mayatools Documentation

Release 0.1

Western X

March 03, 2016

Contents

A collection of Python tools for working with(in) Autodesk's Maya.

CHAPTER 1

Contents

1.1 Context Managers

Context managers for restoring state after running requested tools.

These are generally quite useful for creating tools which must modify state that is normally under control of the user. By using these, the state will be set back to what the user left it at.

For example, several of Maya's tools work easiest when they are operating on the current selection, but that selection is not something that we want to expose to the user of our higher level tool.

1.1.1 Attributes

mayatools.context.**attrs**(**args*, ***kwds*)

Change some attributes, and reset them when leaving the context.

Parameters

- **args** – Mappings of attributes to values.
- **kwargs** – More attributes and values.

Returns A dictionary of the original values will be bound to the target of the with statement. Changed to that dictionary will be applied.

This is very for tools that must modify global state:

```
>>> with mayatools.context.attrs({'defaultRenderGlobals.imageFormat': 8}):
...     # Playblast with confidence that the render globals will be reset.
```

1.1.2 Selections

mayatools.context.**selection**(**args*, ***kwds*)

A context manager that resets selections after exiting.

Parameters

- **args** – Passed to `cmds.select`.
- **kwargs** – Passed to `cmds.select`.

A list of the original selection will be bound to the target of the with statement. Changes to that list will be applied.

Example:

```
>>> with selection(clear=True):
...     # Do something with an empty selection, but restore the user's
...     # selection when we are done.
```

1.1.3 Commands

mayatools.context.**command**(*func*, **args*, ***kwargs*)

A context manager that uses the standard query interface.

Pass any values via keyword arguments and their original values will be saved via `func(*args, query=True, yourAttribute=True)`, and finally restored via `func(*args, yourAttribute=original)` or `func(*args, edit=True, yourAttribute=original)` if you also specify `edit=True`.

Parameters

- **func** – A callable, or name of a Maya command.
- **args** – Positional arguments for the given `func`.
- **kwargs** – Values to set within the context. `edit` is special and marks if values should be set with an `edit` flag or not.

A dictionary of the original values will be bound to the target of the with statement. Changed to that dictionary will be applied.

If you are already using a query pattern like:

```
>>> current_time_unit = cmds.currentUnit(time)
>>> cmds.currentUnit(time='film')
>>>
>>> try:
...     # Do something.
... finally:
...     cmds.currentUnit(time=current_time_unit)
```

then you can use this manager directly:

```
>>> with command(cmds.currentUnit, time='film') as originals:
...     # Do something.
```

or as a context manager factory:

```
>>> currentUnit = command(cmds.currentUnit)
>>> with currentUnit(time='film') as originals:
...     # Do something.
```

If your command requires the `edit` keyword, pass it to this function:

```
>>> with ctx.command(cmds.camera, my_camera, edit=True, overscan=1):
...     # Do something with the camera.
```

1.1.4 User Interface

UI Refreshes

mayatools.context.**suspend_refresh**()
A context mananger that stops the graph from running or the view updating.

Can be nested, where only the outermost context manager will resume refresing.

```
>>> with suspend_refresh():
...     do_something_with_high_drawing_cost()
```

See also:

There are caveats with disabling the refresh cycle. Be sure to read about cmds.refresh.

Action Progress

class mayatools.context.**progress**(*status*, *max=100*, *min=0*, *cancellable=False*)
A context manager to assist with the global progress bar.

Parameters

- **status** (*str*) – The status message.
- **max** (*int*) – The maximum value.
- **min** (*int*) – The minimum value.
- **cancellable** (*bool*) – If the process is cancellable.

If the process is cancellable, you must periodically check *was_cancelled()* to see if the user did cancel the action. You must be more defensive in your programming than normal since this must allow the main event loop to process user events.

```
with progress("Testing", max=100, cancellable=True) as p:
    for i in range(100):
        if p.was_cancelled():
            cmds.warn('You cancelled the process!')
            break
        time.sleep(0.02)
        p.update(i, 'Testing %d of 100' % (i + 1))
```

hide()
Hide the progress bar.

show()
Show the progress bar.

step(*size=1*)
Increment the value.

update(*value=None*, *status=None*, *min=None*, *max=None*)
Update the value and status.

was_cancelled(*max_time=0.01*)
Check if the user requested the action be cancelled.

Parameters **max_time** (*float*) – The maximum number of seconds to spend in the main event loop checking for user actions.

Returns **bool** True if the user requested the action be cancelled.

1.2 Downgrading Maya Scenes

In the current Western X pipeline, some steps are being performed with different versions of Maya for various reasons. Unfortunately, the version number does not always move up as scenes move from step to step, so sometimes we must downgrade while retaining as much information as possible.

The *downgrade_to_2011()* function is being developed as needs see fit to downgrade ASCII scene files to Maya 2011.

mayatools.downgrade.**downgrade_to_2011**(*src_path*, *dst_path*)

Downgrade the given Maya ASCII scene to version 2011.

Parameters

- **src_path** (*str*) – The scene >2011 to downgrade.
- **dst_path** (*str*) – Where to write the 2011 version.

Currently handles:

•cameras

•image planes

•meshes (roughly)

1.3 Binary Files

1.3.1 API Reference

This packages provides classes for reading and writing Maya's IFF inspired *binary file format*.

High-Level

class mayatools.binary.**Parser**(*file*)

Maya binary file parser.

Parameters **file** – The file-like object to parse from; must support `read(size)` and `tell()`.

parse_all()

Parse the entire (remaining) file.

parse_next()

Parse to the next *Group* or *Chunk*, returning it.

This is useful when you want to head the headers of a file without loading its entire contents into memory.

pprint(*_indent=-1*)

Print a structured representation of the file to stdout.

Graph Nodes

class mayatools.binary.**Node**

Base class for group nodes in, and the root node of a Maya file graph.

children = None

The children of this node.

dumps_iter()
Iterate chunks of the packed version of this node and its children.

To write to a file:

```
with open(path, 'wb') as fh:
    for chunk in node.dumps_iter():
        fh.write(chunk)
```

find(*tag*)
Iterate across all descendants of this node with a given tag.

find_one(*tag*, **args*)
Find the first descendant of this node with a given tag.

Parameters

- **tag** (*str*) – The tag to find.
- **default** – What to return if we can't find a node.

Raises **KeyError** – if we can't find a tag and no default is given.

class mayatools.binary.**Group**(*tag*, *type_='FOR4'*, *size=0*, *start=0*)
A group node in a Maya file graph.

pprint(*_indent=0*)
Print a structured representation of the group to stdout.

tag = None
The data type.

type = None
The group type (e.g. FORM, LIST, PROP, CAT).

class mayatools.binary.**Chunk**(*tag*, *data=''*, *offset=None*, ***kwargs*)

data = None
Raw binary data.

floats
Binary data interpreted as array of floats.

This is settable to an iterable of floats.

ints
Binary data interpreted as array of unsigned integers.

This is settable to an iterable of integers.

pprint(*_indent*)
Print a structured representation of the node to stdout.

string
Binary data interpreted as a string.

This is settable with a string.

tag = None
The data type.

Decoding

mayatools.binary.**tag_encoding** = {'INFO': 'string', 'ETIM': 'uint', 'VERS': 'string', 'MADE': 'string', 'CHNG': 'string', 'STIM':

mayatools.binary.**register_encoder**(*names*, *encoder=None*)

Register an *Encoder* for the given type names.

These types are a concept of this module, and have no parallel in the file format itself. These are what we use to unpack the raw binary data into something standard Python types.

Types that are registered upon import include:

- "float";
- "uint" (32-bit big-endian integer);
- "string" (NULL terminated).

Parameters

- **names** – A string, or iterable of strings.
- **encoder** – The *Encoder* to use for this type.

This function can operate as a decorator as well:

```
@register_encoder('attr')
class AttributeEncoder(mayatools.binary.Encoder):
    pass
```

mayatools.binary.**get_encoder**(*tag*)

Get an *Encoder* for the given tag.

Parameters **str** – The 4 character node "tag".

Returns The appropriate *Encoder* or None.

class mayatools.binary.**Encoder**

The base class for encoding/decoding packed data to/from native types.

repr_chunk(*chunk*)

Create string representation of a chunk returned from *split()*.

split(*encoded*, *size_hint*)

Return an iterator over a split version of the encoded data.

Parameters

- **encoded** (*str*) – The packed data to split.
- **size_hint** (*int*) – The suggested split size. Feel free to ignore this if your data has an implicit size of its own.

1.3.2 Anatomy of a Binary File

Lets look a complete frame from a binary fluid cache (in pseudo `hexdump -C` fashion):

```
0000: 464f5234 00000028 43414348 5652534e FOR4...(CACHVRSN
0010: 00000004 302e3100 5354494d 00000004 ....0.1.STIM....
0020: 000000fa 4554494d 00000004 000000fa ....ETIM........
0030: 464f5234 000001a0 4d594348 43484e4d FOR4....MYCHCHNM
0040: 00000014 666c7569 64536861 7065315f ....fluidShape1_
```

```
0050: 64656e73 69747900 53495a45 00000004 density.SIZE....
0060: 0000003c 46424341 000000f0 447a0000 ...<FBCA....Dz..
0070: 44898000 44960000 447c8000 448ac000 D...D...D|..D...
0080: 44974000 447f0000 448c0000 44988000 D.@.D...D...D...
0090: 4480c000 448d4000 4499c000 447a4000 D...D.@.D...Dz@.
00a0: 4489a000 44962000 447cc000 448ae000 D...D...D|..D...
00b0: 44976000 447f4000 448c2000 4498a000 D.``.D.@.D...D...
00c0: 4480e000 448d6000 4499e000 447a8000 D...D.``.D...Dz..
00d0: 4489c000 44964000 447d0000 448b0000 D...D.@.D}..D...
00e0: 44978000 447f8000 448c4000 4498c000 D...D...D.@.D...
00f0: 44810000 448d8000 449a0000 447ac000 D...D...D...Dz..
0100: 4489e000 44966000 447d4000 448b2000 D...D.``.D}@.D...
0110: 4497a000 447fc000 448c6000 4498e000 D...D...D.``.D...
0120: 44812000 448da000 449a2000 447b0000 D...D...D...D{..
0130: 448a0000 44968000 447d8000 448b4000 D...D...D}..D.@.
0140: 4497c000 44800000 448c8000 44990000 D...D...D...D...
0150: 44814000 448dc000 449a4000 43484e4d D.@.D...D.@.CHNM
0160: 00000017 666c7569 64536861 7065315f ....fluidShape1_
0170: 7265736f 6c757469 6f6e0000 53495a45 resolution..SIZE
0180: 00000004 00000003 46424341 0000000c ........FBCA....
0190: 40400000 40800000 40a00000 43484e4d @@..@...@...CHNM
01a0: 00000013 666c7569 64536861 7065315f ....fluidShape1_
01b0: 6f666673 65740000 53495a45 00000004 offset..SIZE....
01c0: 00000003 46424341 0000000c 00000000 ....FBCA........
01d0: 00000000 00000000                   ........
```

This is a very small fluid constructed purely for reverse-engineering purposes. It is 3x4x5 (all relatively prime to easily spot patterns in iteration order), and the densities have been set to `1000 + 100 * xi + 10 * yi + zi` to directly read the coordinates of each density value. I have also left off velocity (which has a different layout).

High Level Overview

Nearly every binary file that Maya will generate has the same basic structure: a DAG (directed acyclic graph) in which every node has a 4 character type, a 32 bit size, and then a blob of packed binary data of the previously given size. A limited set of node types are known to be "groups" (i.e. non-leaf nodes) and the rest are data nodes (i.e. leaves of the graph).

In the case of groups, the packed data is a 4 character group type (so that it may have a purpose beyond simply being flagged as a group) and the group's serialized children.

In the case of data nodes, the packed data is interpreted as floats, ints (big-endian), strings (with trailing `NULL` s), etc..

Structure of a Group

Take a closer look at the first few bytes of the file:

```
0000: 464f5234 00000028 43414348 5652534e FOR4...(CACHVRSN
```

The first four bytes (hex `464f5234` or ascii `FOR4`) flags the start of a group node. There are four base group types (`"FORM"`, `"CAT "`, `"LIST"`, and `"PROP"`) with three different alignments variations (defaulting to 2 bytes, a `4` suffix for 4 bytes, and a `8` suffix for 8 bytes). Ergo, `FOR4` is a `FORM` group where its direct children are aligned to 4 byte boundaries.

(Don't ask me what the 4 different group types mean, as I don't really know at this point. Nearly all of the groups I have seen are `"FOR4"`....)

The next four bytes (hex `00000028`) are a 32-bit (big-endian) unsigned integer indicating the size of this node's data. Every node in the DAG has this size field. In this case, our group's data type and children take up 40 (i.e. 0x28) bytes.

The first four bytes of the group's data (ascii `CACH`) indicate that this is a "cache header" group.

The remaining 36 bytes are packed child nodes.

Data Nodes

Look at the complete `CACH` node, taking up the first 48 bytes (4 for the group's node type, 4 for the size, and 40 for the group's type and children):

```
0000: 464f5234 00000028 43414348 5652534e FOR4...(CACHVRSN
0010: 00000004 302e3100 5354494d 00000004 ....0.1.STIM....
0020: 000000fa 4554494d 00000004 000000fa ....ETIM........
```

First we have the node type, data size, and group type (as discussed above).

Then we hit 3 data nodes in a row: A `VRSN` of length 4 with data `302e3100` (i.e. the string `"0.1"`), a `STIM` of length 4 with data `000000fa` (i.e. the integer 250), and a `ETIM` of length 4 with data `000000fa` (i.e. also 250).

Unfortunately, the interpretation of the values above into strings and integers is due to manual intervention on my part; the encoded data says nothing about the way the data itself is packed. I have slowly been building a schema of types and their interpretation, but it is extremely far from complete.

Padding

The size of packed data must be a multiple of the alignment of the group which contains it. In this file, all of the groups are of type `"FOR4"`, and so the packed size of the nodes must be a multiple of 4. If the natural size is not, then `NULL`s will be added to the end, but the size field of the node will not reflect that padding; you must determine the padding size on your own as you read the file.

Notice that the second `CHNM` ("fluidShape1_resolution") is reported as being 23 bytes (22 characters and a terminating `NULL`), but it is padded out to 24 bytes because it is in a `FOR4` group.

Interpreting It All

Running this file through *the parser*, all of the structure is revealed:

```
CACH group (FOR4); 40 bytes for 3 children:
    VRSN; 4 bytes as string(s)
        0014: 302e31                            '0.1'
    STIM; 4 bytes as uint(s)
        0020: 000000fa                          250
    ETIM; 4 bytes as uint(s)
        002c: 000000fa                          250
MYCH group (FOR4); 416 bytes for 9 children:
    CHNM; 20 bytes as string(s)
        0044: 666c7569 64536861 7065315f 64656e73 697479 'fluidShape1_density'
    SIZE; 4 bytes as uint(s)
        0060: 0000003c                          60
    FBCA; 240 bytes as float(s)
        006c: 447a0000 44898000 44960000 447c8000 1000.0 1100.0 1200.0 1010.0
        007c: 448ac000 44974000 447f0000 448c0000 1110.0 1210.0 1020.0 1120.0
        008c: 44988000 4480c000 448d4000 4499c000 1220.0 1030.0 1130.0 1230.0
        009c: 447a4000 4489a000 44962000 447cc000 1001.0 1101.0 1201.0 1011.0
        00ac: 448ae000 44976000 447f4000 448c2000 1111.0 1211.0 1021.0 1121.0
```

```
        00bc: 4498a000 4480e000 448d6000 4499e000 1221.0 1031.0 1131.0 1231.0
        00cc: 447a8000 4489c000 44964000 447d0000 1002.0 1102.0 1202.0 1012.0
        00dc: 448b0000 44978000 447f8000 448c4000 1112.0 1212.0 1022.0 1122.0
        00ec: 4498c000 44810000 448d8000 449a0000 1222.0 1032.0 1132.0 1232.0
        00fc: 447ac000 4489e000 44966000 447d4000 1003.0 1103.0 1203.0 1013.0
        010c: 448b2000 4497a000 447fc000 448c6000 1113.0 1213.0 1023.0 1123.0
        011c: 4498e000 44812000 448da000 449a2000 1223.0 1033.0 1133.0 1233.0
        012c: 447b0000 448a0000 44968000 447d8000 1004.0 1104.0 1204.0 1014.0
        013c: 448b4000 4497c000 44800000 448c8000 1114.0 1214.0 1024.0 1124.0
        014c: 44990000 44814000 448dc000 449a4000 1224.0 1034.0 1134.0 1234.0
    CHNM; 23 bytes as string(s)
        0164: 666c7569 64536861 7065315f 7265736f 6c757469 6f6e 'fluidShape1_resolution'
    SIZE; 4 bytes as uint(s)
        0184: 00000003                           3
    FBCA; 12 bytes as float(s)
        0190: 40400000 40800000 40a00000         3.0 4.0 5.0
    CHNM; 19 bytes as string(s)
        01a4: 666c7569 64536861 7065315f 6f666673 6574 'fluidShape1_offset'
    SIZE; 4 bytes as uint(s)
        01c0: 00000003                           3
    FBCA; 12 bytes as float(s)
        01cc: 00000000 00000000 00000000         0.0 0.0 0.0
```

1.4 Debugging

from mayatools import debug debug.enable_verbose_commands()

cmds.about(version=True)

mayatools.debug.**disable_verbose_commands**()
x

mayatools.debug.**enable_verbose_commands**()
x

1.5 Interoperating with Qt

mayatools.qt.**get_maya_window**()
Get the main Maya window as a QtGui.QMainWindow.

mayatools.qt.**maya_to_qt**(*maya_object*)
Convert a Maya UI path to a Qt object.

Parameters **maya_object** (*str*) – The path of the Maya UI object to convert.

Returns QtCore.QObject or None

Todo

- camera import/export
- geocache import/export
- playblasting
- reference edit import/export

- locator export
- testing
- cachefile parsing
- shelfs and menus
- Shotgun tickets

m