = "vpl"\
      attrlist="1,2,3,4,5"\
      1="Line 1" 2="Line 2" 3="Line 3" 4="Line 4" 5="Line 5"

@code _id="pushlist"\
      type="exec"\
      src="from .utility import CodeHelpers;CodeHelpers.pushlist('{{self.attrlist}}')"\
      attrlist="var.vpl.attrlist"\
      _help="[sp.2]*{{self._}}(attrlist=\"var.attr.list\")*[bb]\
[sp.4]**attrlist** - an existing variable or attribute[b]\
[sp.6]If **attrlist** is a variable, then that variable *must* contain an attribute
named **attrlist**.[b]\
[sp.6]If **attrlist** is an attribute, then that attribute *must* contain a list of
other attributes in the same variable.[bb]\
[sp.4]This macro pushes one or more lines specified by **attrlist** onto the input
stream. The attributes must be in the same variable.[bb]\
[sp.4]**NOTE:** See **var.vpl** for a template of what the **attrlist** variable should
look like."
```

A few more things about @code

Here are just a few more things about the **@code** namespace to help drive home your understanding of the declaration and usage of variables within it.

@code attributes cannot be permanently changed via _null_ or when markdown is applied. You must use **@set** to change a **@code** attribute value.

Not specific to **@code**, but you'll likely encounter it here: You cannot use parenthesis **( )** inside parameter strings, as it breaks the parser... You can workaround this by using the **[E.lp]/[E.rp]** constants.

The restriction of using parenthesis inside parameter strings leads to this issue/shortcoming: you cannot nest markdown variables that require parameters, since that would require that you use () inside parameter strings. Unfortunately, there is no direct workaround for this right now.

Here's an example:

[encode_smd(t="<var.revision.plain*(*v=\"1.4.2\"*)*>")] - emits **[encode_smd(t="")]**

[encode_smd(t="<var.revision.plain*[E.lp]*v=\"1.4.2\"*[E.rp]*>")] - emits **[var.revision.plain(v="1.4.2")]**

Providing a solution to this shortcoming is on the wishlist for a future version of the parser, but there is no ETA at this time.

Table of Contents

Creating Sophisticated Layouts

Now that you've got the basics out of the way, it's time to dig in a little deeper and look at the tools available to create bigger and more sophisticated layouts for your documents. In the chapters that follow, we will look at importing files in order to create reusable building blocks, managing the default block tags emitted when no specific HTML markdown is specified, and a set of common **div**'s that you can use and/or customize to your specific needs.

Table of Contents

Importing, Embedding and Watching files

In this chapter, we will cover three different keywords used in **smd**:

@import - Used to import markdown files
@embed - Used to embed files directly into the output stream
@watch - Used to manually add a file to the watch list

The @import keyword

The most often used of these keywords is **@import**. It begins during **smd** startup, when it is used to import the builtins, and continues throughout processing depending on the complexity of your markdown documents.

A few things that are import to keep in mind when using **@import**:

- Any given file will only be **@import**'ed one time. Subsequent attempts are ignored.
- These files can and usually do contain markdown
- They can be nested to any level
- **@wrap** tags used within any imported file are automatically cleared when EOF is reached on the imported file.
- If a path begins with '$' or '$/', they are replaced with the path of the currently opened file; this allows relative importing of dependencies.

Here are some examples of **@import**:

Example @import statements

```
@import "filename"
@import "/abs/path/to/filename"
@import "../relative/path/to/filename"
@import "$/relative/to/current/filename"
```

The **@import** statement can be used to include documents that contain commonly used contents for your scripts, such as common aliases, links, headers, sections, etc. You can use fully qualified paths, relative paths, or paths based on the current file being processed. The latter uses the symbol '$' to designate that the path to this file should begin with the path the current file being processed. For example, assume the current file being processed is: /mydir/myfile.md. The statement:

```
@import "$/vars.md"
```

Would be the same as writing **@import "/mydir/vars.md"**. This is useful because it doesn't require that you use absolute paths for everything. The path can be specified as either **'$/path/filename'** or **'$path/filename'**. In other words, you can specify the '/' after '$' or leave it off, whatever your preference is.

The @embed keyword

The **@embed** keyword is used to embed files directly into the output stream. It's a bit like **@import**, except that the files are **not** parsed. The contents are embedded directly into the output stream. In most cases, you will use **@embed** to embed HTML markdown directly into your output stream, but technically, it can be used to embed anything. Just make sure that whatever you embed is properly formatted for the context, as there isn't any way to modify it. Here are few fun facts about **@embed**:

- Any given file can be **@embed**'ed as many times as you want.
- These files usually do **not** contain **smd** markdown, although it's okay if they do, the content just won't be parsed...
- Like **@import**, **@embed** files can be nested to any level
- Also like **@import**, **@embed** supports the relative import prefix of '$' or '$/'

@embed Syntax

@embed "/path/filename" [<escape | esc> = "True" | "Yes"]

The @embed keyword supports an option **escape** or **esc** that allows you to specify that the output should be encoded HTML:

< becomes < > becomes > ...

This allows you to display HTML encoded files as text, otherwise the browser would interpret and render it as HTML. The default is **False**.

Example @embed statements

```
@embed "myembedfile.html"
@embed "myembedfile.html" escape="true"
```

The @watch keyword

@watch isn't used very often, since most files that are either imported, embedded or otherwise handled via one of the command line interfaces are automatically added to the watch list.

What is the watch list?

The watch list is simply a list of all the files that the **smd** parser has encountered while performing markdown on your content. The higher level command line utilities such as **smdparse** and **ismd** also add files to the watch list, but for the most part, it's the underlying **smd** parser that is responsible for keeping track.

Interestingly enough, as far as **smd** is concerned, the watch list isn't very important. It doesn't act on it in any way, but it's useful for other programs and utilities, such as **ismd**, as a way to detect changes occurring to files that are directly involved with the current document being processed, and act on it. In the case of **ismd**, for example, when changes to the files that make up a document are detected, it will invoke the parser again to refresh the content, and then notify all the output monitors that they need to update their output windows. This is extremely useful while developing new content, because it gives you a sort of WYSIWYG experience, always updating the output view as you make changes!

So why do we need **@watch** then? I mean, if **smd** and other command line utilities already track everything, why do I need it? Good question, grasshopper. The most common reason ties back to the **@embed** command. Since the content of embedded files is not parsed or scanned in any way, there isn't a way for the parser to know if that content references something that needs to be monitored. Enter **@watch**. Using this keyword, you can add files to the watch list, and then if they are changed, and one of the underlying utilities that monitors the watch list detects it, it will reparse the content and update the output monitors. Pretty cool, huh?

The usage is simple, identical to **@embed** as a matter of fact (except that the **escape** option isn't supported -- nor does it make sense here):

Example @watch statement

```
@watch "mywatchfile.html"
```

You can use the **@dump** keyword with option **tracked=** to review the files that are currently being watched. This is a good way to just make sure that your watch command is working as expected during the creation process. For example, let's say you **@embed** the file "myscript.html" which includes a **<script>** tag that references **myjavascript.js**. We could make sure that our **@watch** markdown is correct by using the following markdown:

```
Debug hack for making sure my @watch file is monitored

// first let's dump the embed file inline escaped
@embed '$/../import/myscript.html' escape="true"
<script src="myjavascript.js"></script>

[smdcomment.il] now embed it again, but this time don't escape it
@embed '$/../import/myscript.html'

// now let's see which files are being watched
@dump tracked=".*(myjavascript.js|myscript.html)"
  ----------------------------------------
  Files seen during parsing: .*(myjavascript.js|myscript.html)
  ----------------------------------------
    /Users/ken/Dropbox/shared/src/script/smd/docs/import/myscript.html

// okay, so only the myscript.html was picked up, as expected. Let's add a watch for the JS file...
@watch '$/../import/myjavascipt.js'

// Let's check again to see what's being monitored...
@dump tracked=".*(myjavascript.js|myscript.html)"
  ----------------------------------------
  Files seen during parsing: .*(myjavascript.js|myscript.html)
  ----------------------------------------
    /Users/ken/Dropbox/shared/src/script/smd/docs/import/myscript.html
    /Users/ken/Dropbox/shared/src/script/smd/docs/import/myjavascript.js

// Cool! Now both files are being watched ...
```

When you review the preceding section in the docs, you will notice a lot more than is actually needed in practice. This is because I wanted to show the steps and the actual content to help you understand all the steps. In reality, all you needed was:

Just the needed steps for the debug hack...

```
@embed 'path/to/myscript.html'
@watch 'path/to/myjavascript.js'
@dump tracked=".*(myjavascript.js|myscript.html)"
```

Summary of @import, @embed and @watch

That wraps up the section on importing, embedding and watching files. Onward!

Table of Contents

@wrap and @parw

By default, **smd** does not place any type of wrapper around emitted contE. This is a good default, however, many times adding wrappers can greatly assist your formatting efforts. This is where @wrap comes in handy...

First, let's review the complete syntax for both **@wrap** and **@parw**

> **@wrap and @parw command syntax**
>
> **@wrap <html.tag [, html.tag [, ...]] | [nop | null]>**
>
> **@parw [* | all | #]**
>
> ***NOTE: tag variables passed to @wrap must be in the @html namespace.***

The concept for **@wrap** is straightforward: whenever anything is written to the output stream, it will be wrapped with the specified variable from the **@html** namespace. Let's look at an example, **smd** session:

@wrap demonstration

```
// By default, there are no wrap tags in effect
This line will not have any tags
Output -->This line will not have any tags
@wrap p
This line will be wrapped with HTML p tags
Output --><p>This line will be wrapped with HTML p tags</p>
This line also...
Output --><p>This line also...</p>
@parw
But not this line
Output -->But not this line
```

See how that works? When we issue the **@wrap p** command, subsequent output is wrapped with tags. It remains in effect until it is cleared with the **@parw** command, or if inside an **@import**'ed file, when EOF is reached on the current file.

> It's important to note that any **@wrap** tags that are in effect during the processing of an imported file will be cleared when EOF is reached on that imported file, so that when you return to the parent file, the **@wrap** tag stack for it will be restored to what it was before the file was **@import**'ed.

@wrap tags can be nested as well. Consider the following example:

@wrap tag nesting

```
@wrap div
This line will be wrapped with HTML div tags
Output --><div>This line will be wrapped with HTML div tags</div>
@wrap p
This line, however, will be wrapped in HTML p tags
Output --><p>This line, however, will be wrapped in HTML p tags</p>
@parw
And now we are back to being wrapped in HTML div tags again
Output --><div>And now we are back to being wrapped in HTML div tags again</div>
```

You can nest as deeply as you wish.

In addition, you are not limited to a single HTML tag when wrapping. Take a look at this example:

@wrap tag multiples

```
@wrap div, p
This line will be wrapped with HTML both a div and a p tag
Output -->&lt;div&gt;&lt;p&gt;This line will be wrapped with HTML both a div and a p tag&lt;/p&gt;&lt;/div&gt;
```

It is important to distinguish that if multiple tags are listed on a single **@wrap** tag, the parser considers this a single wrap tag with multiple elements. That is, when you use **@parw**, it will clear a single item from the stack, and in the case of a multiple **@wrap** tag, it clears all of the tags associated with that tag. Let's demonstrate it just to be sure:

Clearing @wrap tag multiples

```
Begin by showing that input is not wrapped by default
Output -->Begin by showing that input is not wrapped by default
@wrap div, p
This line will be wrapped with HTML both a div and a p tag
Output -->&lt;div&gt;&lt;p&gt;This line will be wrapped with HTML both a div and a p tag&lt;/p&gt;&lt;/div&gt;
@parw
And now this line has no tags
Output -->And now this line has no tags
```

If you issue **@parw** when the wrap stack is empty, the parser issues a warning:

@parw with empty wrap stack

```
@parw
WARNING: wrapper stack is empty<br />
```

Additional options

The **@wrap** has two specialized parameters **nop** and **null** which do the same thing: they temporarily suspend wrapping of content with the previously specified **@wrap** tag. Here's an example:

Temporarily suspend wrapping of content

```
@wrap div, p
This line will be wrapped with HTML both a div and a p tag
Output -->&lt;div&gt;&lt;p&gt;This line will be wrapped with HTML both a div and a p tag&lt;/p&gt;&lt;/div&gt;
@wrap null
And now this line has no tags
Output -->And now this line has no tags
@parw
And now this line will have the previously specified wrap tags of div, p
Output -->&lt;div&gt;&lt;p&gt;And now this line will have the previously specified wrap tags of
div, p&lt;/p&gt;&lt;/div&gt;
```

The **@parw** also accepts parameters: *** | all** or alternatively a **#** specified how many tags to clear from the stack. By default, **@parw** will clear a single item from the wrap stack. If you pass an integer **n**, then it will attempt to clear **n** items from the wrap stack. If you pass either ***** or **all**, then it will clear all items from the wrap stack, but only those that were added within the current scope. The current scope just means the current **@import** file, and it goes back to the note we made up front about remove items from the wrap stack. Here are a couple of examples of the parameters to **@parw**:

Clearing the wrap stack

```
@wrap div, p
This line will be wrapped with HTML both a div and a p tag
Output -->&lt;div&gt;&lt;p&gt;This line will be wrapped with HTML both a div and a p tag&lt;/p&gt;&lt;/div&gt;
@wrap li
And now this line a list item tag
Output -->&lt;li&gt;And now this line a list item tag&lt;/li&gt;
@parw *
And now this line has no tags
Output -->And now this line has no tags

@wrap div
This line will be wrapped with HTML div tags
Output -->&lt;div&gt;This line will be wrapped with HTML div tags&lt;/div&gt;
@wrap p
This line, however, will be wrapped in HTML p tags
Output -->&lt;p&gt;This line, however, will be wrapped in HTML p tags&lt;/p&gt;
@parw 25
WARNING: only 2 items found on the stack that were cleared<br />
```

The final thing we'll cover is a specialized **@code** macro called **wrap_stack**. This macro gives you some additional options to use with **@wrap** tags for a few specialized cases. Here are the details on this macro.

The first thing is the macro itself, here's the builtin help string for it:

code.wrap_stack help string

wrap_stack(w="< | > | # | tag.< | or tag.>" [encode="True | False"])

w = what you want to get from the wrap stack.
< - Will print the opening wrap tag
> - Will print the closing wrap tag
 # - Will print the current wrap tag index
tag.< - Will print the opening wrap tags in text form
tag.> - Will print the closing wrap tags in text form

encode = True if you want the HTML escaped, False for raw (default)

See the documentation on @wrap for details on how to use this macro

If you invoke it with no parameters *[code.wrap_stack]*, it will return the complete starting and ending tags for the current wrap tag.

If you pass <, it returns the opening tag(s) sequence, and conversely, > returns the closing tag(s).

If you pass #, it returns the current stack size, which is really only useful for debugging purposes.

Finally, are the options tag.< and tag.>. These are similar to the < > options, but they return the tags in text form instead of as the actual HTML markup. tag.< is the most useful, because you can use it to maintain the current **@wrap** tags, while adding one or more additional tags.

Here are some examples:

Viewing the wrap stack

```
@wrap div, p
[code.wrap_stack]
<div><p><div><p></p></div></p></div>

@@[code.wrap_stack]
<div><p></p></div>

@@[code.wrap_stack(w="<")]
<div><p>

@@[code.wrap_stack(w=">")]
</p></div>

@@[code.wrap_stack(w="#")]
1

@@[code.wrap_stack(w="tag.<")]
div,p
```

At first glance, the initial call to **[code.wrap_stack]** might look like a bug, because the tag appears twice. However, this is behaving properly, since technically, a wrap tag of **div,p** is currently in effect, and thus the output from the call to **code.wrap_stack** is being wrapped in the tags! On the subsequent calls, you'll notice we prefix each line with **@@**, something we haven't discussed yet. This is the **raw** prefix, and when used at the start of any line, any wrap tags in effect are ignored.

This concludes the chapter on **@wrap** and **@parw**.

Table of Contents

Divs

In this chapter, we will discuss the predefined **DIV**'s that are declared inside the builtin file **divs.md**.

> NOTE: If you have not read the chapters on **@var**, **@set** and **@html**, stop right now and go read them. Otherwise, you might have trouble understanding the concepts that will be covered in this chapter...

The predefined DIVs in the **divs.md** builtin are organized into four types to provide several options for your content. Let's start with a summary of the divs that are available:

Generic DIVs
[section] - Generic content
[section_pbb] - Generic content but with the class **pbb** (i.e. page break before -- when printing)
[toc] - Table of Contents
[syntax] - Syntax content
[review] - Review content
[review_pba] - Review content with **pba** class (i.e. page break after -- when printing)
[plain] - Plain content with class plainTitle (draws a bottom-border) after the title

Source DIVs
[code] - Source content - See notes for rendering details

Note DIVs
[note] - Note content - See notes for rendering details on this and remaining DIVs
[box] - Box content
[generic] - Generic content
[greyout] - Greyout content
[important] - Important content
[question] - Question content
[vo] - Voiceover content

Miscellaneous DIVs
[extras] - Wrap content in class="extras" DIV
[divxp] - Like **extras**, but also wraps content inside **<p>** tag
[terminal] - Terminal content
[terminal2] - Variation of **terminal** content

Lists
[ulist] - Unordered bulleted list content
[ulistplain] - Unordered list no bullets content
[olist] - Ordered list with decimal numbers content

Generic DIVs

There are four different styles of DIVs that are predefined for you, and you can add more as well as customize these to your hearts content. Each of these has a similar interface, so let's see what that is, and how it is used. Only one of each different type will be covered, since the interface on the others is identical! We will start with the ***section**-style* DIVs, of which you have **section, section_pbb, toc, syntax, review, review_pba and plain**.

First, let's take a look at the actual definition of **section** and it's associated **@html** and **@var** variables:

@html Support for *section*

_section_div_=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=div
class=section

_section_p_=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=p
class=divTitle

_section_p_content_=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=p

@var definition for *section*

section=
_format=@@ {{self.inline}}
with_content=@@ {{self.wc_inline}}
wc_inline={{self.wc_open_inline}}{{self.wc_content}}{{self.wc_close_inline}}
wc_open={{code.pushlines(t="@@{{self.wc_open_inline}}\n@wrap {{self.wrapID}}")}}
wc_close={{code.pushlines(t="@parw 1\n@@{{self.wc_close_inline}}")}}
inline={{html._section_div_.<}}{{html._section_p_.<}}{{self.t}}{{html._section_p_.>}}{{html._section_div_.>}}
wc_open_inline={{html._section_div_.<}}{{html._section_p_.<}}{{self.t}}{{html._section_p_.>}}
wc_close_inline={{html.div.>}}
wc_content={{html._section_p_content_.<}}{{self.c}}{{html.p.>}}
sID=section
wrapID=_section_p_content_
t=section default title string
c=section default content data

If you examine any of other styles in the **Generic Groups**, you will find they have an identical set of attributes/methods. So once you are familiar with one of them, you know how to use all of them! Here is the help string for the **section** var, which applies to all of the generic groups:

Built-in help string for ***section*** DIV
section(t="title" c="content")

Common Parameters
t - The title to use for the section div
c = The content to use for the section div

Attributes
sID - the div identifier **section**
wrapID - the **@wrap** tag(s) *_section_p_content_*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert section div with text **t**
wc_inline(t,c) - insert section div with title **t** and content **c**
wc_open(t) - open a section div with title **t** and **@wrap _section_p_content_** ready for content

wc_close - close section div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close section div opened with **wc_open_inline**

@raw Versions

The **@raw** versions will emit using **@@**, which signals the output formatter to suppress any **@wrap** tags in effect. This is done to prevent any unintentional formatting from changing how the browser will render the markup. Let's try each of the methods using the default values, starting with **[section]**:

section default title string

If we look at the HTML for it, you will see:

Raw HTML emitted by the [section] variable

@@ section default title string

As mentioned, this is in the **@raw** section, so the parser emits the double **@@** prefix, which prevents the output formatter from prefixing any **@wrap** tags. Let's take a look at the remaining built-in attributes and how they render.

[section.with_content] renders like this:

section default title string

section default content data

If you examine the built-in help, you will see that these two versions rely on the parameters **t** and **c** to override default values. Thus, if we were to write the following:

Passing parameters to section methods

```
[section(t="My heading")]
[section(t="My with_content heading" c="My important content")]
```

The parser emits the following:

My heading

My with_content heading

My important content

So in any of the methods that take either **t** and/or **c**, you can easily specify your own values when you invoke the method in your markdown. Moving on, let's take a look at the **wc_open** method, which allows you to create a section but leave the **DIV** open to accept the content up until it encounters the **wc_close** method.

Consider the following markdown:

Using section.wc_open

```
[section.wc_open]
and now we have to write [section.wc_close] which will close the previous div.
We can keep typing lines, though, and they are added to this section until you close it
with [section.wc_close].
Okay, the next line in the user docs will contain: [section.wc_close]
[section.wc_close]
```

The parser emits the following:

section default title string

and now we have to write **[section.wc_close]** which will close the previous div.

We can keep typing lines, though, and they are added to this section until you close it with **[section.wc_close]**.

Okay, the next line in the user docs will contain: **[section.wc_close]**

You can also nest them, let's look at an example of that. Here I will just write **[section.wc_open]** three times in a row, followed by three **[section.wc_close]** tags.

section default title string

section default title string

section default title string

As we previously mentioned, all of the variables within a group contain the exact same attributes/methods. Given that, we can nest them within each other. For example:

Nesting other general DIVs

```
[section.wc_open(t="This is the section.wc_open markdown")]
  [syntax.wc_open(t="This is the syntax.wc_open markdown")]
    [toc.wc_open(t="This is the toc.wc_open markdown")]
      [plain.wc_open(t="This is the plain.wc_open markdown")]
      [plain.wc_close]
    [toc.wc_close]
  [syntax.wc_close]
[section.wc_close]
```

This is the section.wc_open markdown

This is the syntax.wc_open markdown

This is the toc.wc_open markdown

This is the plain.wc_open markdown

That is a good segue into changing the default values for these examples, as we touched on briefly earlier.

All of these attributes (across all of the different div types) use the same attribute names for parameters. **t** for the title and **c** for the content.

So, when we write **[section.with_content(t="my title" c="my content"]**, we will see this:

mytitle

my content

Similarly, if we write **[section(t="my different title"]**, we get:

my different title

And now if we got back and use that same nested sequence from before, we would get:

my different title

my different title

my different title

Recall how specifying a new value for any parameter in namespace variables (other than the **@code** namespace) is sticky. That is, when you override a value on a method call, that value will continue to be used until it is overridden again. That's an important concept to remember, as it applies throughout **smd**. If we use the **.with_content** instead, we get:

my different title

my content

See how that works? Remember, this goes back to the side effect discussed in the @var Namespace chapter that discussed when attribute values are updated they stay updated. So in this last example, the default values used were those from the previous time we used **section** in this chapter (I was going to say *section*, but it seems like I've worn that out already).

Okay, now let's look at the **inline** versions of the **[section]** variable.

Inline Versions

The **_inline** versions of the attribute methods match up to their **@raw** cousins. The only difference between these versions and their *non-_inline* counterparts is that they do not prefix the output with the **@raw** tag. Here are some examples of how the output looks when using them:

First, here's **[section.inline]**:

my different title

And now, **[section.wc_inline]**:

my different title

my content

Essentially what you get when using them is output wrapped with whatever the current **@wrap** tag happens to be. This gives you plenty of flexibility to style the output by only controlling the block elements that contain the **_inline** versions of each.

Currently, the **@wrap** tag is set to: *** ***. Let's set it to **@wrap nop**, and then add the same two previous markdown elements:

my different title

my different title

my content

Now they are identical to using the **non-*_inline*** versions, since **@wrap nop** is essentially telling the output formatter to write in raw mode! The final two inline modes will be covered in the next section on nesting DIVs, but there isn't any real magic to them. They match up to their counterparts exactly like the two we just reviewed...

Final notes on nesting DIVS

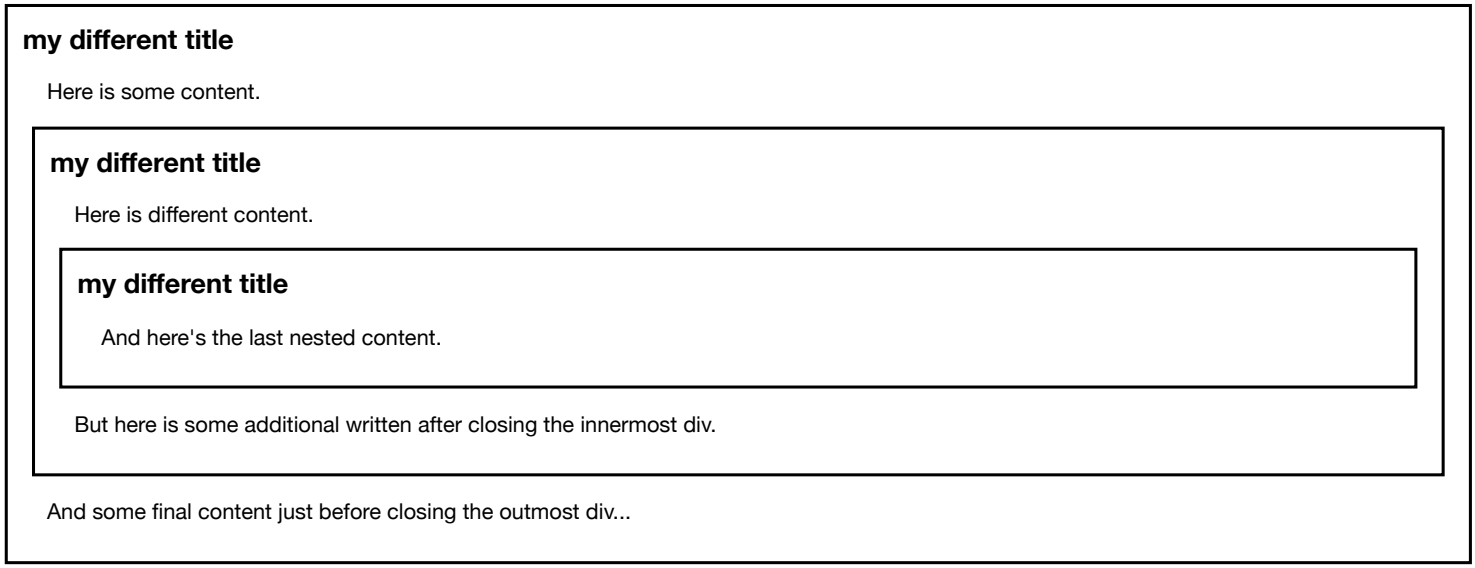
Before we leave Generic DIVs, let's look at a few more examples of nesting. You can get into some precarious formatting issues if you aren't careful, due to how most modern browsers treat block elements, as you'll see below.

In the first example, we are using **wc_open_inline** followed by content, then another **wc_open_inline**, etc. We are also using the **@raw** tag on each of the wc_* calls, to prevent the output formatter from wrapping the divs with the current wrap tags. Let's take a look at the markdown, and then what the parser emits:

Nesting wc_open_inline with content

```
@@[section.wc_open_inline]
  Here is some content.
  @@[section.wc_open_inline]
    Here is different content.
    @@[section.wc_open_inline]
      And here's the last nested content.
    @@[section.wc_close_inline]
    But here is some additional written after closing the innermost div.
  @@[section.wc_close_inline]
  And some final content just before closing the outmost div...
@@[section.wc_close_inline]
```

And here's how the browser will render that:



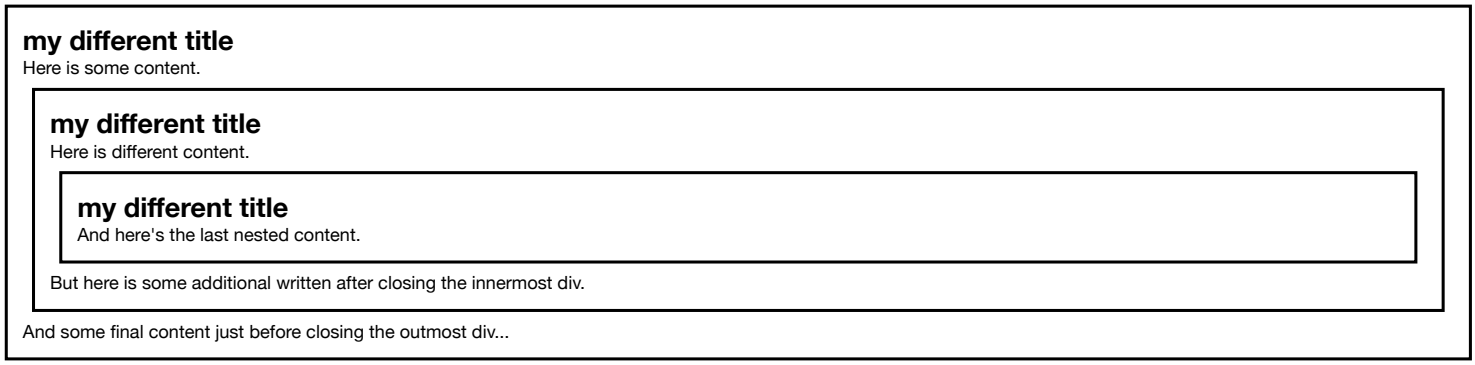
This works okay (as long as you use the **@raw** tag when emitting the inline divs, because **@wrap** is set to ***<p>***, so each line is wrapped as a paragraph.

In this next example, we will do the same thing, but without any line breaks. It looks similar, except that most modern browsers will close an open block tag if a new block tag is encountered. Unfortunately, there isn't any simple way to address this because you can't rely on the output formatter of **smd** to solve the issue (like we did above), because there is only a single line being output...

Nesting wc_open_inline with content on a single line

```
@@[section.wc_open_inline] Here is some content. [section.wc_open_inline] Here is
different content. [section.wc_open_inline] And here's the last nested content.
[section.wc_close_inline] But here is some additional written after closing the
innermost div. [section.wc_close_inline] And some final content just before closing the
outmost div... [section.wc_close_inline]
```

Which will render like this:



In this final example on nesting, we will use the raw attribute methods again, just writing the content on separate lines. This behaves exactly like the first version above, just perhaps a bit more readable since you don't need the **@raw** tags. First, let's look at the markdown:

Nesting @raw wc_open with content

```
[section.wc_open]
  Here is some content.[b]
  and more content
  [section.wc_open]
    Here is different content.
    [section.wc_open]
      And here's the last nested content.
    [section.wc_close]
    But here is some additional written after closing the innermost div.
    what about this?
  [section.wc_close]
  And some final content just before closing the outmost div...
  and more
  and more
[section.wc_close]
```

And now here's what the browser renders.

my different title

Here is some content.

and more content

my different title

Here is different content.

my different title

And here's the last nested content.

But here is some additional written after closing the innermost div.

what about this?

And some final content just before closing the outmost div...

and more

and more

Other Generic DIVs

The remaining Generic DIVs have all the same behaviour and attributes as **section**. The only difference in how they look goes back to how they are styled in the **smd.css** file. Try a few out, and/or take a look at the **tests/in/divs.md** unittest file to see them in action! Here is the online help for each of the remaining divs, **toc**, **syntax**, **review** and **plain**.

toc Generic DIV

Built-in help string for *toc* DIV
toc(t="title" c="content")

Common Parameters
t - The title to use for the toc div
c = The content to use for the toc div

Attributes
sID - the div identifier toc

wrapID - the **@wrap** tag(s) *_toc_p_content_*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert toc div with text **t**
wc_inline(t,c) - insert toc div with title **t** and content **c**
wc_open(t) - open a toc div with title **t** and **@wrap** **_toc_p_content_** ready for content
wc_close - close toc div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close toc div opened with **wc_open_inline**

toc.with_content example

[toc.with_content(t="My TOC heading" c="My TOC content")] renders like this:

My TOC Heading
My TOC content

syntax Generic DIV

Built-in help string for ***syntax*** DIV
syntax(t="title" c="content")

Common Parameters
t - The title to use for the syntax div
c = The content to use for the syntax div

Attributes
sID - the div identifier **syntax**
wrapID - the **@wrap** tag(s) *_syntax_p_content_*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert syntax div with text **t**
wc_inline(t,c) - insert syntax div with title **t** and content **c**
wc_open(t) - open a syntax div with title **t** and **@wrap** **_syntax_p_content_** ready for content
wc_close - close syntax div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close syntax div opened with **wc_open_inline**

syntax.with_content example

[syntax.with_content(t="My SYNTAX heading" c="My SYNTAX content")] renders like this:

My SYNTAX Heading
My SYNTAX content

review Generic DIV

Built-in help string for ***review*** DIV
review(t="title" c="content")

Common Parameters
t - The title to use for the review div
c = The content to use for the review div

Attributes

sID - the div identifier **review**
wrapID - the **@wrap** tag(s) *_review_p_content_*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert review div with text **t**
wc_inline(t,c) - insert review div with title **t** and content **c**
wc_open(t) - open a review div with title **t** and **@wrap** **_review_p_content_** ready for content
wc_close - close review div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close review div opened with **wc_open_inline**

review.with_content example

[review.with_content(t="My REVIEW heading" c="My REVIEW content")] renders like this:

My REVIEW Heading

My REVIEW content

plain Generic DIV

Built-in help string for ***plain*** DIV
plain(t="title" c="content")

Common Parameters
t - The title to use for the plain div
c = The content to use for the plain div

Attributes
sID - the div identifier **plain**
wrapID - the **@wrap** tag(s) *_plain_p_content_*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert plain div with text **t**
wc_inline(t,c) - insert plain div with title **t** and content **c**
wc_open(t) - open a plain div with title **t** and **@wrap** **_plain_p_content_** ready for content
wc_close - close plain div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close plain div opened with **wc_open_inline**

plain.with_content example

[plain.with_content(t="My PLAIN heading" c="My PLAIN content")] renders like this:

My PLAIN Heading

My PLAIN content

Source DIV

The **source** DIV provides formatting for inline source code, using the **<code>** HTML tag. It is used throughout the user manual displaying sample **smd** markdown as well as variable definitions.

Here is the definition of **source** (displayed with **source**, of course) and it's associated **@html** and **@var** variables:

@html Support for *source*

source=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=code
style=font-size:1.4em;font-weight:bold

_source_content_=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=code
style=font-size:1.2em

_source_div_=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=div
class=source

@var definition for *source*

source=
_format=@@ {{self.inline}}
with_content=@@ {{self.wc_inline}}
wc_inline={{self.wc_open_inline}}{{self.wc_content}}{{self.wc_close_inline}}
wc_open={{code.pushlines(t="@@{{self.wc_open_inline}}\n@wrap {{self.wrapID}}")}}
wc_close={{code.pushlines(t="@parw 1\n@@{{self.wc_close_inline}}")}}
inline={{html._source_div_.<}}{{html._source_.<}}{{self.t}}{{html._source_.>}}{{html._source_div_.>}}
wc_open_inline={{html._source_div_.<}}{{html._source_.<}}{{self.t}}{{html._source_.>}}
wc_close_inline={{html.div.>}}
wc_content={{html._source_content_.<}}{{self.c}}{{html._source_content_.>}}
sID=source
wrapID=_source_content_
t=@var definition for source
c=source default content data

If you compare the declaration of **source** to any of the **generic** DIVs, the first thing you'll notice is that both the *_source_p_* and *_source_p_content_* **@html** variables have been replaced with *_source_* and *_source_content_*. This minor difference essentially causes content to be wrapped with the **<code> @html** tag instead of the **<p>** tag, like all of the **generic** DIVs.

Besides that, it's pretty much identical to the **generic** DIVs we discussed in the previous section.

source DIV

Built-in help string for ***source*** DIV
source(t="title" c="content")

Common Parameters
t - The title to use for the source div
c = The content to use for the source div

Attributes
sID - the div identifier **source**
wrapID - the **@wrap** tag(s) *_source_content_*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert source div with text **t**
wc_inline(t,c) - insert source div with title **t** and content **c**
wc_open(t) - open a source div with title **t** and **@wrap** **_source_content_** ready for content
wc_close - close source div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close source div opened with **wc_open_inline**

source.with_content example

[source.with_content(t="My SOURCE heading" c="My SOURCE content")] renders like this:

My SOURCE Heading My SOURCE content

As you can see by reviewing the output, the **<code> @html** tag does not apply any formatting to the provided content, so when you use it, adding the appropriate formatting and line breaks is quite important. Let's try that again:

source.with_content example

[source.with_content(t="My SOURCE heading" c="[bb]My SOURCE content")] renders like this:

My SOURCE Heading

My SOURCE content

When using **source**, you will likely want nest the output inside another DIV in order to prevent it from smashing into the left margin of the browser window. There are many different ways to accomplish this; in the user manual, this is normally done by wrapping the **source** DIV with either the **bmgreybg** or **bigmargin @html** tag like this:

source.with_content example wrapped with *bigmargin* tag

[*bigmargin*._open]

[*source.with_content*(t="My SOURCE heading" c="[bb]My SOURCE content")]

[*bigmargin*._close]

renders like this:

My SOURCE Heading

My SOURCE content

Note DIVs

Note DIVs provide formatting for inline notes. They are similar to the **generic** DIVs, but instead of having a dedicated class for their containing DIV, they all use the **extras** class for their DIV container, and then have styling specific to the content contained within. In addition, they do not support the concept of **title/content**, i.e. the variables **t,c**. Instead, only **c** is used to specify the content.

Not surprisingly, one of the builtins for the **Note** DIVs is called simply **note**. We will start our examination of this class of DIVs by taking a look at the actual definition of **note** and it's associated **@html** and **@var** variables:

@html Support for *note*

_note_div_=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=div
class=extras

_note_p_=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=p
class=note

@var definition for *note*

note=
_format=@@ {{self.inline}}
with_content=@@ {{self.wc_inline}}
wc_inline={{self.wc_open_inline}}{{self.wc_content}}{{self.wc_close_inline}}
wc_open={{code.pushlines(t="@@{{self.wc_open_inline}}\n@wrap {{self.wrapID}}")}}

wc_close={{code.pushlines(t="@parw 1\n@@{{self.wc_close_inline}}")}}
nd=@@{{self.inline_nd}}
nd_inline={{self.inline_nd}}
nd_open={{code.pushlines(t="@@{{self.nd_open_inline}}\n@wrap {{self.wrapID}}")}}
nd_close={{code.pushlines(t="@parw 1\n@@{{self.nd_close_inline}}")}}
inline={{html._note_div_.<}}{{self.inline_nd}}{{html._note_div_.>}}
inline_nd={{html._note_p_.<}}{{self.c}}{{html._note_p_.>}}
wc_open_inline={{html._note_div_.<}}{{self.nd_open_inline}}
wc_close_inline={{html.p.>}}{{html.div.>}}
wc_content={{self.c}}
nd_open_inline={{html._note_p_.<}}
nd_close_inline={{html.p.>}}
sID=note
wrapID=nop
c=note default content data
t=You can also install smd in edit mode (pip install -e .). This will point the installation of the smd package to the current cloned repository directory instead of copying it to your site-packages installation directory; useful if you plan on making changes and don't want to reinstall each time you make a change.

Just like how all the attributes/methods of the **Generic Groups** are similar to the those in **section** DIV, if you examine any of other styles in the **Note Groups**, you will find they have an identical set of attributes/methods. So once you are familiar with those in the **note** DIV, you know how to use all of them! Here is the help string for the **note** var, which applies to all of the note groups:

note Note DIV

Built-in help string for *note* DIV
note(c="content")

Common Parameters
c = The content to use for the note div

Attributes
sID - the div identifier **note**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
with_content(c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(c) - insert note DIV with content **c**
inline_nd(c) - insert note with content **c** without DIV wrapper
wc_inline(c) - insert note DIV with content **c**
wc_open - open a note DIV and **@wrap nop** ready for content
wc_close - close note DIV opened with **wc_open**
wc_open_inline - like **wc_open** but does not use **@wrap**
wc_close_inline - close note DIV opened with **wc_open_inline**

No DIV **variants**
nd(c) - **inline_nd** with raw prefix (i.e. *@@ inline_nd*)
nd_inline(c) - same as **inline_nd**
nd_open - open note without DIV wrapper using **wrap nop**
nd_close - close note opened with **nd_open**
nd_open_inline - like **nd_open** but does not use **@wrap**
nd_close_inline - close note opened with **nd_open_inline**

Like we mentioned above, the **Note** DIVs have a similar set of attribute and methods, just like the other DIVs we've covered so far. The real difference is the addition of the ***nd, _nd*** variants to the methods. **ND**, in this context, stands for **no div**, which, as you might guess, are variants that do not emit **DIV** wrappers around the content. These are most useful when combined with the **AV Shot** support, as you will see later in the chapters that cover the examples. Here are a few examples to illustrate how these look.

@raw Versions

The **@raw** versions will emit using **@@**, which signals the output formatter to suppress any **@wrap** tags in effect. This is done to prevent any unintentional formatting from changing how the browser will render the markup. Let's try each of the methods using the default values, starting with **[note]**:

> note default content data

If we look at the HTML for it, you will see:

Raw HTML emitted by the [note] variable

```
@@ <div class="extras"><p class="note">note default content data</p></div>
```

Given these are in the **@raw** section, the parser emits the double **@@** prefix, preventing the output formatter from prefixing any **@wrap** tags. Let's take a look at the remaining built-in attributes and how they render.

[note.with_content] renders like this:

> note default content data

If you examine the built-in help, you will see that these two versions rely on the parameter **c** to override default values. Thus, if we were to write the following:

Passing a parameter to note methods

```
[note(c="My content with no method specified")]
[note.with_content(c="My content using the *with_content*")]
```

The parser emits the following:

> My content with no method specified

> My content using the *with_content*

So, in the methods that take **c**, you can specify your own value when you invoke the method in your markdown. Moving on, let's take a look at the **wc_open** method, which allows you to create a section but leave the **DIV** open to accept the content up until it encounters the **wc_close** method.

Consider the following markdown:

Using note.wc_open

```
[note.wc_open]
Now we need to write [e_var.em(t="note.wc_close")] to close the previous div.
As we keep typing lines, they are added to this note until you close it with
[e_var.em(t="note.wc_close")].

The next line will contain: [e_var.em(t="note.wc_close")]
[note.wc_close]
```

The parser emits the following:

> Now we need to write *[note.wc_close]* to close the previous div. As we keep typing lines, they are added to this note until you close it with *[note.wc_close]*. The next line will contain: *[note.wc_close]*

Take note that breaking the lines and/or inserting blank lines do not affect the formatting; whitespace is ignored. Also, unlike the **Generic Group**, the **Note Group** doesn't support nesting.

Okay, now let's look at the **inline** versions of the **[note]** variable.

Inline Versions

Like the with the **Generics**, the **_inline** versions of the attribute methods match up to their **@raw** cousins. The only difference between these versions and their *non-_inline* counterparts is that they do not prefix the output with the **@raw** tag. Here are some examples of how the output looks when using them:

First, here's **[note.inline]**:

> note default content data

And now, **[note.wc_inline]**:

note default content data

As is readily evident, the *wc_** variants of the **Note DIVs** are somewhat redundant, given the output is identical as just shown. However, they have been maintained in order to keep consistency between the **Generic** and **Source** DIVs.

Just like the **Generic DIVs**, what you get when using them is output wrapped with whatever the current **@wrap** tag happens to be. This gives you plenty of flexibility to style the output by only controlling the block elements that contain the **_inline** versions of each.

As we eluded to earlier, the **Note DIVs** do have an additional set of methods referred to as the *No-DIV* or *nd* variants, which emit the content without using a DIV tag. This can be useful in many different situations, including when using in combination with the **A/V and avshot** builtins. This allows notes to be inserted in either the left or right hand columns inline without affecting the floats in effect to format the AV script.

For example, **[note.nd_inline]** emits note default content data which renders as:

note default content data

Currently, the **@wrap** tag is set to: *** ***. Let's set it to **@wrap nop**, and then write all three inline markdown methods (**note.inline**, **note.wc_inline** and **note.nd_inline**):

note default content data

note default content data

note default content data

Here you can see that the *nd_* version will be styled according to it's context within the HTML document (in addition to the styling defined as part of the **note** CSS class in **smd.css**.)

Other Note DIVs

The remaining **Note DIVs** have all the same behavior and attributes as **note**. The only difference in how they look goes back to how they are styled in the **smd.css** file. Try a few out, and/or take a look at the **tests/in/divs.md** unittest file to see them in action! Here is the online help for each of the remaining **Note DIVs**: **box**, **generic**, **greyout**, **important**, **question** and **vo**.

box Note DIV

Built-in help string for ***box*** DIV
box(c="content")

Common Parameters
c = The content to use for the box div

Attributes
sID - the div identifier **box**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
with_content(c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(c) - insert box DIV with content **c**
inline_nd(c) - insert box with content **c** without DIV wrapper
wc_inline(c) - insert box DIV with content **c**
wc_open - open a box DIV and **@wrap nop** ready for content
wc_close - close box DIV opened with **wc_open**
wc_open_inline - like **wc_open** but does not use **@wrap**
wc_close_inline - close box DIV opened with **wc_open_inline**

No DIV **variants**
nd(c) - **inline_nd** with raw prefix (i.e. *@@ inline_nd*)
nd_inline(c) - same as **inline_nd**
nd_open - open box without DIV wrapper using **wrap nop**
nd_close - close box opened with **nd_open**
nd_open_inline - like **nd_open** but does not use **@wrap**
nd_close_inline - close box opened with **nd_open_inline**

box example

[box(c="My BOX content")] renders like this:

My BOX content

generic Note DIV

Built-in help string for *generic* DIV
generic(c="content")

Common Parameters
c = The content to use for the generic div

Attributes
sID - the div identifier **generic**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
with_content(c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(c) - insert generic DIV with content **c**
inline_nd(c) - insert generic with content **c** without DIV wrapper
wc_inline(c) - insert generic DIV with content **c**
wc_open - open a generic DIV and **@wrap nop** ready for content
wc_close - close generic DIV opened with **wc_open**
wc_open_inline - like **wc_open** but does not use **@wrap**
wc_close_inline - close generic DIV opened with **wc_open_inline**

No DIV **variants**
nd(c) - **inline_nd** with raw prefix (i.e. *@@ inline_nd*)
nd_inline(c) - same as **inline_nd**
nd_open - open generic without DIV wrapper using **wrap nop**
nd_close - close generic opened with **nd_open**
nd_open_inline - like **nd_open** but does not use **@wrap**
nd_close_inline - close generic opened with **nd_open_inline**

generic example

[generic(c="My GENERIC content")] renders like this:

My GENERIC content

greyout Note DIV

Built-in help string for *greyout* DIV
greyout(c="content")

Common Parameters
c = The content to use for the greyout div

Attributes
sID - the div identifier **greyout**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
with_content(c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(c) - insert greyout DIV with content **c**
inline_nd(c) - insert greyout with content **c** without DIV wrapper
wc_inline(c) - insert greyout DIV with content **c**
wc_open - open a greyout DIV and **@wrap nop** ready for content
wc_close - close greyout DIV opened with **wc_open**

wc_open_inline - like **wc_open** but does not use **@wrap**
wc_close_inline - close greyout DIV opened with **wc_open_inline**

No DIV **variants**
nd(c) - **inline_nd** with raw prefix (i.e. *@@ inline_nd*)
nd_inline(c) - same as **inline_nd**
nd_open - open greyout without DIV wrapper using **wrap nop**
nd_close - close greyout opened with **nd_open**
nd_open_inline - like **nd_open** but does not use **@wrap**
nd_close_inline - close greyout opened with **nd_open_inline**

greyout example

[greyout(c="My GREYOUT content")] renders like this:

My GREYOUT content

important Note DIV

Built-in help string for ***important*** DIV
important(c="content")

Common Parameters
c = The content to use for the important div

Attributes
sID - the div identifier **important**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
with_content(c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(c) - insert important DIV with content **c**
inline_nd(c) - insert important with content **c** without DIV wrapper
wc_inline(c) - insert important DIV with content **c**
wc_open - open a important DIV and **@wrap nop** ready for content
wc_close - close important DIV opened with **wc_open**
wc_open_inline - like **wc_open** but does not use **@wrap**
wc_close_inline - close important DIV opened with **wc_open_inline**

No DIV **variants**
nd(c) - **inline_nd** with raw prefix (i.e. *@@ inline_nd*)
nd_inline(c) - same as **inline_nd**
nd_open - open important without DIV wrapper using **wrap nop**
nd_close - close important opened with **nd_open**
nd_open_inline - like **nd_open** but does not use **@wrap**
nd_close_inline - close important opened with **nd_open_inline**

important example

[important(c="My IMPORTANT content")] renders like this:

MY IMPORTANT CONTENT

question Note DIV

Built-in help string for ***question*** DIV
question(c="content")

Common Parameters

c = The content to use for the question div

Attributes
sID - the div identifier **question**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
with_content(c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(c) - insert question DIV with content **c**
inline_nd(c) - insert question with content **c** without DIV wrapper
wc_inline(c) - insert question DIV with content **c**
wc_open - open a question DIV and **@wrap nop** ready for content
wc_close - close question DIV opened with **wc_open**
wc_open_inline - like **wc_open** but does not use **@wrap**
wc_close_inline - close question DIV opened with **wc_open_inline**

No DIV **variants**
nd(c) - **inline_nd** with raw prefix (i.e. *@@ inline_nd*)
nd_inline(c) - same as **inline_nd**
nd_open - open question without DIV wrapper using **wrap nop**
nd_close - close question opened with **nd_open**
nd_open_inline - like **nd_open** but does not use **@wrap**
nd_close_inline - close question opened with **nd_open_inline**

question example

[question(c="My QUESTION content")] renders like this:

My QUESTION content

vo Note DIV

Built-in help string for ***vo*** DIV
vo(c="content")

Common Parameters
c = The content to use for the vo div

Attributes
sID - the div identifier **vo**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
with_content(c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(c) - insert vo DIV with content **c**
inline_nd(c) - insert vo with content **c** without DIV wrapper
wc_inline(c) - insert vo DIV with content **c**
wc_open - open a vo DIV and **@wrap nop** ready for content
wc_close - close vo DIV opened with **wc_open**
wc_open_inline - like **wc_open** but does not use **@wrap**
wc_close_inline - close vo DIV opened with **wc_open_inline**

No DIV **variants**
nd(c) - **inline_nd** with raw prefix (i.e. *@@ inline_nd*)
nd_inline(c) - same as **inline_nd**
nd_open - open vo without DIV wrapper using **wrap nop**
nd_close - close vo opened with **nd_open**
nd_open_inline - like **nd_open** but does not use **@wrap**
nd_close_inline - close vo opened with **nd_open_inline**

vo example

[vo(c="My VO content")] renders like this:

> My VO content

Okay, onward to the **Miscellaneous DIVs**.

Miscellaneous DIVs

The following DIVs don't fit into any of the other categories, although they share much of the same syntax and semantics. **extras** and **divxp** are very simple DIVs, and **terminal** and **terminal2** are closest to the **section** DIVs, and used primarily in the user documentation for wrapping content that covers command line and built-in help.

extras is very basic, allowing you to wrap content **c** inside a *class="extras"* ** **.

extras Miscellaneous DIV

> Built-in help string for ***extras*** DIV
> *extras(c="content")*
>
> **Parameters**
> **c** = The content to use for the extras
>
> **Methods**
> ***NONE*** - emit content **c** wrapped with div class="extras"

extras example

[extras(c="My EXTRAS content")] renders like this:

My EXTRAS content

Like **extras**, **divxp** is also very basic, allowing you to wrap content **c** inside a *class="extras"* ** **, but also wrapping inside an HTML paragraph tag **<p>**.

divxp Miscellaneous DIV

> Built-in help string for ***divxp*** DIV
> *divxp(c="content")*
>
> **Common Parameters**
> **c** = The content to use for the divxp
>
> **Public Methods**
> ***NONE*** - If no method specified, uses **inline** with raw prefix (i.e. *@@ inline*)
> **inline(c)** - emit content **c** wrapped with div class="extras" p
> **open** - invokes *_open*
> **close** - invokes *_close*
>
> **Private Methods**
> **_open** - emit open tags *div class="extras" p*
> **_close** - emit close tags */p /div*

divxp example

[divxp(c="My DIVXP content")] renders like this:

My DIVXP content

Moving on to the **terminal** DIVs, they are closely related to the **section** divs, so if you're familiar with those, then the **terminal** counterparts will be easy to use. Let's start by examining the definition of the variables that make up **terminal**:

@html Support for *terminal*

```
_terminal_div_=
  _tag=div
  class=plain
  style=padding-left:3em;padding-right:3em
  _format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>

_terminal_prewrap_=
  _format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
  _tag=pre
  style=background-color:lightgray;font-style:italic;padding:5px 10px;{{html.prewrap.style}}
  class=divTitle

_terminal_prewrap_content_=
  _format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
  _tag=pre
  style=background-color:lightgray;font-weight:100;padding:5px 10px;{{html.prewrap.style}}
  class=divTitle
```

@var definition for *terminal*

```
terminal=
  _format=@@ {{self.inline}}
  inline={{html._terminal_div_.<}}{{html._terminal_prewrap_content_.<}}{{self.t}}{{html._terminal_prewrap_content_.>}}
{{html._terminal_div_.>}}
  with_content=@@ {{self.wc_inline}}
  wc_inline={{self.wc_open_inline}}{{self.c}}{{self.wc_close_inline}}
  wc_open={{code.pushlines(t="@wrap {{self.wrapID}}\n{{self.wc_open_inline}}")}}
  wc_close={{code.pushlines(t="{{self.wc_close_inline}}\n@parw 1")}}
  wc_open_inline={{html._terminal_div_.<}}{{html._terminal_prewrap_.<}}{{self.t}}{{html._terminal_prewrap_.>}}
{{html._terminal_prewrap_content_.<}}
  wc_close_inline={{html.prewrap.>}}{{html.div.>}}
  t=var.[self.sID] default title
  c=var.[self.sID] default content data
  wrapID=nop
  sID=terminal
```

Just like with **section** DIVs, **terminal** DIVs have **@raw** methods which include the default *[terminal]* and *[terminal.with_content]*, as well as the inline variants: *inline*, *wc_inline*, *wc_open*, *wc_close*, *wc_open_inline*, *wc_close_inline*. For a more indepth look at these methods, refer to the documentation on **section** DIVs, as we are not going to repeat it here. With that, let's have a look at the builtin help and an example use of **terminal**.

terminal Miscellaneous DIV

Built-in help string for ***terminal*** DIV

terminal(t="title" c="content")

Common Parameters
t - The title to use for the terminal div
c = The content to use for the terminal div

Attributes
sID - the div identifier **terminal**
wrapID - the **@wrap** tag(s) *nop*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert terminal div with text **t**
wc_inline(t,c) - insert terminal div with title **t** and content **c**
wc_open(t) - open a terminal div with title **t** and **@wrap nop** ready for content
wc_close - close terminal div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close terminal div opened with **wc_open_inline**

terminal example

[terminal(t="My TERMINAL title")] renders like this:

```
My TERMINAL title
```

terminal2 is very similar to **terminal**, in fact it uses the same **@html** variables, but overrides *wc_open*, *wc_close*, *wc_open_inline* and *wc_close_inline*, and also provides some additional methods. Let's have a look at the definition of **terminal2**:

@var definition for *terminal2*

terminal2=
_format=@@ {{self.inline}}
inline={{html._terminal_div_.<}}{{html._terminal_prewrap_content_.<}}{{self.t}}{{html._terminal_prewrap_content_.>}}
{{html._terminal_div_.>}}
with_content=@@ {{self.wc_inline}}
wc_inline={{self.wc_open_inline}}{{self.c}}{{self.wc_close_inline}}
wc_open={{code.pushlines(t="@@{{self.wc_open_inline}}\n@wrap {{self.wrapID}}")}}
wc_close={{code.pushlines(t="@parw 1\n@@{{self.wc_close_inline}}")}}
wc_open_inline={{html._terminal_div_.<}}{{html._terminal_prewrap_.<}}{{self.t}}{{html._terminal_prewrap_.>}}
wc_close_inline={{html.div.>}}
t=var.[self.sID] default title
c=var.[self.sID] default content data
wrapID=html._terminal_prewrap_content_
sID=terminal2
wc_open_content={{code.pushlines(t="@wrap nop\n{{html._terminal_prewrap_content_.<}}")}}
wc_close_content={{code.pushlines(t="{{html._terminal_prewrap_content_.>}}\n@parw 1")}}

If you compare the definitions of **terminal** and **terminal2**, you'll notice several differences. Let's have a look at those now:

First, the *wc_open* method reverses the order of emitting the opening HTML tags and setting up the **@wrap** tag, which is also different. Before, the **@wrap** tag was **nop**, and now it is set to *html._terminal_prewrap_content_*. And this order change also requires that *wc_close* be updated to also reverse the cleanup and closing tag emittal.

Second, *wc_open_inline* does not emit the open tag for *html._terminal_prewrap_content_* like it does in **terminal**, because the **wrapID** has been changed to automatically emit the wrapper tag around any content.

Finally, two new methods have been added, *wc_open_content* and *wc_close_content*, which provide the ability to change the default behavior for accepting content back to how **terminal** works. That is, when *wc_open_content* is invoked, the **@wrap** tag is set to **nop**, and the *html._terminal_prewrap_content_* is opened until *wc_close_content* is called. This minor change in the semantics of **terminal2** was done to provide more control over how the formatting of content is done.

Here's a look at the builtin help for **terminal2** along with an example usage:

terminal2 Miscellaneous DIV

Built-in help string for *terminal2* DIV

terminal2(t="title" c="content")

Common Parameters
t - The title to use for the terminal2 div
c = The content to use for the terminal2 div

Attributes
sID - the div identifier **terminal2**
wrapID - the **@wrap** tag(s) *html._terminal_prewrap_content_*

Methods
NONE - If no method specified, uses **inline** with raw (i.e. *@@ inline*)
with_content(t,c) - **wc_inline** raw (i.e. *@@ wc_inline*)
inline(t) - insert terminal2 div with text **t**
wc_inline(t,c) - insert terminal2 div with title **t** and content **c**
wc_open(t) - open a terminal2 div with title **t** and **@wrap html._terminal_prewrap_content_** ready for content
wc_close - close terminal2 div opened with **wc_open**
wc_open_inline(t) - like **wc_open** but does not use **@wrap**
wc_close_inline - close terminal2 div opened with **wc_open_inline**
wc_open_content - changes **wrapID** to **nop** and opens the content tag *html._terminal_prewrap_content_*
wc_close_content - close the *html._terminal_prewrap_content_* and restore previous **wrapID**

terminal2 example

[terminal2(t="My TERMINAL2 title")] renders like this:

```
My TERMINAL2 title
```

To see many more examples of the **terminal** and **terminal2** DIVs, review the user manual content; they are used quite heavily throughout the docs. Let's move on to the **List Group** builtins.

Lists

Lists provide formatting for both ordered and unordered lists, covering the **HTML** tags **<ol>** and **<ul>**. Their attributes and methods are similar to **Note** DIVs; they support the concept of **content**, that is, the variable **c**, although in most all cases the **wc_open** and **wc_close** wrappers are used to create & manage lists. They also support nesting, and can be intermixed so long as the closing methods are called in proper order.

One key difference with how the **Lists** operate is that by default, they do **not** wrap content with an HTML ** ** tag. When you use any of the common methods such as **with_content** or **wc_open**/**wc_close**, any **@wrap** tags in effect are ignored, and the methods emit in raw mode. In fact, they operate similar to the *no div* i.e. **nd** methods available with all the **Note** DIVs. Because of this, they provide two special wrapper mechanisms **_tag** and **_wrap** which can be used to wrap lists with a specific content.

The **_tag** support is implemented with the methods **wc_tag_open** and **wc_tag_close**, both of which rely on an **@html** variable specified in the **tag** attribute, which by default is **html.divx**. This will wrap the specified list with * * ... * *. You can change the value of **tag** to any valid **@html** variable to override the default.

The **_wrap** support is implemented with the methods **wc_wrap_open** and **wc_wrap_close**, both of which rely on either an **@html** or **@var** variable specified in the **wrap** attribute, which by default is **var.divxp**. Whichever variable type is used for **wrap**, it must contain one method named **_open** and another named **_close**, and those methods then emit the open and close tags respectively. This will wrap the specified list with * * ... * *. You can change the value of **wrap** to any valid **@html** or j**@var** variable to override the default, just make sure they have both an **_open** and **_close** method.

Okay, let's start with the unordered list **ulist**, and begin our examination of this class by taking a look at the actual definition of **ulist** and it's associated **@html** and **@var** variables:

@html Support for *ulist*

ulist=
_format=<{{self._tag}}{{self._public_attrs_}}></{{self._tag}}>
_tag=ul
class=ulist

@var definition for *ulist*

ulist=
_format=@@ {{self._inline}}
with_content=@@ {{self._open_inline}}{{self._content}}{{self._close_inline}}
wc_open={{code.pushlines(t="{{self._open}}")}}
wc_close={{code.pushlines(t="{{self._close}}")}}
wc_tag_open={{code.pushlines(t="@@{{[self.tag].<}}\n{{self._open}}")}}
wc_tag_close={{code.pushlines(t="{{self._close}}\n@@{{[self.tag].>}}")}}
wc_wrap_open={{code.pushlines(t="@@{{[self.wrap]._open}}\n{{self._open}}")}}
wc_wrap_close={{code.pushlines(t="{{self._close}}\n@@{{[self.wrap]._close}}")}}
tag=html.divx
wrap=var.divxp
_open=@wrap li\n@@{{self._open_inline}}
_close=@@{{self._close_inline}}\n@parw 1
_content={{html.li.<}}{{self.c}}{{html.li.>}}
_inline={{html.ulist.<}}{{self._content}}{{html.ulist.>}}
_open_inline={{html.ulist.<}}
_close_inline={{html.ulist.>}}
sID=ulist
c=var.[self.sID] default content data

Just like how all the attributes/methods of the **Generic Groups** are similar to the those in **section** DIV, if you examine any of other styles in the **Lists**, you will find they have an identical set of attributes/methods. So once you are familiar with those in **ulist**, you know how to use all of them!

Here is the help string for the **ulist** var, which applies to all of the lists:

ulist List DIV

Built-in help string for *ulist* DIV
ulist(c="content")

Common Parameters
c = The content to use for the ulist

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *ulist.<* **_content** *ulist.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.ulist.<*
_close_inline - emit *html.ulist.>*

Using the List Groups

Like we mentioned above, the **List** groups have a similar set of attribute and methods to the **Generic** DIVs. Here are a few examples to illustrate how these look.

Using the ulist Group

```
[ulist.wc_tag_open]
  unordered list item 1
  unordered list item 2
  unordered list item 3
[ulist.wc_tag_close]
```

which renders as:

- unordered list item 1
- unordered list item 2
- unordered list item 3

In this next example, we'll see how to nest list groups. Take the following markdown:

Nesting the List Groups

```
[olistRoman.wc_wrap_open]
  This is item 1
  This is item 2
  [olistAlpha.wc_open]
    Subitem 1
    Subitem 2
    Subitem 3
  [olistAlpha.wc_close]
  This is item 3
[olistRoman.wc_wrap_close]
```

Which will render as:

I. This is item 1
II. This is item 2
    A. Subitem 1
    B. Subitem 2
    C. Subitem 3
III. This is item 3

That's really about all there is to using the **List Groups**. Essentially, you'll use either the **wc_tag_open** or **wc_wrap_open** variants on the initial or outer list group (if you are nesting them), and then just the **wc_open** / **wc_close** variants on any of the inner lists.

More on _tag and _wrap Methods

Recall that the **List Groups** introduce the specialized **tag** and **wrap** variations of the **wc_open** / **wc_close** methods. Because none of the list group macros by default will wrap content using HTML ** ** sections, these two variations provide the extensions to accomplish that.

If you need the ability to wrap list output with a single HTML tag, then the **wc_tag_open**/**wc_tag_close** methods along with the **tag** attribute are exactly what you need. Simply set **tag** equal to any HTML variable, and the output will be wrapped using the open/close tag builtins. For example, if you set **tag** = **divx**, then your output is written as **[divx.<]** *your content* **[divx.>]**.

Similarly, if you need the ability to wrap list output with something more complex that a single HTML tag, then the **wc_wrap_open**/**wc_wrap_close** methods along with the **wrap** attribute will be what you need. You can use either an **@html** or **@var** variable for the value of **wrap**, the only requirement being that they contain the methods **_open** and **_close**. In this case, you set **wrap** = **var.divxp**, then your output will be written as **[divxp._open]** *your content* **[divxp._close]**. Let's go ahead and expand the **_open**/**_close** methods on **divxp** to see what would actually be emitted:

 your content

These are fairly simple examples, but you should be able to see how they can easily be extended to a much more complex sequence of content.

Other List Groups

The remaining **List Groups** all have the same behavior and attributes as **ulist**. The only difference in how they look goes back to how they are styled in the **smd.css** file. Try a few out, and/or take a look at the **tests/in/divs.md** unittest file to see them in action! Here is the online help for each of the remaining **List Groupss**: **ulistplain**, **olist**, **olistAlpha**, **olistGreek**, **olistRoman**, **olistalpha**, **olistgreek**, and **olistroman**.

ulistplain Group

Built-in help string for ***ulistplain***
ulistplain(c="content")

Common Parameters
c = The content to use for the ulistplain

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *ulistplain.<* **_content** *ulistplain.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.ulistplain.<*
_close_inline - emit *html.ulistplain.>*

ulistplain example

[ulistplain(c="My ULISTPLAIN content")] renders like this:

My ULISTPLAIN content

olist Group

Built-in help string for ***olist***
olist(c="content")

Common Parameters
c = The content to use for the olist

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *olist.<* **_content** *olist.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.olist.<*
_close_inline - emit *html.olist.>*

olist example

[olist(c="My OLIST content")] renders like this:

1. My OLIST content

olistAlpha Group

Built-in help string for ***olistAlpha***
olistAlpha(c="content")

Common Parameters
c = The content to use for the olistAlpha

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *olistAlpha.<* **_content** *olistAlpha.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.olistAlpha.<*
_close_inline - emit *html.olistAlpha.>*

olistAlpha example

[olistAlpha(c="My olistAlpha content")] renders like this:

A. My olistAlpha content

olistGreek Group

Built-in help string for ***olistGreek***
olistGreek(c="content")

Common Parameters
c = The content to use for the olistGreek

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *olistGreek.<* **_content** *olistGreek.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.olistGreek.<*
_close_inline - emit *html.olistGreek.>*

olistGreek example

[olistGreek(c="My olistGreek content")] renders like this:

1. My olistGreek content

olistRoman Group

Built-in help string for ***olistRoman***
olistRoman(c="content")

Common Parameters
c = The content to use for the olistRoman

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods

NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *olistRoman.<* **_content** *olistRoman.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.olistRoman.<*
_close_inline - emit *html.olistRoman.>*

olistRoman example

[olistRoman(c="My olistRoman content")] renders like this:

I. My olistRoman content

olistalpha Group

Built-in help string for ***olistalpha***
olistalpha(c="content")

Common Parameters
c = The content to use for the olistalpha

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *olistalpha.<* **_content** *olistalpha.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.olistalpha.<*
_close_inline - emit *html.olistalpha.>*

olistalpha example

[olistalpha(c="My olistalpha content")] renders like this:

a. My olistalpha content

olistgreek Group

Built-in help string for ***olistgreek***

olistgreek(c="content")

Common Parameters
c = The content to use for the olistgreek

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *olistgreek.<* **_content** *olistgreek.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.olistgreek.<*
_close_inline - emit *html.olistgreek.>*

olistgreek example

[olistgreek(c="My olistgreek content")] renders like this:

a. My olistgreek content

olistroman Group

Built-in help string for ***olistroman***
olistroman(c="content")

Common Parameters
c = The content to use for the olistroman

Attributes
tag = An **@html** variable used to wrap content when **_tag** methods are used
wrap = An **@html** or **@var** variable used to wrap content when **_wrap** methods are used

Public Methods
NONE - If no method specified, uses **_inline** with raw prefix (i.e. *@@ _inline*)
with_content(c) - emit content **c** wrapped with *_open_inline* / *_close_inline*
wc_open(c) - emit *_open*
wc_close - emit *_close*
wc_tag_open(c) - emit *_open* prefixed with *tag.<*
wc_tag_close - emit *_close* with *tag.>* appended
wc_wrap_open(c) - emit *_open* prefixed with *wrap._open*
wc_wrap_close - emit *_close* with *wrap._close* appended

Private Methods
_content - emit *html.li.<* **c** *html.li.>*
_inline - emit *olistroman.<* **_content** *olistroman.>*
_open - set **@wrap** li then emit *_open_inline*
_close - emit *_close_inline* then **@parw 1**
_open_inline - emit *html.olistroman.<*
_close_inline - emit *html.olistroman.>*

olistroman example

[olistroman(c="My olistroman content")] renders like this:

i. My olistroman content

And that wraps up the discussion on the **List Groups** available in the builtin **divs.md**.

DIVs Summary

DIVs are one of the fundemental building blocks for styling HTML pages. Play around with the builtins, and create your own. Remember, you can add your own classes in the **smd.css** CSS file, and then reference them by building your own custom divs using the div factory **_dfactory**.

Table of Contents

Cover, Revision & Contact Sections

There are three (3) builtins that can be used in your document to add commonly used sections in script files. They are defined in *sys.imports/report.md*:

var.cover - To add a cover section
var.revision - To add a revision section
var.contact - To add a contact section

The details for each type of section are discussed below.

@var.cover

Cover Title Syntax

[var.cover(title="your title", author="author name", logline="logline or short description")]

[var.cover.inline(title="your title", author="author name", logline="logline or short description")]

Each attribute is optional, and they can appear in any order. Also note that the value of any parameter can be whatever you want. Just because it says "author", doesn't mean you have to put the author name there. You could instead write "Roses are Red", and that would be just fine...

Let's see how **[var.cover]** renders out:

[defaults.title]

[defaults.author]

[defaults.logline]

Interesting. Looks like it relies on several ***default*** values from the **smd** variable **var.default**. That gives us two options to proceed: Either set the defaults (which you might opt to do in your enviroment, giving them better default values for your own markdown projects), or you can pass them on the fly, using the syntax above. Let's try again, only this time, we'll use the parameters from above:

your title

author name

logline or short description

That's better! So that's pretty much all there is to **var.cover**. It's used in a few of the samples provided in the docs, and you might find it useful for your own projects, along with its companions, **var.revision** and **var.contact**. Let's take a look at them.

@var.revision

Revision Syntax

[var.revision(v="1.0")]

[var.revision.plain(v="1.0")]

[var.revision.inline(v="1.0")]

[var.revision.inline_plain(v="1.0")]

Specify the revision number of your document within quotes. The default rendering of the var.revision variable is to include a timestamp at the end of the string. You can request a plain revision string using the plain attribute e.g. [var.revision.plain]. In our example below, we are going to use the plain version, because this file is included as part of the unittests, and timestamping would cause the test to fail every time. :)

Let's see how **[var.revision.plain]** renders out:

Revision: *[defaults.revision]*

And once again, we see that the default revision if not specified, looks to **var.default** for the **revision** attribute. Let's go ahead and specify the revision in the markdown using the following syntax: **[var.revision.plain(v="1.4.2")]**

Revision: *1.4.2*

Now that you've seen both **var.cover** and **var.revision**, you can probably guess how **var.contact** works just by reviewing the syntax. Let's take a look!

@var.contact

Contact Syntax

[var.contact(cn="name" ph="phone" em="email" c1="copyright line 1" c2="copyright line 2" c3="copyright line 3")]

[var.contact.inline(cn="name" ph="phone" em="email" c1="copyright line 1" c2="copyright line 2" c3="copyright line 3")]

Each element is optional, and the elements can appear in any order. By default, the system looks in the *var.defaults* variable for the definitions of *cn, ph, em, c1, c2 & c3*.

For this example, let's look at an alternative of editing defaults.md to specify the defaults for a markdown document, since it's likely you would vary these on a per-project basis. As it turns out, you can conveniently set them using a single call:

```
@set _id="defaults" \
    cn="Ken Lowrie" \
    ph="512-555-1234" \
    em="me@mycompany.com" \
    c1="Copyright © 2020 My Company, LLC." \
    c2="All Rights Reserved." \
    c3="www.mydomain.com"
```

So, I'm going to do that now, and then we'll write the **[var.contact]** markdown and see what we get.

Copyright © 2020 My Company, LLC.
All Rights Reserved.
www.mydomain.com

Ken Lowrie
512-555-1234
me@mycompany.com

Pretty cool, huh? Of course you would likely create **@link** variables for the email address and the website so you could make them hyperlinks, but I'm going to leave that up to you to do.

So that's it for these special section divs. Let's move on!

Table of Contents

Advanced Topics: @raw and more

We've been using **@raw** (which can also be written as **@@**) throughout this documentation, and when and if you ever peer into the builtins that come with **smd**, you'll see it used quite often. Essentially, beginning a line with **@raw** will suppress any **@wrap** formatting that is currently in effect, making it so you can control exactly what the parser will emit.

There is one thing to keep in mind about this if markdown is written after the **@raw** qualifier: since the markdown could effectively change the line to something else completely, say if a line is pushed onto the input stream, you may not get the results you were expecting!

@break and @exit

In the precursor to **smd**, a formatter called **avscript**, **@break** and **@exit** were used quite heavily to control the formatting, primarily as a means to clear the floats in effect when formatting **AV script**.

In **smd**, however, they are only used sparingly, but as you might have guessed, it will be in **AV Script** markdown documents. In order to avoid unnecessary blank space between elements, the **avs** macros do not automatically emit a block element that will clear the floats, since in most cases, an **AV** script will simply begin a new shot which avoids the issue. However, if you have a need to insert other information between shots, then you'll likely have to use **@break**/**@exit** in order to forcibly clear the floats.

So that pretty much sums up the advanced section. Apparently there aren't too many advanced topics after all... Hopefully it was helpful, if not, ask questions, and I'll clarify. Or better yet, improve the docs, and submit a pull request. :)

Table of Contents

Predefined classes

When you examine the default CSS provided with **smd** (located in **css/smd.css**), you will see quite a few predefined classes that have been provided to style the various types of documents you can create with **smd**. Of course you can add more, and/or change the existing ones to your hearts content, and get things styled the way you want them.

In many cases, these different styles are referenced via the **class=** attribute on HTML variables. But there's also another way you access them in your inline markdown by using a special span syntax: e.g.: This text will be RED. There are two ways you can use this:

1. Add it to the beginning of any line in your markdown
2. Add it to the start of an attribute value

Let's have a look at the syntax.

> Span Syntax a.k.a. Variable Decorators
>
> **@var variable="{:.class}value"**
>
> **So, if you declared this: @var mynewvar="{:.bigandbold.red}My new big bold value", and then write [mynewvar], you'd get this:**
>
> **My new big bold value**

There are a number of predefined classes in the primary CSS file that can be used to quickly style your AV scripts. You can add others as required, and decorate your elements as needed. Here are a few of them, used outside the AV DIV, and then again inside an AV DIV.

> Predefined classes
> **{:.note} -- This is a note.**
> **{:.question} -- This is a question.**
> **{:.vo} -- This is a VO note**
> **{:.important} -- This is important.**
> **{:.greyout} -- This is grey text on grey background.**

Here they are used outside an AV Section.

This is a note.

This is a question.

This is a VO note

THIS IS IMPORTANT.

This is grey text on grey background.

Here they are again, used inside an AV DIV section

CU: Predefined Classes used inside AV section

This is a note.

This is a question.

This is a VO note

THIS IS IMPORTANT.

This is grey text on grey background.

Here are a few more of the predefined classes available, and remember, you can tailor these or add more as required for your particular purpose.

More predefined classes

{:.pbb}-- Page Break Before (when printing).

{:.pba}-- Page Break After (when printing).

{:.red}-- To color text red.

{:.green}-- To color text green.

{:.blue}-- To color text blue.

{:.center}-- To center text.

{:.left}-- To left align text.

{:.right}-- To right align text.

{:.bigandbold}-- To increase text size and make it bold.

{:.box}-- To put a box around it.

{:.dashed}-- To put a dashed line around it.

{:.greybg}-- To make the background grey.

{:.ignore}-- So it won't display in the output.

You can stack multiple classes by simply stringing them together. For example, on the next line, I'll write **{:.greybg.bigandbold.blue}This is a big and bold blue note on a grey background.**

This is a big and bold blue note on a grey background.

If you have text you want included in the HTML document, but do not want it rendered by the browser, use the **{:.ignore}** class prefix. For example, on the next line, we'll write *{:.ignore}You won't see this.*

When you examine the HTML, you'll see the prior text wrapped in **<p>** tags, inside ** ** markup. However, it will not be rendered by the browser, unless you modify the CSS rule for the ignore class.

Lines that begin with a double forward slash **//** are treated as comments, and are discarded by **smd**. They will not appear in the HTML at all. As another example, we'll write *//This will not appear in the HTML* on the next line.

Once again, if you examine the HTML output, you will not see the previous line in the output.

Table of Contents

Audio/Visual (AV) Script Formatting

In the prior incarnation of **smd**, the primary purpose of the application was to create Audio/Visual (AV) style scripts from plain text markdown files. In fact, the name of the prior version was **AVScript**, and it consisted of two Python command line utilities: **avscript.py** and **mkavscript.py**. You may recall that this version relied on the **BBEdit** text editor in order to provide the WYSIWYG preview support while building your markdown documents.

As it morphed into the current version where no reliance on a specific text editor was required, it also shed some of the syntax and semantics of the prior version which limited it to being useful only for creating AV Script documents. This led the way to it being useful for creating many other types of content, from simple web pages to dynamic pages and new monitoring features including endpoints for direct HTTP service.

But the roots of being useful for generating A/V Scripts remains through the use of specialized built-ins provided as part of the distribution package. This support is contained within the **[sys.imports]/avs** directory.

The AVS builtin library

The **AVS** builtins provide everything you need to create A/V-style scripts, from the simple, mostly text based scripts, to complex shot-breakdown documents that help you communicate your vision to everyone, be it your Executive Producers or the all important Crew Members!

Although there are a number of markdown files in the **avs** directory, the two you will use primarily are:

[sys.imports]/avs/avshot.md - for simple A/V scripts
[sys.imports]/avs/avs.md - for access to the shot breakdown builtins.

Let's see a quick example now. Consider the following markdown:

Generate a simple AV shot

```
@import "[sys.imports]/avs/avshot.md"

[avshot.shot_with_desc(_s="WS:Sunrise", _d="\
    There is just something about a sunrise that gets the blood flowing...\
    And here is some additional narration.[bb]\
    and here are some additional shot notes.\
")]
```

This is what will be rendered by the browser:

WS:Sunrise

There is just something about a sunrise that gets the blood flowing... And here is some additional narration.

and here are some additional shot notes.

And here is the actual HTML code that the parser emits:

Raw HTML Output from prior avshot.shot_with_desc markdown

```
<div class="av"><ul>
<li>WS:Sunrise</li>
</ul>
<p>There is just something about a sunrise that gets the blood flowing...    And here
is some additional narration.<br><br>    and here are some additional shot notes.</p>
</div>
```

While this is useful for simple A/V Shot generation, many times it might be easier and clearer to write things in a more free-from style. As it turns out, **avshot** has builtin methods for that as well. Take a look at this:

Generate a simple AV shot using the section methods

```
[avshot.visual]
    There is nothing like waking up to the smell of coffee percolating in the outdoors.
[avshot.audio]
    *After we fade into the early morning wide-shot of the camp-site, we will cut to
this close-up, making sure the client's product logo is visible in the shot.*
[avshot.end]
```

And here is how the browser will render this markdown:

CU:Coffee pot heating on wire rack of fire pit

```
There is nothing like waking up to the smell of
coffee percolating in the outdoors.

After we fade into the early morning wide-shot
of the camp-site, we will cut to this close-up,
making sure the client's product logo is
visible in the shot.
```

In both the **.visual** and **.audio** sections, you can have as much information as required, just keep writing, even starting new regular paragraphs. You can insert any markdown as well, to aid your reader in understanding what you want. When you're done, close out the one shot, and start another. You know, lather, rinse, repeat...

In the upcoming Samples chapters, we will cover **avshot** in much more detail as well as the entire set of **avs** builtins... For now, this hopefully provided a quick introduction to what's in store!

Guide to the Sample Documents

Before we leave this chapter, here's a summary of the examples provided in the upcoming chapters. This should point you to which specific sample(s) you may want to review to jump start your project, although it's probably best that you read them in the order listed, since each one builds upon concepts covered in a predecessor.

Sample 1 - Creating a Project Proposal
Sample 2 - How to use the *avshot* builtin
Sample 3 - Real World Example - AAT A/V script using *avshot*
Sample 4 - A Music Video Treatment
Sample 5 - Using the Advanced Image and Shot Support

Table of Contents

Debugging your markdown files

Okay, most of you will never need any of the things we are going to discuss here. Why would you? Your markdown will be flawless and perfect the first time. That's good for you, unfortunately for me, that's not the case. Because of that, two special keywords were added to assist with the markdown debugging process: **@debug** and **@dump**. This chapter is going to go over both of them in detail, and show how they can be used to assist in the debugging process.

The @debug keyword

We will start with the **@debug** keyword, as this is useful for getting debug messages written into the output stream, so that you can see what is happening in real-time.

> @debug Keyword Syntax
>
> **@debug [*tag*="*regex*" [*tag*="*regex*" [...]]]**
>
> ***tag* is one of <on | off | toggle | enabled | tags>**
> ***regex* is any Python-compliant regular expression**

Okay, to start, you can see that the parameters to **@debug** are *all* optional. So, if we type *@debug*, this is what happens:

@debug keyword

```
// Toggle everything on
@debug
Toggling Debug Mode

Method(toggle): _SYSTEM is now enabled
Method(toggle): bookmarks is now enabled
Method(toggle): cache is now enabled
Method(toggle): cache.import is now enabled
Method(toggle): markdown is now enabled
Method(toggle): ns is now enabled
Method(toggle): ns.add is now enabled
Method(toggle): ns.code is now enabled
Method(toggle): ns.html is now enabled
Method(toggle): ns.image is now enabled
Method(toggle): ns.link is now enabled
Method(toggle): ns.var is now enabled
Method(toggle): smd is now enabled
Method(toggle): smd.line is now enabled
Method(toggle): smd.raw is now enabled
Method(toggle): stdinput is now enabled
Method(toggle): utility is now enabled

// Toggle everything back off
@debug
Toggling Debug Mode

Method(toggle): _SYSTEM is now disabled
Method(toggle): bookmarks is now disabled
Method(toggle): cache is now disabled
Method(toggle): cache.import is now disabled
Method(toggle): markdown is now disabled
Method(toggle): ns is now disabled
Method(toggle): ns.add is now disabled
Method(toggle): ns.code is now disabled
Method(toggle): ns.html is now disabled
Method(toggle): ns.image is now disabled
Method(toggle): ns.link is now disabled
Method(toggle): ns.var is now disabled
Method(toggle): smd is now disabled
Method(toggle): smd.line is now disabled
Method(toggle): smd.raw is now disabled
Method(toggle): stdinput is now disabled
Method(toggle): utility is now disabled
```

Okay, awesome, you can see that **@debug** on it's own, simply toggles the state of each registered debug handler. In our case, everything was off, and so the first time **@debug** was used, it toggled everything on. Then, the second time, everything was toggled back off.

However, if one or more of the registered handlers was **on**, then the first time we issued the **@debug** it would have toggled those off, and everything else on! It's a toggle, so that should be self explanatory. If you aren't sure, open up an **smd** session and try it, and see how it works.

> NOTE: To be honest, enabling *all* of the debug handlers at once will most likely not be very useful for you. There are hundreds of debug messages that will be printed depending on what your markdown contains, and it'll be very tricky to follow along. In most cases, you will choose one or two of the handlers to enable, and then use the output from them to decode what is going on.

The @debug parameters

The parameters to **@debug** are:

Parameter	Description
on="regex"	turn the registered debug handler(s) on for all that match the **regex** string.
off="regex"	turn the registered debug handler(s) off for all that match the **regex** string.
toggle="regex"	toggle the state of the registered debug handler(s) for all that match the **regex** string.
enabled="regex"	dumps the current registered debug handler(s) and their state, without changing it(s) off for all that match the **regex** string.
tags="regex"	dumps the registered debug handler(s) names; if *name emphasized* it is disabled, or **name bold** it is enabled. Does this off for all that match the **regex** string.

Next, let's talk about the **regex** string that can be specified with each of these options. Regular Expressions are a convenient way of matching multiple things using a specific language if you will. I'm not going to get into describing that, but there are plenty of sites on the Internet that will cover it quite thoroughly. The important thing to remember is that the **regex *must*** be a Python-compliant regular expression in order for it to work as expected. Let's look at a few examples of how you can use it to match multiple debug handlers in **smd**.

The various modules and subsystems in **smd** can register a debug handler using a name. If you look at the list of handlers above, you'll see the names within the output strings. For example, **ns**, **ns.add** and **ns.code** are all registered debug handler names. Let's say you want to toggle the state of all handlers that begin with *ns*. You could do that by typing *@debug toggle="ns"* or *@debug toggle="ns**"*. Just be sure to put the **@debug** statement on a line by itself, because otherwise it will be treated as inline text.

But what if you wanted to toggle just the **ns** handler, but not all of the other ones that begin with *ns*? You would do that like this: *@debug toggle="**ns$***. The **$** signals the end of the string, or name in this current context. See how that works so far?

Now, let's say we wanted to toggle the state of the *utility* and *ns* debug handlers. Do we need to **@debug** statements for that? Nope. Use this: *@debug toggle="**utility | ns$***" statement. NOTE: I added spaces around the pipe **|** so it would be easier to read. Do **not** do that for real, or you won't get the results you expect.

Keep in mind that this method of using regular expressions for the parameter strings works the same in **@debug** and **@dump**, so it's definitely worth getting more familiar with.

This wraps up the section on the **@debug** statement. Let's move along into **@dump** now, another useful debugging trick when writing your own builtins and macros.

The @dump keyword

The **@dump** keyword is used to dump the current declarations of variables. As you will see, you can dump a single variable, or multiple variables, in one or more namespaces. Let's first take a look at the syntax:

> @dump Keyword Syntax
>
> **@dump [*tag*="*regex*" [*tag*="*regex*" [...]]]**
>
> ***tag* is one of <var | html | link | image | code | sysdef | tracked | help>**
> ***regex* is any Python-compliant regular expression**

Okay, to start, you can see that, like **@debug**, the parameters to **@dump** are *all* optional. So, if we type *@dump*, this is what happens:

```
@dump keyword

// Dump all the sysdef's, tracked files and variables in all namespaces
@dump
----------------------------------------
System Defaults: .*
----------------------------------------
    Current system defaults
----------------------------------------
Files seen during parsing: .*
----------------------------------------
    Files seen during parsing
----------------------------------------
NAMESPACE: var
----------------------------------------
    Current @var variables
----------------------------------------
NAMESPACE: link
----------------------------------------
    Current @link variables
----------------------------------------
NAMESPACE: html
----------------------------------------
    Current @html variables
----------------------------------------
NAMESPACE: image
----------------------------------------
    Current @image variables
----------------------------------------
NAMESPACE: code
----------------------------------------
    Current @code variables
```

Because there are hundreds of variables defined in even the simplist **smd** markdown session, I've chosen to show nothing in the list above. If you would like to see for yourself, then start **smd** interactively (type **[smd.il] -nd** in your terminal window), and then **@dump**. Let's review all of tag options for **@dump**.

The @dump parameters

The parameters to **@dump** are:

Parameter	Description
sysdef="regex"	Dump the system defaults whose name matches the **regex** string
tracked="regex"	Dump all the files that match the **regex** string
var="regex"	Dump all **@var** variables whose name matches the **regex** string
link="regex"	Dump all **@link** variables whose name matches the **regex** string
html="regex"	Dump all **@html** variables whose name matches the **regex** string
image="regex"	Dump all **@image** variables whose name matches the **regex** string
code="regex"	Dump all **@code** variables whose name matches the **regex** string
help="True|False"	The allowable values for this parameter are either **True** or **False**. This controls whether or not **@dump** will print the **_help** attribute, if present

Anyway, note that like **@debug**, **@dump** also allows regular expressions to be used in the parameter for any of the tags. Given that, consider this example:

```
@dump keyword

// Dump b$ and bb from @var namespace
@dump var="b$|bb"
----------------------------------------
NAMESPACE: var
----------------------------------------
b=
    _format=

bb=
    _format={{b}}{{b}}
```

Unlike **@debug**, you can also specify multiple tags with **@dump**. Here's an example of that:

```
@dump keyword - multiple tags

// Dump b$ and bb from @var namespace and table from @html namespace
@dump var="b$|bb" html="table"
----------------------------------------
NAMESPACE: var
----------------------------------------
b=
    _format=

bb=
    _format={{b}}{{b}}

----------------------------------------
NAMESPACE: html
----------------------------------------
table=
    _format=<{{self._tag}}{{self._public_attrs_}}>
    _tag=table

table_2=
    _format=<{{self._tag}}{{self._public_attrs_}}>
    _tag=table
    style=margin-left:auto;margin-right:auto
```

Go ahead and try dumping different variables from the various namespaces to get familiar with using the regular expressions to choose only those you are interested in. This will be very helpful to you when you start writing your own markdown documents, you know, if you end up having an issue in your markdown that you need to debug. :)

There is also a builtin macro in the **@code** namespace called **code.dump** that can be used to display the help string for a given variable. It is used quite a bit in the user documentation, although unless you are writing documentation for your own extensions to **smd**, it isn't likelly something you will get much use out of.

One final note is the special tag **help**. First, it does not take a regular expression like the other tags. Instead, it takes either **True** or **False**. By default, it is set to False, which tells it that it should **not** dump the **_help** attribute on a variable if one exists. If you set it to True, however, and a given variable has the **_help** attribute, then it will dump that too. It is set to False by default because in most cases, you don't want to see the help string, or if you do, it makes much more sense to display it with **varname.?** or **varname.??**, so it will be formatted in a way that you can read it!

A few common situations you might encounter

In this section, we will document how to debug common issues you may encounter while writing markdown for your project. Currently I have only identified a couple that are common things that happen to me, so I will address them now. As more things are identified going forward, I will add them here.

Markdown nesting error

Consider the following markdown:

Markdown nested too deeply error

```
// Declare two variables
@var x="{{y}}"
@var y="{{x}}"
[x]
// Boom
smd.core.exception.NestingError: Markdown().markdown--Expansion nested too deeply.
Enable @debug on="markdown"
```

Did your computer make a small explosive sound? LoL. No? Okay, bad joke. Anyhow, it's likely obvious what has happened here: variable **x** says it should evaluate to **y** and **y** says it should evaluate to **x**. So this battle will continue 25 times, and then the markdown code will give up. The good news is that it pretty much tells you how to debug the issue at the end: *Enable @debug on="markdown"*. So, let's change the code as follows:

Markdown nested too deeply error - enable debug

```
// Declare two variables
@var x="{{y}}"
@var y="{{x}}"
@debug on="markdown"
[x]
// Boom
smd.core.exception.NestingError: Markdown().markdown--Expansion nested too deeply.
Enable @debug on="markdown"
```

Looks the same, but if you scroll up a bit to the top of the exception stack, you will notice the following set of messages:

Markdown recursion dump with debug enabled

```
markdown recursion dump:
    Item: 26: {{y}}
    Item: 25: [x]
    Item: 24: [y]
    Item: 23: [x]
    Item: 22: [y]
    Item: 21: [x]
    Item: 20: [y]
    Item: 19: [x]
    Item: 18: [y]
    Item: 17: [x]
    Item: 16: [y]
    Item: 15: [x]
    Item: 14: [y]
    Item: 13: [x]
    Item: 12: [y]
    Item: 11: [x]
    Item: 10: [y]
    Item: 9: [x]
    Item: 8: [y]
    Item: 7: [x]
    Item: 6: [y]
    Item: 5: [x]
    Item: 4: [y]
    Item: 3: [x]
    Item: 2: [y]
    Item: 1: [x]
Traceback (most recent call last):
```

You can see from the recursion dump that **x** refers to **y**, then **y** refers back to **x**, and so on and so forth until it dies. Now this is a very simplistic example, but the good news is that in most cases, it will be easy to figure out if you toggle on the **markdown** debug handler, because it will show you each line that it tried to process, which will identify which variables and/or attributes are involved in the showdown.

Next up is using **ismd** to debug the raw HTML...

Debugging the raw HTML

Another useful thing to know is how to debug the raw HTML instead of looking at the rendered output, which at times might depend on the browser you are using to display it. Usually though, this method will help you figure out where your HTML markup is messing up, so it's definitely a good trick to know.

If you skipped the chapter on **ismd**, now might be a good time to review it.

Okay, so that wraps the chapter on debugging. And it also marks the end of the primary documentation for **smd**. In the remaining chapters, we will look at several examples that illustrate how to accomplish some real-world applications for **smd**.

Table of Contents

Sample Projects built with smd

We will now look at several different examples created with **smd** that show some of what you can do with this app. Most of these samples were taken from actual projects I did, with just some minor cleanup to hopefully give you ideas and/or a starting point for creating your own content.

I recommend reading them in order, since each builds upon concepts covered in one or more of the prior examples.

Table of Contents

Creating a Proposal using smd

In this example, we will use **smd** to create a project proposal document. The objective here is simply to show that you can format content in all sorts of different ways using **smd**, not that this example is necessarily the best way to create a project proposal. A regular old word processor would likely be a better option for this task...

Unknown Documentary Project

The purpose of this document is to provide an estimate for the production and post production costs associated with producing the documentary tentatively titled *Just Shut It Off*."

Production Estimate

This estimate *does NOT* include costs for:

1. Stock footage (photos, video, audio, graphix, score, etc.) Producer will be responsible for purchasing clips to be used in the film. In the most common case, we will locate clips and provide purchasing details (costs, source, licensing info). The Producer will either purchase them directly and provide the footage to the editor, or we will send an invoice, and once it's paid in full, we will purchase the clips. Keep in mind that most sites have a **no refunds, no returns policy**. So once they are purchased, you own them.
2. Shooting new footage. Any footage that we need to shoot will be invoiced separately. As is the case with stock footage, an invoice will be issued for footage we are to shoot, and it must be paid in advance before we will shoot it.
3. Creation of custom graphics or animations.
4. Foley creation
5. Voiceover
6. DVD and/or Blu-ray Mastering

This estimate does include costs for:

1. Color correction
2. Color grading

3. If we already have (own) stock footage that can be used in the film, we are happy to use it. No additional costs will be associated with using the footage that we already own, but any copyrights on said footage remain with us.[dblbrk]
4. Audio mixing (including FX, VO, score)[dblbrk]
5. Music score (provided you choose a title from our existing royalty free library)[dblbrk]
6. Mastering to high resolution H.264 digital media. Most festivals accept submissions in this format nowadays; also, this format can be remastered to either DVD or Blu-ray.

Cost options

Normally, there are three (3) billing options to choose from when hiring [Cloudy Logic Studios] to edit your film:

1. **Per runtime minute of film**. $525/minute. Minimum charge: $2,625 (5 minute minimum, regardless if final film is less than 5 minutes long).

> Example: If the **TRT** (total run time) of the finished film is eight (8) minutes, the total cost would be $4,200. Partial minutes will be billed in fifteen (15) second increments, i.e. ~$131 per one quarter (1/4) minute.
>
> **Considerations when choosing TRT option**
>
> This rate applies to editing normal footage (video, photos, audio). As an example, the demo film that we produced would have all fit under this rate, with one exception (green screen footage that was keyed for ending). However, if we are given footage that requires *specialized attention*, a different rate will apply on a per clip basis.
>
> Some examples that require specialized attention are: Footage that is shot on green screen and must be keyed. Footage that is shaky and must be stabilized. Footage shot with incorrect color temperature. Any media (video/photos) that requires rotoscoping. Audio that must be cleaned up.

2. **Per hour**. Editing for this project will be billed at $60/hr. Since we bill based upon hours spent, there are no exceptions to the type of footage being edited, as is the case with the ***"Per runtime minute of film"*** estimate above.

3. **Per project**. An estimate for the entire project requires a completed script in AV format, with all details ironed out, including exact clips to be used, footage to be shot, etc. Given those requirements, it is not feasible to provide this option for consideration at this time.

NOTE: *For this project, only billing options 1 and 2 will be offered.*

Retainer

If you decide to hire [Cloudy Logic Studios] for this project, no retainer will be required.

Invoicing

Once the project is underway, we will invoice periodically as various milestones are met. These milestones will be set and agreed upon by both parties prior to commencement. They will be something like this:

1. Rough cut - *might be in stages*
2. Picture lock - *no changes to timeline after this*
3. Color grade complete - *possible after audio mixing*
4. Audio mixing complete - *possible before color grade*
5. Film completion

> If the project is being billed per runtime minute, the milestone invoices will be billed in fractional estimates of the minimum runtime up until picture lock, at which time remaining invoices will be billed based on the actual total runtime.
>
> If the project is being billed per hour, then invoices at each milestone will reflect the total based on hours invested to achieve the milestone.
>
> Usually, work on the next milestone will not commence until the invoice for the prior milestone has been paid.
>
> Finally, keep in mind that final media will not be released until all invoices are paid in full.

Summary

If you would like to move forward with the project, here are the **next steps**:

1. Choose a billing option
2. Iron out milestones
3. Acceptance of Terms - return signed contract

Thanks again for choosing [Cloudy Logic Studios] for your production needs. We look forward to working with you on this project.

Please let me@mydomain.com know if you have any questions regarding this estimate, or if you need any clarifications and/or changes.

Warmest regards,

Firstname LastName
Producer
Production Company
512.867.5309
__DELETEME__mailto:you@youremailaddress.com

Terms and Signatures

If you agree to the terms and conditions outlined in this proposal, select a billing option, initial each page, sign/print/date below, and return the entire contract to [info@cloudylogic.com].

Choose your Billing Option:

{ } Per runtime minute.

{ } Per hour.

Upon final acceptance of video, an invoice will be generated outlining the final amount due. Payment is due within 30 days of final acceptance.

> **NOTE: Final media is *not delivered* until all outstanding invoices have been paid in full.**

Signatures

Customer Name:

Signature: _____

Printed: _____

Date: _____

Cloudy Logic Studios

Signature: _____

Printed: _____

Date: _____

Table of Contents

Sample A/V Script Document

In this example, I will show you how to use **smd**'s **avshot** builtin to create an Audio/Visual (A/V) Script. AV Scripts are simple two column scripts that describe the visuals (i.e. shots) on the left, and the audio (i.e. narration or voiceover), on the right. **smd** has several builtins that assist with creating this type of script, but in this example, we are going to focus on **avshot**.

Start by including **@import [sys.imports]/avs/avshot.md**, and you will get everything you need to use **avshot**.

A/V Script Series

Title of Script

Script Author

Script summary goes here and can be as long as needed. Let it wrap around if you have softwrap, or just go on forever.

Revision: *1a*

Contact Name
Phone
me

Notes to Reviewers

Please send me any and all feedback, preferably by marking up the PDF using embedded comments. If you edit the PDF text, do so inline using comment boxes, or if you edit the text directly, change the color and/or font size so I can easily find it. **additions are marked like this** ~~deletions are marked like this~~

Film Pitch for Client Name

Client Name:

Please review the proposed script for your upcoming project. We believe that it will show the type of production value that we will bring to your project, focusing primarily on ...

A/V Script for **Client Name** project entitled "Best-est Social Media Campaign for Acme Widgets, LLC"

[avshot.visual] begins the definition of a shot. You can list one or more shots in this section, and include formatting and other markdown. When you are done listing the shots, you will use either **[avshot.audio]** or **[avshot.noaudio]** to either begin listing the narration or voiceover, or simply close out the shot.

In this example, we will open with **avshot.visual**, write out shot description, and the close with **avshot.noaudio**, so there won't be anything listed in the right column.

CU: Staring out the window

Not too exciting just yet, but we'll get there. The more common case is to use **avshot.visual**, then describe the shot, use **avshot.audio**, describe the narration or voiceover, then close with **avshot.end**. Let's do that now, and see how it looks.

WS: Busy freeway rush hour traffic at a standstill

```
In this example, after the preceding shot
declaration, we used [avshot.audio] to close
out the visuals section, and begin the
narrative section. You can have as much
narration and notes as you need. When you're
done, close out the shot with [avshot.end].

There are a bunch of useful shot notation
mnemonics defined in the
sys.imports/avs/shotacro.md file, so you'll
definitely want to review that to get an idea
of those goodies!
```

Mixing AV Script builtins and other markdown

At any time in your AV Script, you can switch back and forth between shots and normal markdown. All you do is place the normal markdown outside the **avshot** builtins. It's that simple. When you're ready, you just start adding more shots!

PART 1: Description for part 1
PART 1A
PART 1B

```
Link to Article

notes for the first part.

and some other notes.

And a few more.
```

WS:Couple watching TV

CU:Couple looking concerned

The narrative for the shots on the left would go here.

MS:Paranoid guy looking thru blinds
If you indent a line following a shot, then that text becomes part of the prior shot, allowing you to put a little more description if you need it.

If you want to put inline notes, questions or use any of the Simple Divs (*note, question, vo, important, greyout*), you need to use the ***no-div** version. The **no-div** version uses the suffix **_nd** on the key attribute names. For example, if I use the default and write:

[note(t="this is my note")], this is what I will see:

this is my note

See how the box jams up against the left margin of the right column? It also clears the floats, which will mess up the formatting for the next element in the audio section... ugh!

This time, i'll write **note.nd**, to use the no-div variant, and let's see how that looks:

this is my note

Much better! Just remember to use the **nd** versions of any of the simple divs when you are using them inside the **avshot** builtins!

ECU:Perspective looking thru peephole.

CU:Locking door

More narrative here that goes with the shot on the left...

PART 2: The middle section

This is a description for this section

Link to Article <-- That should have been turned into a link

MS/CU:Clips of people angry
You can add more information about a shot by simply typing more info. Each line break in your text begins a new line in the section.
This will be on a new line,
And so will this.

The narration for the Clips of people angry would be here...

New lines over here do the same thing. Each time you insert a line in your markdown, it will start a new paragraph.

Like this...

And this!

WS:violence
This adds additional visual info on your shot.
You can also prefix each new line with a different class for formatting...

CGI Websites and blogs

CGI Use PIP to fill the screen

Words, words, words, blah, blah, blah

Video Segment Part 2 is next ...

CGI:

```
CGI Text is here
```

FTB: *fin*

```
The End.
```

WS:Stock clips of TV shows that ...

```
So many programs on television today ...
We should practice tolerance and ...
It's also crucial for people to ... and ....
We are past due for ...
```

TS1

Specialized avshot builtins

There are two short versions of avshot that can be used for quick shot only cases or shot with short description. They are:

[avshot.shot_only] and **[avshot.shot_with_desc]**

They are just simplified versions you can use to quickly generate a shot only or a shot with short description.

```
Simplified avshot methods
// If all you have is a shot (visual), use the shot_only method
[avshot.shot_only(_s="shot info")]
// If you have a shot (visual) and audio (narration,...), use the shot_with_desc method
[avshot.shot_with_desc(_s="shot info" _d="shot description")]
// An example of each method
[var.avshot.shot_only(_s="[cu] Hands holding phone")]
[var.avshot.shot_with_desc(_s="[ecu] text message on phone" _d="help[bb]me")]
```

And here is how the two **avshot** methods above will render:

CU: Hands holding phone

ECU: text message on phone

```
help
me
```

Sometimes all you need are these shorthand methods to describe your shots, but in other cases the separate section-based methods (**avshot.visual**, **avshot.audio**, **avshot.end** are better. The good news, you can mix and match as you like. Before we leave this example, let's review the builtin hep string for **avshot**.

For reference, here is the help for avshot:

```
avshot

  Used to emit shots in A/V Script-style documents.

Common Parameters
  _s - shot info. e.g. WS: Lake with people swimming near shore
  _d - shot description/narrative for opening wide shot

Methods
  visual - Open a shot declaration, ready for shot(s)
  audio - Close shot section and transition to audio; ready for shot description
  end - Close shot declaration
  noaudio - Close shot declaration when no shot description will be set
  shot_only(_s) - Create a complete shot declaration; shot(s) specified in _s
  shot_with_desc(_s,_d) - Create complete shot declaration; shot(s) _s, description
_d.
```

Okay, that wraps up this chapter on creating A/V scripts with **smd**.

Table of Contents

Sample A/V Script using *avshot* builtin

In this example, we will review an actual AV script from a project I did a few years back. This version was originally written using the old **avscript** version of the software, before it supported embedding images, so it's entirely done using text to describe the visuals and narration. It was converted to conform to the new **smd** builtins for inclusion in the user guide.

AAT 4th Gen Video Series

Using iOS Devices with the AAT

Ken Lowrie

This video shows how to use iOS devices to connect to the AAT's internal web server to control and manage the AAT. This video demonstration will use an iPhone and iOS 8.3.

Revision: *1a*

Ken Lowrie
407-555-1000
me

Notes to Reviewers

Please send feedback by marking up the PDF using embedded comments or notes. If you edit the PDF text directly, be sure to change your font color so that I can easily find the changes.

Notes that begin with "VOICEOVER NOTE" or

> are inside a box like this

can be ignored; they are for the voiceover guy.

Additions **are now marked like this** and deletions ~~are now marked like this~~

Thanks in advance for reviewing and commenting on the script!

Script begins here

CGI: Using iOS devices with the AAT
RECAP: Show clips from finished video timelines that recap
CGI:iPhone or iPad Devices can operate the AAT
CGI:iOS v7.1.2 or later required

Welcome to *Using iOS devices with the AAT*.

In this video, we will show you how to control your AAT using an iOS device. It's simple and efficient, using tools that you are already familiar with: Your iOS Device and a Web Browser! Here's a quick preview:

First, power on the AAT and wait for AZM to begin flashing

Then, connect to the AAT hotspot with your iOS device

Finally, open aat.sunsight.com in your browser, or just touch the AAT home page icon

And that's all there is to it!

Once you've connected, you can do things like print site reports, email those reports to your customers, and more.

Using iOS devices to connect to the AAT's internal web server allows you to access the User Interface or UI, which in turn, enables you to fully manage and/or control the AAT.

Additional setup and shortcuts are covered
Adobe Acrobat Reader for viewing PDF reports
How to create a PDF Report
Sending PDF reports to your customers

In the remainder of this video, we will cover a one-time setup for your iOS device that enables you to efficiently control your AAT.

We will cover how to set up a home screen shortcut, to make accessing the AAT quick and easy, and we'll discuss Adobe's Acrobat Reader application for viewing PDF reports on your device.

After a quick introduction on generating PDF reports, we'll show you how to get the PDF reports to your customers.

Requirements and Limitations
The AAT should be powered on and ready for clients to attach
Refer to ***Connecting to the AAT with the Android Phone*** for details
Only one (1) user can be connected at a time

This video assumes the AAT is powered on and ready for clients to attach to its internal hotspot. View the video entitled ***Connecting to the AAT with the Android Phone*** to learn about the AAT startup process.

Keep in mind that only one user can control the AAT through its onboard website at a time. Multiple users connected at the same time is not supported.

SIDEBAR: Connecting to the AAT's Wireless Network
IMAGE: Picture of an iPhone or an iPad
Connecting to the network will look different depending on your hardware and software
Process is *identical* to connecting to *any* wireless hotspot

You'll need an iOS device that is running version 7.1.2 or later in order to use it to control the AAT.

In the demonstration, we will be using an iPhone 6+ running iOS 8.3 with the native Safari browser. You can use other iOS browsers, such as Google Chrome, if you prefer.

Refer to your device documentation for additional help

Depending on the hardware and software version of your iOS device, along with the browser you are using, this process may look slightly different than what is shown in the video.

The good news is that connecting to the AAT's wireless network is ***identical*** to connecting to any wireless hotspot, so if you are having trouble figuring out how to do this, refer to the documentation that came with your iOS device for additional assistance.

Let's go ahead and watch the demonstration of using an iOS device to control the AAT. The first step is to connect to the AAT's Wireless Network or hotspot.

SCAST: Show AAT WiFi Network Connect via iOS 8.3

To do this, on your iOS device, choose SETTINGS, and then select Wi-Fi.

Ensure Wi-Fi is on, and then look for the AAT Wireless hotspot.

The AAT hotspot has a network name beginning with *AAT 90*, followed by a series of digits that represent the serial number of the unit. The serial number is on a printed decal affixed to the back of the tool.

Touch the AAT hotspot in the list to initiate the connection.

You can tell that you have successfully connected to the hotspot when the name appears with a checkmark next to it, *and* the WiFi icon is displayed in the top left status bar, as shown in the video.

Connecting to the AAT
SCAST: Show the iOS home screen display and dock, and touch the Safari icon

Now that we are connected to the AAT hotspot, we are ready to access the AAT User Interface. To do that, you need to launch Safari, so go ahead and find the *Safari* icon on your iOS Device, normally located on the dock, and touch it to launch the browser.

Launch the AAT User Interface
SCAST: Touch the new tab icon in Safari
SCAST: Show aat.sunsight.com and then Home page
SCAST: Show 192.168.0.50 and then Home page

```
GREGG: BEWARE!!!

Here is that AAT dot SUNSIGHT dot COM
again... :)
```

To start the User Interface, open an empty browser tab, and in the address field, enter the URL *aat.sunsight.com*, and then press *GO*. The AAT user interface home page will be displayed, and you're now ready to setup, manage profiles and print reports!

Alternatively, you can enter the IP address of the AAT, which is usually, 192.168.0.50. Enter it ***exactly*** as shown, with the periods separating the four groups of digits. Press the GO key and wait for the home page of the AAT user interface to display.

Optional Step - Saving a shortcut or bookmark
Save a shortcut on the home screen to quickly return to the AAT home page

To save time starting the AAT User Interface in the future, you can create a shortcut on your home screen, which you can then tap to go directly to the AAT home page.

SCAST: Show the sharing icon and save to home screen

To add a shortcut to the home screen, start Safari and launch the AAT User Interface by typing *aat.sunsight.com* in the address bar, and

SCAST: Show launching AAT UI via home screen icon

then touch the sharing icon (located at the bottom of the page on an iPhone, or at the top next to the address bar on an iPad).

Choose *Add to Home Screen*, and type in a name, such as *AAT*, press the *ADD button*, and the shortcut will be added to your homescreen.

From now on, after connecting to the AAT's WiFi network, you can simply touch the shortcut icon on the home screen to start the AAT User Interface, instead of launching the browser and typing in the URL or IP Address!

SCAST: Show the sharing icon and add bookmark icon
SCAST: Show launching AAT UI via browser bookmark

In addition to saving a shortcut on the home screen, you can create a bookmark by tapping the Bookmark icon, also available on the sharing submenu.

Enter the name you'd like to use, then touch *Save*.

To access the bookmark later, simply tap the bookmark icon from inside the browser, and then tap on the saved AAT Bookmark.

Adobe Acrobat
Allows viewing and managing PDF reports
Provides a means of saving PDF reports so they can be recalled later
This app is optional; it is not needed to view or email PDFs on iOS

You can install the free Adobe Acrobat application from the App store in order to view and manage AAT reports in PDF format on your iOS device.

PDF format is a very convenient way to send reports generated by the AAT to your customers, usually via email. In addition, the Adobe Acrobat app provides a convenient way to save reports and email them later.

Keep in mind that installing the Adobe Acrobat application is optional, and is not needed to view or email PDF reports on your iOS device.

SCAST: Show how to download/install Adobe Acrobat via the App store

To install Adobe Acrobat, start the *App Store* by touching the *App Store icon* on the home screen.

Press the *Search button* in the bottom bar, and then, in the search area, type *ADOBE ACROBAT* to locate the free download, and then install the application on your device.

> **NOTE:** Your device must be configured with an account in order to access the App Store. Refer to the documentation that came with your device for further information.

Generating Reports
Site Reports can be generated in Adobe PDF format
View reports before sharing to verify data is complete and accurate

Once you have completed your work using the AAT, you can generate site reports in Adobe PDF format to send to your customers.

SCAST: Show the site report being generated

To generate the PDF site report, touch the "Profiles, Captures, Reports" tab on the AAT Website, and then touch the PDF button next to the site you want to report on, or select "All PDF" to report on all sites.

The PDF report is generated and shown in the browser tab, at which point you can review the report, then when you're ready, press the SHARE

	icon, and choose your preferred method for sharing with your customer.
SCAST: Show opening the PDF in Adobe Reader app	If you have an optional PDF Viewer installed on your iOS device, such as the free Adobe Acrobat app, you can use the *Open in* feature of iOS to view your report and store it. Once you open a report in the Adobe Acrobat app, it is saved there, and can be reviewed and shared at a later time. This provides a convenient way to access your reports without needing to reconnect to the AAT, and regenerate them.
Sharing Reports To email reports, you must have an email account configured on your iOS device Refer to your iOS documentation or consult your IT department for assistance	In most cases, you'll want to email the generated PDF reports to your customers. In order to do this, an email account must be configured on your iOS device, and you must be connected to the Internet, either via a mobile data connection, or via a wireless connection with Internet connectivity. Refer to the iOS documentation if you need assistance with configuring an email account on your device, or consult with your company's IT staff.. **REMEMBER:** The AAT's internal wireless hotspot does *not* have Internet connectivity. While attached to the AAT, however, you can generate and *share* the PDFs via email, they just won't be sent until the device is connected to the Internet.
SCAST: Email report to customer	To email the report to a customer, touch the *Sharing* icon, and then touch the *mail* icon. Compose your email and press the *SEND* button to finish. You may wish to verify that the email with the report was sent successfully.
Summary	This concludes the video on using iOS devices for operating and managing the AAT. We've seen how to connect to the AAT's built-in WiFi hotspot and launch the AAT User Interface, as well as how to create shortcuts for quickly navigating to the AAT's web site. We also discussed using the Adobe Acrobat application for viewing and managing PDFs. Finally, we learned how to generate a site report and email it to a customer. If you need information on connecting to the AAT using Windows or Android Devices, refer to the specific training videos available on the training page at sunsight.com. Thanks for watching!

Shotlist, Images and Screen Captures for Video

Table of Contents

Creating a Treatment for Film/Video

Music Video Treatment

Title of Project

Confidential

This is the proposed music video treatment for the upcoming **Artist Name** single titled *Title of Project*.

Revision: *1f*

Joe Producer
555-867-5309
joe@prodcompany.com

Title of Project Music Video Casting Call and Character Breakdown

Production Title: *Title of Project*
Independent/Student/Studio: Independent
Production Type: Music Video

Production Location: NE San Antonio

Production Start Date: 08/05/2018
Production Wrap Date: 08/20/2018
Production Schedule: August 13 - 17 (*Preferred*)

We want to shoot between August 13th - August 17th, based on schedules. Prefer if cast can be available to shoot any day/time (weekdays, evenings or weekends), but we will work around schedules. Primary location will be NE San Antonio.

Producer(s):/Director(s): Production Company

Synopsis: "*Title of Project*" will be a combination narrative and performance video. The song is about ...

Character Breakdowns:

All parts are non-speaking, and we are in search of actors that are able to emote well, especially for the role of MOM.

Ethnicities aren't important, however, we will try to cast the **MOM**, **SON** and **DAUGHTER** roles with actors that *could be* related.

MOM (Female, 35-45) This is the lead role in the video. Mother arrives home and discovers... In the various scenes, we follow her through ... This character will be a very emotional role throughout the video. *Must be available to take family pictures before production begins.*

Scene Breakdowns:

Scene 1 - Narrative - Location S:0 Instrumental

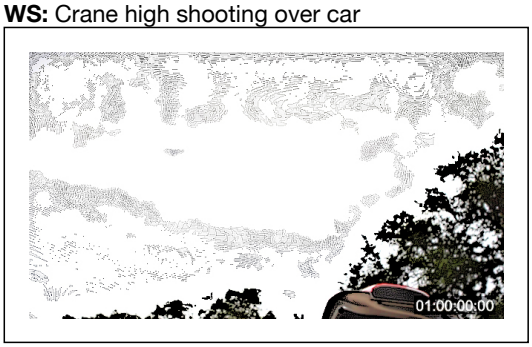
WARDROBE

PROPS Grocery Bags

MAKEUP

CAST MOM, **DAUGHTER**

WS: Crane high shooting over car



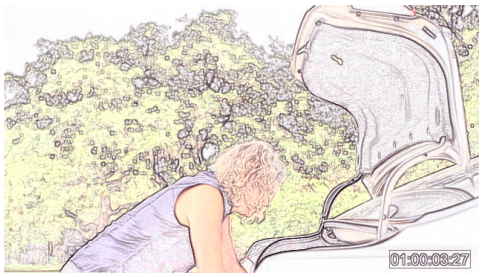
Shot Information	
Item	**Description**
Scene	
Shot	shot0
Desc	***WS:*** *Crane high shooting over car*

```
Title: Title of Project
Artist: Artist Name
Directed by: Dan Director
Produced by: Production Company

Probably just use the Title of Project and
Artist Name titles on this shot.
```

```
This needs to start high enough up that you
can't see the trunk, and make sure the sky
is NOT blown out! It comes down to reveal
MOM opening trunk and reaching in to grab
groceries.
```

Lens	**24mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes

WS: Crane Down to Mom removing groceries



Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	***WS:** Crane Down to Mom removing groceries*
Lens	**24mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes

```
Make sure MOM is already moving when she
comes into frame.
```

Narrative - **continues - Location S:15 L:~14s-**

Song Lyrics

I'm sorry momma
that you're reading this
I always wanted to make you smile
so I must go on with this

Scene 99 - Random Stuff

WS: Describe your opening master shot here

Stock - Clips of moments from his life

```
Various shots implying flashbacks to memories
or moments of his life. If needed
```

Stock - Possibly one w/**MOM** comforting son

Show his smile. Return to stock image. *If needed*

Cast Headshots

For the part of DAUGHTER

Actor1 Name - Source: TAW

Actor2 Name - Source: TAW

Proposed shooting schedule

Originally, production was planned for August 2018, but it has been delayed until September due to scheduling conflicts of both cast and crew. We are anticipating two to three (1/2) days for production as follows:

Scene 3 EXT

Cast: **MOM**, **DAUGHTER**, **SON**

Info: One (1) production day, evening shoot, *call time **5pm**, wrap time **8pm***.

Scene 1 EXT & Scene 2 INT

Cast: **MOM**, **DAUGHTER**

Info: 1/2 Production day, *call time **8a**, wrap time **12pm** (noon)*

Scene 1 EXT & Scene 2 INT

Cast: **MOM**

Info: 1/2 Production day, *call time **3p**, wrap time **8p***. This could be scheduled the same day as Scenes 1 and 2 if the actor is available and prefers this option (longer day)

NOTE: Actual shoot days will be scheduled according to cast availability

Table of Contents

Embedding Images in AV Shots (avshot)

This chapter will cover using the various builtins for using images in combination with the **avshot** builtin. If you have yet to review the section on **@image** builtins, go ahead and do that now, because the rest of this chapter assumes you have! We will provide some additional examples on using the **@image** builtins in **avshot**'s, and then move on to the more powerful builtins that are declared in the **avs/*.md** files.

Recall from the section on **@image** builtins that you can manipulate the size of the image using the **IMG_SIZE** macros. The image sizing macros, when used inside the **avshot** builtin will seem different, given the fact that the visual and audio columns split the document window in half, and thus the width will be a percentage of that half... Here's an example to illustrate the point, starting with the markdown, and the rendering immediately follows it:

```
***Using image variables inside avshot builtin***

// Declare an image variable

@image _id="myshot" src="[image_path]/shot1.jpg" style="
[!var.IMG_STYLE.inline_border!]"

// Start the A/V shot declaration

[avshot.visual]

// Change the image size to large and render it

[IMG_SIZE.large]

[myshot]

// Change the image size to thumb and render it

[IMG_SIZE.thumb]

[myshot]

// Close the A/V shot declaration

[avshot.noaudio]
```

And here is what the browser will render:

So although we specified an image size of large, it only used half the window. This is because in **avshot**'s, each column uses half the window, so the relative sizes used in the styling for images is a percentage of that column. Of course it works the other way too:

```
***Declare an image variable***
// Start the A/V shot declaration and add a shot descriptor
[avshot.visual]
WS: My wide shot tag here
// Switch to the Audio/Narrative
[avshot.audio]
// Change the image size to large and render it
[IMG_SIZE.large]
[myshot]
// Change the image size to thumb and render it
[IMG_SIZE.thumb]
[myshot]
// Close the A/V shot declaration
[avshot.end]
```

WS: My wide shot tag here

You will see many examples using these builtins throughout the user documentation, especially in the **Samples** chapters, so you may want to look around at them to see more.

avs/shot.md builtins

Moving on, let's now spend some time looking at the more powerful builtins that are included in the **sys.imports/avs/shot.md** file that are part of **smd**. As we did previously, let's start by having a look at the help strings for the macros we will be using.

Note that we won't look at all the building blocks in **shot.md**, just the more common things that you will use when writing A/V style markdown documents with shots and images. We will start with the factories: **image_factory**, **image_factory_abs_style** and **shot_factory**, which are used to create images and shots respectively, on the fly. The two image factories have additional macros to assist you, **image_factory_config** and **_img_template_**.

1. **image_factory** - The most commonly used macro
2. **image_factory_abs_style** - A variant that allows CSS styles to be specified directly
3. **image_factory_config** - A macro used to change the default styles used by the two previous macros
4. **_img_template_** - The template that is used when either factory creates a new @image variable.
5. **shot_factory** - A macro used to create a variable to store the technical aspects of an image.

> **What is a shot?**
>
> In this context, think of a *shot* as a way to describe the image in a technical fashion. Things like the camera information (lens, aperture, ISO, etc.) and even textual information. Because the two are *related*, this is one exception to the rule of

avoiding using identical names across variables in namespaces. For this case, it makes sense to create both an **@image** *shot1* and an **@var** *shot1*. Later on, we will see how this is used to create even more flexible macros.

Let's look at the help strings for all of these, including the **shot_factory**.

image_factory help:

```
  image_factory(nm="varname" ip="img_srcpath" st="CSS Style Var/Attr")

    Generates an @image declaration for nm with style specified as variable/attribute.

    See also: image_factory_abs_style.

  Parameters
    nm - Specifies the name to use for the @image variable
    ip - Specifies the image source path; can be absolute or relative
    st - Specifies an existing variable or attribute with CSS styling for the image

      If st is wrapped with exclamation marks, it will defer expansion until runtime
      Otherwise, the expansion occurs immediately. In other words, if you write:

        st="!IMG_STYLE.inline!" - then style is [IMG_STYLE.inline]
        st="IMG_STYLE.inline" - then style is the value of IMG_STYLE.inline

  Examples
    [image_factory(nm="myshot" ip="path/image.png" st="!IMG_STYLE.inline!")]
      Is the same as
    @image _id="myshot" _inherit="_img_template_" src="path/image.png" style="
[!IMG_STYLE.inline!]"

  Notes
    The @image variable created inherits attributes from _img_template_, which provides
methods
    to easily change the styling for the image after the variable is created. See
_img_template_.?.
```

image_factory_abs_style help:

```
  image_factory_abs_style(nm="varname" ip="img_srcpath" st="CSS Style")

    Generates an @image declaration for nm with style specified inline.

    See also: image_factory.

  Parameters
    nm - Specifies the name to use for the @image variable
    ip - Specifies the image source path; can be absolute or relative
    st - Specifies the CSS styling for the image. Default: {{IMG_STYLE.inline_border}}

  Examples
    [image_factory(nm="myshot" ip="path/image.png" st="{{IMG_STYLE.inline}}")]
      Is the same as
    @image _id="myshot" _inherit="_img_template_" src="path/image.png" style="
{{IMG_STYLE.inline}}"

  Notes
    The @image variable created inherits attributes from _img_template_, which provides
methods
    to easily change the styling for the image after the variable is created. See
_img_template_.?.
```

_img_template_ - template for @image variables help:

Variable created by **image_factory** or **image_factory_abs_style.**

Methods

_set_style(st) - Set the CSS styling for this variable to **st**
_set_style_as_var(st) - Set the CSS styling for this variable to the variable **st**

Examples

[imagevar._set_style(st="{{IMG_STYLE.inline}}")]
Sets **imagevar.style** to *margin-left:auto;margin-right:auto;width:90%;*

[imagevar._set_style_as_var(st="IMG_STYLE.block")]
Sets **imagevar.style** to *[IMG_STYLE.block]*

> *_img_template_* is inherited by variables created with one of the image factories, so each of the above attributes will exist in any **@image** variables created. They provide a means for easily changing the CSS styling used by an image after it has been created.

image_factory_config help:

image_factory_config Macro

This macro is used to change the default styling used by the image factories.

Methods
set_default_style(st) - Set the default style in **code.image_factory** to **st.**
set_default_style_abs_style(st) - Set the default style in **code.image_factory_abs_style** to **st.**

Notes
st is interpreted differently depending on which method is being used.
For **set_default_style,** it is an existing variable/attribute that is expanded when the image is rendered.
For **set_default_abs_style,** it is the CSS styling to be used when the image is rendered.

Examples
image_factory_config.set_default_style(st="IMG_STYLE.inline_border")
Sets **code.image_factory.st** to *[IMG_STYLE.inline_border]*

image_factory_config.set_default_abs_style(st="[IMG_STYLE.inline]")
Sets **code.image_factory.st** to *margin-left:auto;margin-right:auto;width:90%;*

shot_factory help:

shot_factory(nm="varname" d="desc" notes="shot notes" ...)

Generates a **shot** declaration on the fly

Shots are simply **@var** variables with a number of common attributes predefined, and several methods provided to perform common operations..

Parameters
nm - Specifies the name to use for the **@var** (i.e. *shot*) variable
d - The shot description.
l - The lens focal length.
f - The f/stop.
i - The ISO.
h - The camera/crane height.
c - Crane shot (Yes or No).
s - The scene name.
notes - The shot notes.

Note
When parameters are specified, they will override the default values.

Attributes
desc - The shot description. Default: *[_shot_defs_.desc]*
lens - The lens focal length. Default: *[_shot_defs_.lens]*
fstop - The f/stop. Default: *[_shot_defs_.fstop]*
iso - The ISO. Default: *[_shot_defs_.iso]*
height - The camera/crane height. Default: *[_shot_defs_.height]*
crane - Crane shot (Yes or No). Default: *[_shot_defs_.crane]*
scene - The scene name. Default: *[_shot_defs_.scene]*
notes - The shot notes. Default: *[_shot_defs_.notes]*

Note
The defaults are taken from **var._shot_defs_** unless they are specified when the shot is declared.

Methods:

NONE - If no method specified, returns a basic shot info table e.g. **[varname]**
visual - generate a shot info table suitable for the visual section of an avshot.
visual_wn - generate a visual section table with shot notes included.
audio - generate a shot info table suitable for the audio section of an avshot.
audio_wn - generate an audio section table with shot notes included.
with_notes - like **visual_wn** but with larger font and smaller table width.

Examples
[shot_factory(nm="myshot" s="my scene" d="opening shot")]

For our first example of using the factories, let's write the markdown for both the **image_factory** and the **shot_factory**, and test them out in an **avshot** sequence. This is one of the more common ways that you will use the factories in your documents. For example, if we write:

```
***Declare an image and a shot and then render them in AV style***
// Declare the image and the shot using the factories
[image_factory(nm="myshot" ip="/path/shot1.jpg" st="!IMG_STYLE.inline_border!")]
[shot_factory(nm="myshot" d="WS: Crane down" notes="Opening crane shot" c="Yes")]
// Start the shot and render the image and notes inline
[avshot.visual]
[image.myshot]
// Switch to audio/narration and render the shot info with notes
[avshot.audio]
[var.myshot.audio]
[avshot.end]
```

This is what we will get:

Shot Information	
Item	**Description**
Scene	
Shot	myshot
Desc	*WS: Crane down*
Lens	**24mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	Yes

Alternatively, you can inline the shot info directly below the shot image. Take a look at how to do that:

```
***Declare an image and a shot and then render them in AV style***
// Declare the image and the shot
[image_factory(nm="myshot" ip="/path/shot1.jpg" st="!IMG_STYLE.inline_border!")]
[shot_factory(nm="myshot" d="WS: Crane down" notes="Opening crane shot" c="Yes")]
// Start the shot and render the image and notes inline
[avshot.visual]
[image.myshot]
[var.myshot]
// Switch to audio/narration and render just the shot notes
[avshot.audio]
[var.myshot.notes]
[avshot.end]
```

This is what we will get:

```
Opening crane shot
```

Shot Information	
Item	**Description**
Scene	
Shot	myshot
Desc	*WS: Crane down*
Lens	**24mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	Yes

Because we are using the dynamic styling version of the image factory, if we use one of the **IMG_SIZE** methods to change the width, the same exact code will render the image differently. For example, assume the same sequence of markdown as before, only this time, right before we render the sequence, let's add **[IMG_SIZE.medium]**. Now we get this:

```
Opening crane shot
```

Shot Information	
Item	**Description**
Scene	
Shot	myshot
Desc	*WS: Crane down*
Lens	**24mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	Yes

So far so good, right? We have now seen how to use the image and shot factories to easily create variables that we can use when creating AV scripts. And by combining those with the **avshot** builtin, we can easily create detailed breakdowns for a film/video shoot.

Of course we can use these same *factory-created* image and shot variables outside **avshot**, so let's take one quick look at that before moving on to the next topic.

```
Use image and shot factory variables outside avshot
// Declare the image and the shot
[image_factory(nm="myshot" ip="/path/shot1.jpg" st="!IMG_STYLE.inline_border!")]
[shot_factory(nm="myshot" d="WS: Crane down" notes="Opening crane shot" c="Yes")]
// Render the image and then the shot info
[image.myshot]
[var.myshot.notes]
```

Which will render the following:

Shot Information	
Item	**Description**
Scene	
Shot	myshot
Desc	*WS: Crane down*
Lens	**24mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	Yes

One thing to notice if you didn't catch it is that you will always want to prefix the variable names with their namespace to guarantee that the parser will emit what you intend. If not, it will always emit the **var.** variable, since the **var** namespace variables have precedence over **image** namespace variables.

Before moving on, let's see a few more combinations of using the factory-created image and shot variables in different contexts.

More examples using the avs/shot.md builtins

Begin by creating a few variables we can use in our examples.

Declare some common variables to use in examples

```
// Declare the image and the shot
@var ss="{{var.IMG_STYLE.inline_border}}"
@var trythis="{:.red.bold}Try to get this shot"
@var beforeshoot="{:.red.bold}NEED TO GET THIS DONE BEFORE PRODUCTION"
```

Although it's convenient to declare as many of the attributes as possible when the factory variable is created, it is also possible to add/update them after the fact. This is most commonly done with variables created by the **shot_factory**.

Declaring shot attributes after factory creation step

```
// Declare the variable using the factory and render it
[shot_factory(nm="shot1")]
[var.shot1]
// Update attributes using the ._null_ method and render again
[var.shot1._null_(d="*WS: Crane down to reveal MOM*" c="yes" l="85mm")]
[var.shot1]
```

And here is what we get:

Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*Your shot description here.*
Lens	**24mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	No

Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes

You can also add more attributes using the same technique. For example, say you want to add **title** and **class** attributes to your factory image variable. Here's how you could do that:

Adding image attributes after factory creation step

```
// Declare the variable using the factory and render it
[IMG_SIZE.small]
[image_factory(nm="shot1" ip="/path/shot1.jpg" st="!IMG_STYLE.inline_border!")]
[image.shot1]
// Add attributes using the ._null_ method and render again
[image.shot1._null_(title="title text for image" class="myimageclass")]
[image.shot1]
```

After the declaration of **image.shot**, the public attributes are:

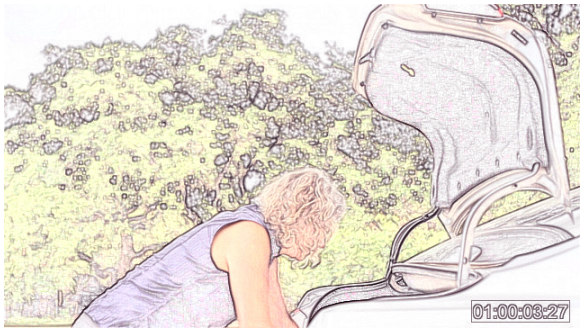
> ***src***=/Users/ken/Dropbox/shared/src/script/smd/docs/samples/image/shot1.jpg
> ***style***=margin-left:auto;margin-right:auto;width:40%;border:1px solid;padding:1em;

And it renders like this:

After the addition of the **title** and **class** attributes, the public attributes are:

> ***src***=/Users/ken/Dropbox/shared/src/script/smd/docs/samples/image/shot1.jpg
> ***style***=margin-left:auto;margin-right:auto;width:40%;border:1px solid;padding:1em;
> ***title***=title text for image
> ***class***=myimageclass

And it renders like this (hover over both images for a couple of seconds to see the title on the 2nd):



Variables created with the **shot_factory** contain another attribute and methods that are specialized, the *notes* attribute and its companion methods *addNote* and *addBB*. Let's see how they can be used to add additional textual information to a shot variable.

Recall that we previously created a shot variable called **shot1** using the **shot_factory** macro. Let's dump the shot now to see what it contains:

> **shot1=**
> **_format=**{{self.visual}}
> **visual=**{{html._table_si_.<}}{{self._basic_}}{{html.table.>}}
> **visual_wn=**{{html._table_si_.<}}{{self._basic_}}{{self._n_row_}}{{html.table.>}}
> **audio=**{{html._table_si_right_.<}}{{self._basic_}}{{html.table.>}}
> **audio_wn=**{{html._table_si_right_.<}}{{self._basic_}}{{self._n_row_}}{{html.table.>}}
> **with_notes=**{{html._table_si_alt_.<}}{{self._basic_}}{{self._n_row_alt_}}{{html.table.>}}

```
 _basic_={{html.tr.<}} {{html._td_header_.<}}<em><strong>Shot Information</strong></em>{{html.td.>}}{{html.tr.>}}
{{html.tr.<}} {{html._th_item_.<}}Item{{html.th.>}} {{html._th_desc_.<}}Description{{html.th.>}}{{html.tr.>}}{{html.tr.<}}
{{html._td_item_.<}}Scene{{html.td.>}} {{html._td_desc_.<}}<em><strong>{{self.scene}}</strong></em>{{html.td.>}}
{{html.tr.>}}{{html.tr.<}} {{html._td_item_.<}}Shot{{html.td.>}} {{html._td_desc_.<}}{{self._id}}{{html.td.>}}{{html.tr.>}}
{{html.tr.<}} {{html._td_item_.<}}Desc{{html.td.>}} {{html._td_desc_.<}}<em>{{self.desc}}</em>{{html.td.>}}{{html.tr.>}}
{{html.tr.<}} {{html._td_item_.<}}Lens{{html.td.>}} {{html._td_desc_.<}}<strong>{{self.lens}}</strong>{{html.td.>}}
{{html.tr.>}}{{html.tr.<}} {{html._td_item_.<}}f-stop{{html.td.>}} {{html._td_desc_.<}}<strong>{{self.fstop}}</strong>
{{html.td.>}}{{html.tr.>}}{{html.tr.<}} {{html._td_item_.<}}ISO{{html.td.>}} {{html._td_desc_.<}}<strong>{{self.iso}}</strong>
{{html.td.>}}{{html.tr.>}}{{html.tr.<}} {{html._td_item_.<}}Height{{html.td.>}} {{html._td_desc_.<}}<strong>{{self.height}}
</strong>{{html.td.>}}{{html.tr.>}}{{html.tr.<}} {{html._td_item_.<}}Crane{{html.td.>}} {{html._td_desc_.<}}{{self.crane}}
{{html.td.>}}{{html.tr.>}}
 _n_row_={{html.tr.<}} {{html._td_2span_.<}}{{self.notes}}{{html.td.>}}{{html.tr.>}}
 _n_row_alt_={{html.tr.<}} {{html._td_2span_alt_.<}}{{self.notes}}{{html.td.>}}{{html.tr.>}}
desc={{code.get_default(v="self.d", default="{{var._shot_defs_.desc}}")}}
lens={{code.get_default(v="self.l", default="{{var._shot_defs_.lens}}")}}
fstop={{code.get_default(v="self.f", default="{{var._shot_defs_.fstop}}")}}
iso={{code.get_default(v="self.i", default="{{var._shot_defs_.iso}}")}}
height={{code.get_default(v="self.h", default="{{var._shot_defs_.height}}")}}
crane={{code.get_default(v="self.c", default="{{var._shot_defs_.crane}}")}}
scene={{code.get_default(v="self.s", default="{{var._shot_defs_.scene}}")}}
notes={{var._shot_defs_.notes}}
addNote={{code.append(attr1="self.notes", attr2="self.val")}}
addBB={{self.addNote(val="{{bb}}")}}
val={{var._shot_defs_.notes}}
d=<em>WS: Crane down to reveal MOM</em>
l=85mm
f={{var._shot_defs_.fstop}}
i={{var._shot_defs_.iso}}
h={{var._shot_defs_.height}}
c=yes
s={{var._shot_defs_.scene}}
```

For now, let's focus in on **notes**, **addNote** and **addBB**. Let's print the current value of each of those attributes now, just to clear things up:

```
var.shot1.notes={{var._shot_defs_.notes}}
var.shot1.addNote={{code.append(attr1="self.notes", attr2="self.val")}}
var.shot1.addBB={{self.addNote(val="{{bb}}")}}
```

You can see that by default, **notes** returns the value of *{{var._shot_defs_.notes}}* which, if you have not changed it, is an empty string. We can add notes to a shot using the method **addNote**, and we can add a double blank line using **addBB**. So, if we wrote something like this:

```
// Add a shot note to shot1
[shot1.addNote(val="Mom reaches in and takes grocery bags from trunk")]

// Render the shot info table
[shot1.with_notes]
```

Will render as follows:

Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes
Mom reaches in and takes grocery bags from trunk	

If we review the current value of the **notes** attribute, we see:

var.shot1.notes={{var._shot_defs_.notes}}Mom reaches in and takes grocery bags from trunk

Notice that the *{{var._shot_defs_.notes}}* is still there, and our note has been appended to the value. This is intentional, allowing some type of document **default** to be present in the shot notes for every factory-generated shot variable. However, you can easily override it by simply setting the **notes** attribute to an empty string (or any other value), and then any additional calls to **addNote** will append to that.

Moving on, let's say we want to add another note, in this case, the previously defined **trythis** constant we created. We could do that with the following markdown:

```
// Add a double blank and the trythis note to shot1
[shot1.addBB]
[shot1.addNote(val="[trythis]")]
```

And now, it will render as follows:

Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes
Mom reaches in and takes grocery bags from trunk **Try to get this shot**	

If we review the current value of the **notes** attribute, we see:

var.shot1.notes={{var._shot_defs_.notes}}Mom reaches in and takes grocery bags from trunk

Try to get this shot

You get the idea, right? Add as many or as few additional notes to your shots, so that when you generate any documents for your production team, they will be alerted to things that need to be done. Before we leave, it's worth mentioning that it's quite common to actually use the shot table below the shot image in an AV script, and then print the shot notes in the audio/narrative section by simply referencing the **notes** attribute. Let's see how that is done.

```
Show usage of image and shot with notes in AV style
// Start the shot and render the image and shot table inline
[avshot.visual]
[image.shot1]
[var.shot1]
// Switch to audio/narration and render just the shot notes
[avshot.audio]
[var.shot1.notes]
[avshot.end]
```

This is what we will get:

Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes

```
Mom reaches in and takes grocery bags from
trunk

Try to get this shot
```

Up to this point, we have seen how to use the builtins in **avs/shot.md** to create variables that allow us to manage all the details of a *shot*. Again, a *shot* in this context refers to an **@image** variable and a **@var** variable that collectively describe a shot for a film or video project.

And the usage of these *shots* is simplified through the use of the attributes and methods associated with the two variables that together make up a shot, which greatly reduces the amount of markdown required to create rich documents to describe your project to others.

But as we've seen, many times we are still repeating a lot of markdown for each shot, so it seems that additional gains can be made through the addition of another level of builtins. As luck would have it, such a level is part **avs/shot.md**, so let's take a look at it now.

Wrappers for avs/shot.md builtins

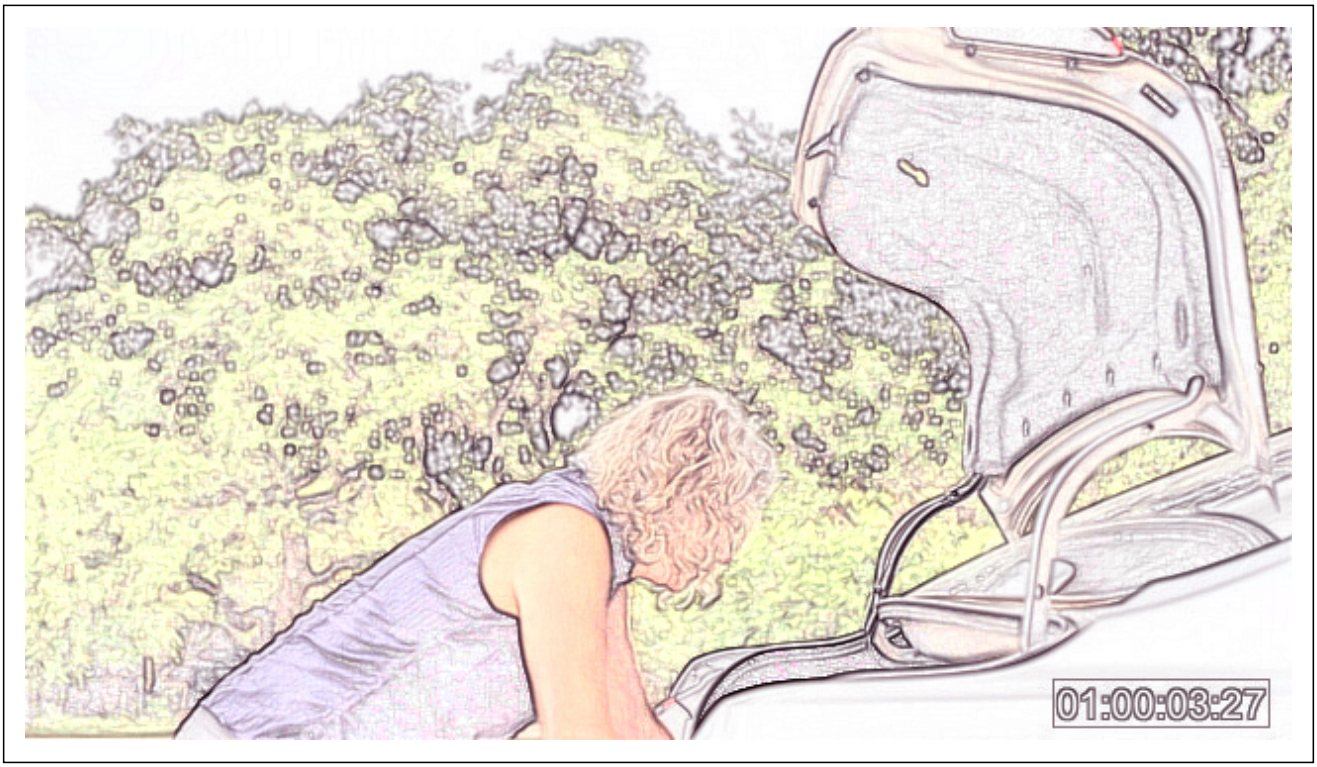
As we were discussing, although the builtins covered so far greatly reduce the markdown needed to manage shots in your documents, we are still repeating quite a bit of code each time we want to emit a shot and all of its accompanying information. The first step we need is a builtin that wraps a *shot*, and emits the more common cases via methods on that builtin. Enter **shotdetail**.

That the following markdown:

```
***Markdown to emit shot details***

[var.shot1.desc]

[image.shot1]

[var.shot1.with_notes]
```

That markdown will render as follows:

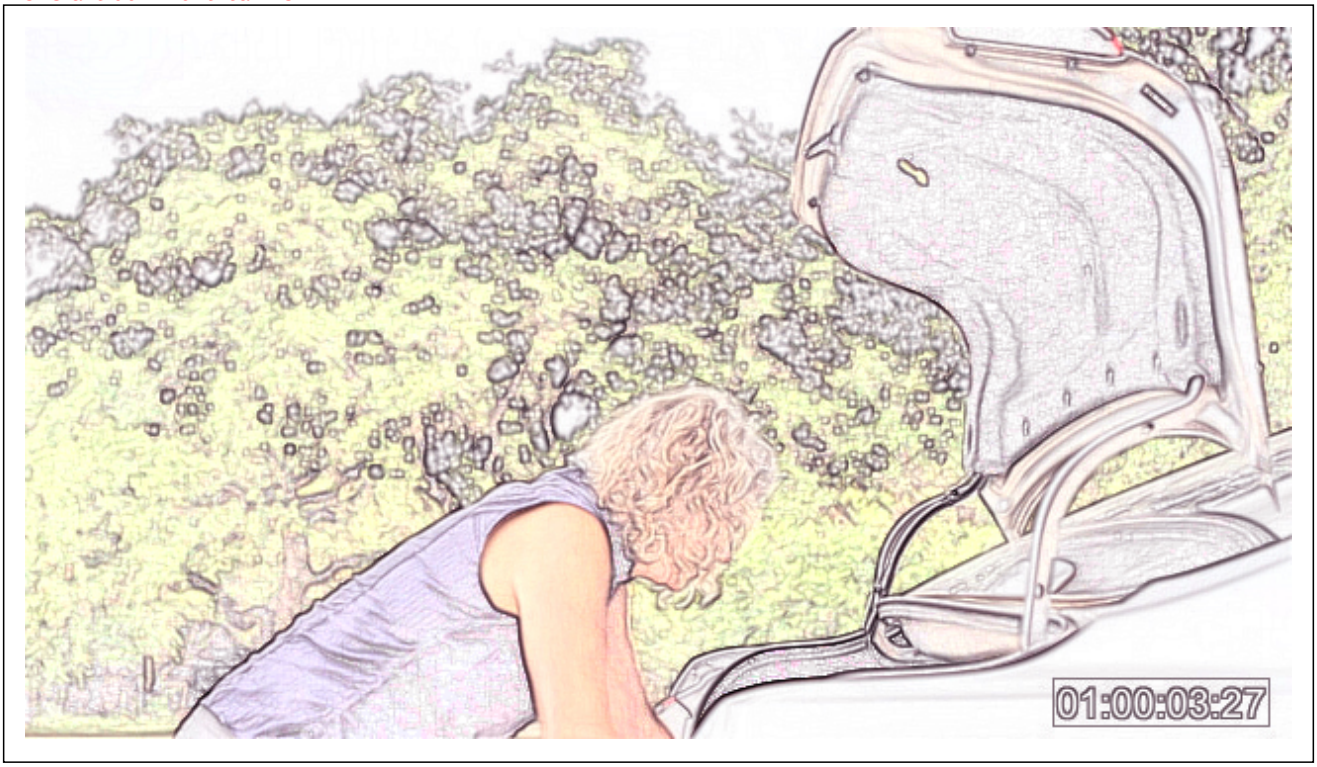
WS: Crane down to reveal MOM



Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes
Mom reaches in and takes grocery bags from trunk **Try to get this shot**	

The first thing to notice is that all three builtins take the name of the shot, in this case *shot1*. Besides that, the markdown would be the same for any shots created with the factories. So what we need is a builtin that takes a parameter, say *shotid*, and substitutes it on each of the lines where *shot1* is written, and then it would only take a single line of markdown to generate the same thing. That builtin is called **shotdetail**, so let's try it now and see what happens. We will write ***[shotdetail.with_notes(shotid="shot1")]***, and this is what the parser will emit:

WS: Crane down to reveal MOM



Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes
Mom reaches in and takes grocery bags from trunk **Try to get this shot**	

Okay, so that's much more abbreviated, and we get the same results. Let's take a quick look at the help for **shotdetail** to see what it supports:

```
***shotdetail help:***

  *shotdetail(shotid)*

    Emit shot details for shotid; requires that the image & shot factories created
shotid

  Methods

    basic(shotid) - Emits the shot description line, the image, and the details for
shotid.
    with_notes(shotid) - Same as basic, but emits the shot notes at the end of the
table.

    needshot(shotid) - Like basic, but uses _needshot_ image instead of image.shotid.
    needshot_wn(shotid) - Same as needshot, but emits the shot notes at the end of the
table.

  Notes

    The needshot options are provided because a sample image may not be available at
this time.
    This provides a placeholder/reminder that you need to get a sample shot/storyboard.

  Examples

    shotdetail.basic(shotid="shot_var_name")
        Emits [var.shot_var_name.desc] [image.shot_var_name] [var.shot_var_name]

    shotdetail.with_notes(shotid="shot_var_name")
        Emits same as shotdetail.basic but uses .with_content for the shot detail
table.

    shotdetail.needshot(shotid="shot_var_name")
        Emits same as shotdetail.basic but uses _needshot_ image instead of
image.shotid.
```

You probably noticed the two special methods **needshot** and **needshot_wn** that are available in **shotdetail**. These methods come in handy when you don't have an image readily available to show the framing or storyboard for the shot you are documenting. In this case, you can use the **needshot** methods, as they will use a system-provided image as a placeholder when generating the markdown for your shot. Later, when you have an image, you can update your document to use either **basic** or **with_notes**.

Let's see how you can use **needshot** in your document. Assume we are documenting **shot38**, which is a drone shot flying over a pasture somewhere. You've written the following markdown, but you know that the **image.shot38** variable will not resolve to a valid image at this point. So instead of using *[shotdetail.basic(shotid="shot38")]*, you will write *[shotdetail.needshot(shotid="shot38")]*.

```
***Sample showing usage of .needshot method***

*// Generate the shot38 variables*

[image_factory(nm="shot38")]

[shot_factory(nm="shot38")]

[var.shot38._null_(d="*WS: Drone flying over pasture*" c="No" l="50mm")]

*// Use needshot image since we don't have our own image for shot38 at this time*

[shotdetail.needshot(shotid="shot38")]
```

That markdown will render as follows:

WS: Drone flying over pasture

Shot Information	
Item	**Description**
Scene	
Shot	shot38
Desc	*WS: Drone flying over pasture*
Lens	**50mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	No

Of course we can also use the **shotdetail** builtin inside **avshot** sequences as well. For example:

```
Using .needshot inside avshot
[avshot.visual]
[shotdetail.needshot(shotid="shot38")]
[avshot.noaudio]
```

That markdown will render as follows:

WS: Drone flying over pasture

Shot Information	
Item	**Description**
Scene	
Shot	shot38
Desc	*WS: Drone flying over pasture*
Lens	**50mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	No

So this seems to point out the next missing piece of the puzzle: builtins that wrap the **avshot** sequences. Let's take a look at those.

Using the avs/shot.md builtins that wrap *avshot*

The last two builtin we will cover is the **shot_emitter**. It provides a convenient way to emit shots in the most commonly used formats. Here's the help for **shot_emitter**:

```
shot_emitter help:
  shot_emitter - Markdown Shot Emitter

    Automates the emitting of markdown for shots

  Methods

    split(shotid) - emits an avshot sequence with the shot left and the details right
    left(shotid) - emits an avshot sequence with the shot and details left and the shot
notes right
    shot_only(shotid) - emits the image inline raw (no @wrap other other HTML)
    shot_with_notes(shotid) - emits the image inline raw with detail table
```

Let's take a closer look at each of them. First up is **shot_emitter.split**, which emits an **avshot** sequence, and splits the visual (image) and the audio (notes) between the two columns. This is a space saving way to emit a shot list with storyboards. Here's what it looks like when we use write **[shot_emitter.split(shotid="shot1")]**:

Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes
Mom reaches in and takes grocery bags from trunk **Try to get this shot**	

Next up is shot left, which also emits an **avshot** sequence, but in this variant, it places the image and details on the left, and then puts the shot notes on the right. This is more typical of how you would use it in an A/V script. The markdown is: **[shot_emitter.left(shotid="shot1")]**:

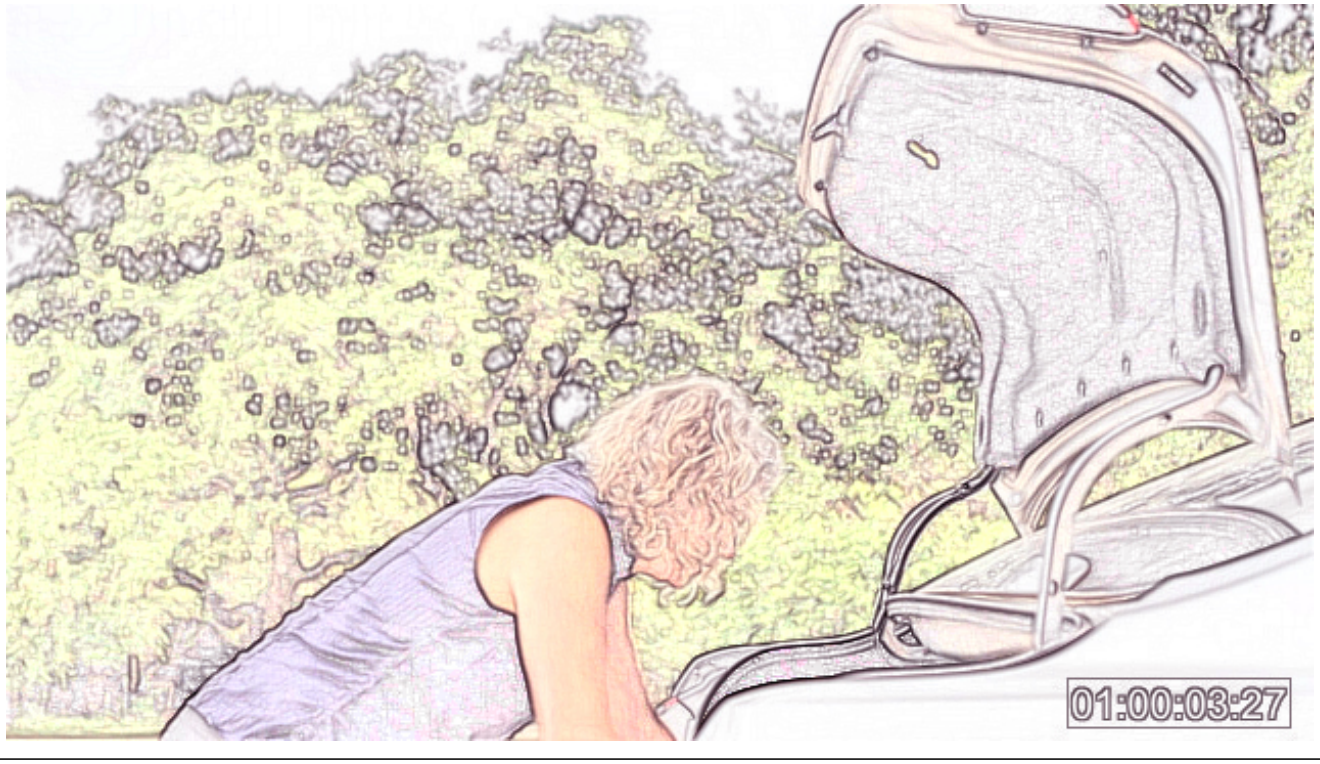
Mom reaches in and takes grocery bags from trunk

Try to get this shot

Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes

Next up is **shot_only**, which, as you probably guessed, simply emits the shot. The markdown: **[shot_emitter.shot_only(shotid="shot1")]**:

And finally, there's **shot_with_notes**, which displays the shot and notes table inline and raw, so you can use this version in or outside of an **avshot** sequence. The markdown: **[shot_emitter.shot_with_notes(shotid="shot1")]**:



Shot Information	
Item	**Description**
Scene	
Shot	shot1
Desc	*WS: Crane down to reveal MOM*
Lens	**85mm**
f-stop	**f/2.8**
ISO	**320**
Height	**36in**
Crane	yes
Mom reaches in and takes grocery bags from trunk **Try to get this shot**	

And we finally come to the end of this chapter. We've covered quite a bit, but hopefully you've got a good idea on how to use the builtins available for generating images and shot details for your documents. Be sure to review the actual markdown that was used to generate this documentation, as it will give more insight into how everything works.

Table of Contents

Summary

Well that's it! Hope you've enjoyed reading the docs for the smd utility. More importantly, I hope that you can use this app to streamline your html document and audio/visual script development!

Table of Contents