bana Documentation

Release 0.1.0

Christopher Crouzet

May 09, 2017

Contents

Welcome! If you are just getting started, a recommended first read is the *Overview* as it shortly covers the *why*, *what*, and *how*'s of this library. From there, the *Installation* then the *Tutorial* sections should get you up to speed with the basics required to use it.

Looking how to use a specific function, class, or method? The whole public interface is described in the *API Reference* section.

Please report bugs and suggestions on GitHub.

CHAPTER 1

User's Guide

Overview

The Maya's Python API is often a good choice over the commands layer whenever performances and robustness are valued. But because of its overall poor design, it is not uncommon that some *fundamental* functionalities are **lacking** out of the box and/or require too much **boilerplate** to get rolling.

Other gotchas to be expected include methods that became too **daunting** to use after porting in the *worst* possible way the API from C++ to Python, **undocumented** behaviours of certain features where error trialing is everything that is left, and methods **throwing** an exception when returning `None` would have been more appropriate.

Bana aims at reducing these shortcomings to provide a more *friendly*, *predictable*, and *efficient* developing environment.

Using the monkey patching package `gorilla`, new methods prefixed with `bn` are inserted within some classes from the `maya.OpenMaya*` modules, thus extending their functionalities while making these new methods feel as if they were built-in into Maya.

Since performances are a primary reason for using the API, a set of benchmarks built with the help of the package `revl` helps to ensure that these extensions remain as fast as possible.

Note: Bana extends on Maya's Python API 1.0 rather than 2.0 because the latter version seems to be still incomplete. That being said, it is encouraged to use the API 2.0 whenever possible since it provides a much more Pythonic interface with increased performances.

Note: Bana does *not* aim at making the API more Pythonic. This could in some cases impact the performances, which goes against Bana's goal of keeping things fast.

Note: Bana *does* aim at following Maya's API philosophy by providing low-level extensions that are not specific to a domain (e.g.: rigging).

Features

- easy retrieval of nodes from the scene.
- robust and predictable specification for pattern matching with wildcards.
- abstract away the usage of the `maya.OpenMaya.MScriptUtil` class.
- performances as a top priority.

Usage

```
>>> import bana
>>> bana.initialize()
>>> from maya import OpenMaya
>>> # Retrieve a transform node named 'root'.
>>> root = OpenMaya.MFnTransform.bnGet(pattern='*|root')
>>> # Recursively iterate over all the DAG nodes child of 'root'.
>>> for node in root.bnFindChildren():
...     print(node)
>>> # Find all the mesh nodes in the scene containing the word 'Shape' but
... # not belonging to any namespace.
>>> for node in OpenMaya.MFnMesh.bnFind(pattern='*|*Shape*'):
...     print(node)
```

See also:

The *Tutorial* section for more detailed examples and explanations on how to use Bana.

Installation

Bana requires to be run from within an Autodesk Maya's Python environment. This is usually done either by running the code from within an interactive session of Maya, or through using the `mayapy` shell. A Python interpreter is already distributed with Maya so there is no need to install one.

Additionally, Bana depends on the `gorilla` package.

Note: Package dependencies are automatically being taken care off when using `pip`.

Installing pip

The recommended[1] approach for installing a Python package such as Bana is to use `pip`, a package manager for projects written in Python. If `pip` is not already installed on your system, you can do so by following these steps:

1. Download `get-pip.py`.
2. Run `python get-pip.py` in a shell.

Note: The installation commands described in this page might require `sudo` privileges to run successfully.

---

[1] See the Python Packaging User Guide

System-Wide Installation

Installing globally the most recent version of Bana can be done with `pip`:

```
$ pip install bana
```

Or using `easy_install` (provided with `setuptools`):

```
$ easy_install bana
```

Development Version

To stay cutting edge with the latest development progresses, it is possible to directly retrieve the source from the repository with the help of Git:

```
$ git clone https://github.com/christophercrouzet/bana.git
$ cd bana
$ pip install --editable .[dev]
```

Note: The `[dev]` part installs additional dependencies required to assist development on Bana.

Tutorial

One cool thing with these extensions is that there isn't much to know to get rolling—you'll be using the same old Maya's Python API as you've always done, only with a few extra methods at your disposal that have been injected here and there.

All there is to make these extensions available as part of Maya's API is to initialize them:

```
>>> import bana
>>> bana.initialize()
```

Done! Now you can head over to the *API Reference* section and make use of any of the extensions listed in there.

Note: Feel free to check out the *Pattern Matching* and *Retrieving Nodes* sections for guides about some core features included with Bana.

Pattern Matching

The API of Maya has a well-defined syntax to describe DG names and DAG paths but the solution offered to match wildcard patterns, through the use of methods such as `maya.OpenMaya.MGlobal.getSelectionListByName()`, can sometimes lead to unexpected results.

As an example, Maya defines the pattern `|*` as matching only the DAG nodes of depth 1, that is the nodes directly parented under the world. Therefore, when using a similar pattern applied to the underworld, for instance `node|shape->|*`, one would intuitively expect that only the nodes located directly beneath the underworld are to

be matched. Instead, Maya's implementation leads to match all the nodes at any depth below the underworld, which is inconsistent.

To alleviate this lack of predictability and to add a whole new set of possibilities loosely borrowed from Python's `re` module, a new specification dedicated to matching name and path patterns is being used across the Bana extensions, mostly through the `bnFind*()` and `bnGet*()` methods (see *Retrieving Nodes*).

This pattern matching specification introduces:

- a new *syntax* built upon Maya's DG names and DAG paths syntaxes, with support for four wildcard operators `*`, `+`, `?`, and `..`
- a well-defined set of *matching rules* describing the expected behaviour when using these wildcards in each possible scenario.

Syntax

The standard syntax defined by Maya, and recognized by Bana, describes DG names and DAG paths as they are expected to be returned by methods like `maya.OpenMaya.MFnDependencyNode.name()` and `maya.OpenMaya.MFnDagNode.fullPathName()`:

```
alpha      ::=   ``a''...''z'' | ``A''...''Z'' | ``_''
character  ::=   alpha | ``0''...''9''
name       ::=   alpha character*
full_name  ::=   (name '':'')* name
path       ::=   (``|'' full_name)+
full_path  ::=   path (``->'' path)* ``->''?
```

The library Bana extends the standard syntax by adding support for the four wildcard operators:

```
wcard            ::=   ``*'' | ``+'' | ''?'' | ''.''
wcard_name       ::=   (alpha | wcard+) (character | wcard+)*
wcard_full_name  ::=   ('':''  wcard+ | wcard_name) ('':''  wcard_name)*
wcard_path       ::=   (``|'' wcard_full_name | wcard+)+
wcard_full_path  ::=   wcard_path (``->'' wcard_path)* ``->''?
```

Note: The syntax groups are listed in ascending precedence order. In other words: *character* < *name* < *full name* < *path* < *full path*. This is useful for determining the *context*.

In English

Names can identify DG nodes, excluding the ones carrying any namespace or hierarchy information. They are made of *character* elements, that is *alphanumeric characters*, *underscores*, and *wildcards*.

Full names can fully identify any DG node. They are composed by one or more *name* elements, each separated by the namespace delimiter `:`.

Paths can identify DAG nodes, excluding the ones carrying any underworld information. They are composed by one or more *full name* elements, each starting with the hierarchy delimiter `|`.

Full paths can fully identify any DAG node. They are composed by one or more *path* elements, each separated by the underworld delimiter `->`.

Patterns can be checked against any of these syntax groups using the corresponding `bana.OpenMaya.MGlobal.bnIsValid*()` method:

```
>>> import bana
>>> bana.initialize()
>>> from maya import OpenMaya
>>> OpenMaya.MGlobal.bnIsValidName('node')
True
>>> OpenMaya.MGlobal.bnIsValidName('node_*', allowWildcards=True)
True
>>> OpenMaya.MGlobal.bnIsValidName('ns:node')
False
>>> OpenMaya.MGlobal.bnIsValidFullName('ns:node')
True
>>> OpenMaya.MGlobal.bnIsValidFullName('*:node', allowWildcards=True)
True
>>> OpenMaya.MGlobal.bnIsValidFullName('|node')
False
>>> OpenMaya.MGlobal.bnIsValidPath('|node')
True
>>> OpenMaya.MGlobal.bnIsValidPath('*|node', allowWildcards=True)
True
>>> OpenMaya.MGlobal.bnIsValidPath('|root->|node')
False
>>> OpenMaya.MGlobal.bnIsValidFullPath('|root->|node')
True
>>> OpenMaya.MGlobal.bnIsValidFullPath('*->|node', allowWildcards=True)
True
```

TL;DR

The composition of *names*, *full names*, *paths*, and *full paths*, can approximately be summed up as follows:

- a *`name`* is composed of one or more *`character`* elements.
- a *`full name`* is composed of one or more *`name`* elements separated by the `:` symbol.
- a *`path`* is composed of one or more *`full name`* elements separated by the `|` symbol.
- a *`full path`* is composed of one or more *`path`* elements separated by the `->` symbol.

Matching Rules

Depending on where a wildcard operator is located within a pattern, it might end up matching a certain number of occurrences of either one of the *`character`*, *`name`*, *`full name`*, or *`path`* syntax groups. For example the wildcard in the pattern `|node_*` matches a *name* formed by any number of *characters*, but the same wildcard in the pattern `*|node` matches a *path* composed by any number of *full names* (e.g.: `|root|parent|node`).

In order to understand what a wildcard, or a combination of wildcards, will precisely match, there are two aspects to take into consideration:

- the *context* in which the wildcards are defined.
- the *number of occurrences* that the wildcards describe.

Context

The context represents the syntax group to be matched. It can be determined by looking at the delimiters surrounding the wildcards, picking the one with the highest precedence, and retrieving the syntax group associated with it as defined in this table sorted in descending precedence order:

delimiter	syntax group
character	*name*
`:`	*full name*
`|`	*path*
`->`	*full path*

For example, the wildcard in the pattern `|ns:*|leaf` is surrounded by the delimiters `:` and `|`, respectively representing the *full name* and *path* syntax groups, hence the context is *full name* since it has a higher precedence than *path*.

When the wildcards are located at the beginning or the end of a string, then the only delimiter found is used to define the context. For example, the context for the wildcard in the pattern `*->|leaf` is *full path*, as per the `->` delimiter.

If one of the delimiters is a *character*, then the context is bound to be *name*. The pattern `|node*->leaf` is an example of such a case.

Finally, if a pattern is only composed of wildcards, then the global context defined by the matching method called is used. For example the method *`MGlobal.bnMatchFullPath()`* defines the global context *full path*.

Number of Occurrences

Remember how, according to the rules of *syntax composition*, a syntax group might be made of one or more elements of another syntax group. With this in mind, the number of occurences specifies how many elements of a context needs to be matched.

The special characters `*`, `+`, `?`, and `.` all carry the same purpose of matching a context element but a different number of times. The quantity being described by these wildcards is the same as their regular expression language counterparts, meaning that:

- `*` matches 0 or more occurrences of a context element.
- `+` matches 1 or more occurrences of a context element.
- `?` matches 0 or 1 occurrences of a context element.
- `.` matches 1 occurrence of a context element.

As an example, if the context is *`full name`*, then the quantifier defines how many *`name`* elements needs to be matched: the wildcard in the pattern `|ns:+|leaf` will match 1 or more *names* separated by the `:` delimiter, thus forming in the end a *full name*.

Matching Nothing

It sometimes makes sense to allow a wildcard to match zero occurrences. This is especially useful when performing recursive searches where the pattern `*|leaf` can match any node named `leaf`, including the one directly par-

ented under the world, and where the pattern `|ns:*:leaf` can match nodes such as `|ns:ns2:ns3:leaf` and `|ns:leaf`.

In some other cases, this doesn't make too much sense. For example the pattern `|ns:*` cannot match any node named `|ns:` because this isn't a valid pattern.

To check if a wildcard is allowed to match zero occurrences or not, see the *TL;DR* table.

TL;DR

The table below regroups all the possible valid uses of wildcard operators located between two adjacent delimiters.

Reminder

If the occurrence of wildcard is not listed in this table, it is bound to belong to the *name* context.

pattern	example	context	can match nothing
`^@$`	`@`	same as the global context	yes
`^@:`	`@:leaf`	*full name*	yes
`^@|`	`@|leaf`	*path*	yes
`^@->`	`@->|leaf`	*full path*	yes
`:@$`	`|ns:@`	*full name*	no
`:@:`	`|ns:@:leaf`	*full name*	yes
`:@|`	`|ns:@|leaf`	*full name*	no
`:@->`	`|ns:@->|leaf`	*full name*	no
`|@$`	`|root|@`	*path*	no
`|@:`	`|root|@:leaf`	*full name*	yes
`|@|`	`|root|@|leaf`	*path*	yes
`|@->`	`|root|@->|leaf`	*path*	no
`->@$`	`|root->@`	*full path*	yes
`->@|`	`|root->@|leaf`	*path*	yes
`->@->`	`|root->@->|leaf`	*full path*	yes

Note: The characters `^` and `$` used in the table respectively refer to the start and the end of a string. As for the character `@`, it is to be replaced by one or more wildcards.

Combining Wildcards

If needed, it is possible to come up with some fancy patterns by successively writing multiple wildcard operators that will combine to define a specific number of occurrences. For example, the pattern `...` matches 3 occurrences of the context element, while `.+` matches at least 2 occurrences, and `..??` matches from 2 to 4 occurrences.

The number of occurrences to match resulting from such a combination is easy to figure out. Let's consider the regular expression notation `{m,n}` describing `from m to n occurrences`, and `{m,}` that specifies `at least m occurrences`. Rewriting the four wildcard operators following this notation gives:

- `*` -> `{0,}`
- `+` -> `{1,}`
- `?` -> `{0,1}`
- `.` -> `{1,1}`

Combining wildcard operators is equivalent to adding their range of occurrences. From the previous example, the pattern `.+` equals to `{1,1} + {1,}`, that is `{2,}`, and the pattern `..??` equals to `{1,1} + {1,1} + {0,1} + {0,1}`, that is `{2,4}`.

Namespace Construct

The pattern `|root|.` allows matching any node which has `root` as direct parent but it is not enough if filtering namespaces is also required. This is why, as per the *syntax rules*, a special construct has been added to allow a full name to start with the `:` delimiter if it is followed by one or more wildcards. With this addition, the pattern `|root|:.` makes it possible to match any node directly parented under `root` that does not belong to any namespce.

Examples

Matching DG Nodes

```
>>> import bana
>>> bana.initialize()
>>> from maya import OpenMaya
>>> # Match the nodes named 'leaf' belonging to any namespace.
>>> OpenMaya.MGlobal.bnMatchFullName('*:leaf', 'leaf')
True
>>> OpenMaya.MGlobal.bnMatchFullName('*:leaf', 'ns:leaf')
True
>>> OpenMaya.MGlobal.bnMatchFullName('*:leaf', 'nsa:nsb:leaf')
True
>>> # Match the nodes directly nested under a namespace 'ns'.
>>> OpenMaya.MGlobal.bnMatchFullName('ns:.', 'ns:leaf')
True
>>> OpenMaya.MGlobal.bnMatchFullName('ns:.', 'ns:nsa:leaf')
False
>>> # Match the nodes recursively nested under a namespace 'ns'.
>>> OpenMaya.MGlobal.bnMatchFullName('ns:+', 'ns:leaf')
True
>>> OpenMaya.MGlobal.bnMatchFullName('ns:+', 'ns:nsa:leaf')
True
>>> OpenMaya.MGlobal.bnMatchFullName('ns:+', 'ns:nsa:nsb:leaf')
True
```

Matching DAG Nodes

```
>>> import bana
>>> bana.initialize()
>>> from maya import OpenMaya
>>> # Match the nodes directly parented under the world.
>>> OpenMaya.MGlobal.bnMatchPath('|.', '|leaf')
True
>>> OpenMaya.MGlobal.bnMatchPath('|.', '|ns:leaf')
True
>>> OpenMaya.MGlobal.bnMatchPath('|.', '|root|leaf')
False
>>> # Match the nodes directly parented under the world but not belonging to
... # any namespace.
>>> OpenMaya.MGlobal.bnMatchPath('|:.', '|leaf')
```

```
True
>>> OpenMaya.MGlobal.bnMatchPath('|:.', '|ns:leaf')
False
>>> OpenMaya.MGlobal.bnMatchPath('|:.', '|root|leaf')
False
>>> # Match the nodes containing 'Shape' anywhere in the hierarchy but not
... # belonging to any namespace.
>>> OpenMaya.MGlobal.bnMatchPath('+|*Shape*', '|cube|cubeShape')
True
>>> OpenMaya.MGlobal.bnMatchPath('+|*Shape*', '|root|sphere|sphereShape1')
True
>>> OpenMaya.MGlobal.bnMatchPath('+|*Shape*', '|cube|ns:cubeShape')
False
>>> # Match the nodes containing 'Shape' anywhere in the hierarchy.
>>> OpenMaya.MGlobal.bnMatchPath('+|*:*Shape*', '|cube|cubeShape')
True
>>> OpenMaya.MGlobal.bnMatchPath('+|*:*Shape*', '|root|sphere|sphereShape1')
True
>>> OpenMaya.MGlobal.bnMatchPath('+|*:*Shape*', '|cube|ns:cubeShape')
True
```

Retrieving Nodes

Out of the box, the Maya API is a bit cumbersome when it comes to retrieving DG and DAG nodes from a scene. This usually leads each TD to write their own code for the task, and this is also something that Bana aims to provide.

The goal here is to offer a higher-level set of methods allowing to retrieve nodes with enough flexibility to cover most of a TD's needs while remaining as fast as possible.

Since these methods are in fact iterators, it is easy to build on top of them in the case where more filtering options are required, such as for example skipping the DAG shapes that are templated.

But what sets this library apart from the usual implementations is its well-defined *pattern matching specification*. When Maya's interpretation of the wildcard character `*` is everyone's guess, Bana offers both precise and predictable results.

Design

In Bana, there are 2 groups of classes from where the scene nodes can be retrieved:

- the function set classes *`MFnDependencyNode`* and *`MFnDagNode`*.
- the lower-level classes *`MObject`* and *`MDagPath`*.

The former group represents the most common use case while the second can be used to slightly speed things up when the extra functionalities brought by the `MFn*` classes are not required.

For each of these classes, two types of methods are then exposed as the API:

- the methods starting with `bnFind` which return an iterator over a collection of nodes matching the input filters.
- the methods starting with `bnGet` which return a single object matching the input filters. If zero or more nodes are found, then `None` is returned.

When using the lower-level family of classes, it is possible to explictely pass a node type to match through the `fnType` parameter but in the case of the function set classes, no `fnType` parameter is defined. Instead the node type to match is deduced from the calling class. In other words, a call to `maya.OpenMaya.MFnDagNode.bnFind()` will match

any node of type `maya.OpenMaya.kDagNode` while calling `maya.OpenMaya.MFnTransform.bnFind()` will only match transform nodes.

Note: Node names are available from the `maya.OpenMaya.MFnDependencyNode` class but not directly from the `maya.OpenMaya.MObject` one. As a result, retrieving `maya.OpenMaya.MObject` objects from a pattern will internally convert them to `maya.OpenMaya.MFnDependencyNode`, in which case there won't be much benefits from using the *MObject.bnFind()* method in place of *MFnDependencyNode.bnFind()*.

DG vs DAG Nodes

Although it is possible to iterate over all DG or DAG nodes using the methods exposed within the *MFnDependencyNode* and *MObject* classes, it is not possible to filter DAG nodes this way using a *path* pattern. Indeed, these methods only accept *name* patterns.

Hence it is recommended to use instead the methods defined in the classes *MFnDagNode* and *MDagPath* whenever DAG nodes are to be retrieved. Furthermore, these offer a boost in performances, especially when only a specific branch of DAG nodes needs to be traversed through the use of the `bnFindChildren()` and `bnGetChild()` methods.

Examples

```
>>> import bana
>>> bana.initialize()
>>> from maya import OpenMaya
>>> # Retrieve a transform node named 'root'.
>>> root = OpenMaya.MFnTransform.bnGet(pattern='*|root')
>>> # Recursively iterate over all the DAG nodes child of 'root'.
>>> for node in root.bnFindChildren():
...     print(node)
>>> # Find all the mesh nodes in the scene containing the word 'Shape' but
... # not belonging to any namespace.
>>> for node in OpenMaya.MFnMesh.bnFind(pattern='*|*Shape*'):
...     print(node)
```

Extension Categories

Each extension provided with Bana is written to answer a specific need belonging to one of these categories:

explicit Maya's implementation was deemed ambiguous possibly because of a lack of well-defined specification or documentation.

fix A specific method needs to be modified but creating a new method prefixed with `bn` isn't an option. Therefore, the original method is fixed in place by being replaced. This approach is only used for magic methods such as `__str__()` and `__hash__()`.

foundation The extension is considered as a fundamental functionality that is missing from Maya's API.

MScriptUtil The original method needs to be wrapped to abstract away the used of the `maya.OpenMaya.MScriptUtil` class. Some of these methods are marked as not implemented to document a better alternative approach.

no throw By default, exceptions are being thrown whenever a method returns a `maya.OpenMaya.MStatus` object with a value that is not `kSuccess`. This is not justified in cases where it is acceptable that the call to a method might or might not output a valid result. For example, it is expected for a `MFn*` class instance to fail accessing its `maya.OpenMaya.MObject` object if the function set hasn't been fully initialized yet, this doesn't have to be considered as an error. A better suited return value here is `None` since it carries the information that *no* valid object can be retrieved at the moment, while being even more convenient to check validity against.

Note: The category for a specific extension can be found in the documentation associated with that extension.

API Reference

All the extensions of Bana are described here.

Initialization

initialize	Initialize the extensions.

bana.**initialize**()
Initialize the extensions.

The patches from the Bana package are searched and applied to the Maya API. Patches that seem to have already been applied are skipped.

Extensions

All the patches to apply to the Maya API are listed here and are named after their destination class.

OpenMaya

OpenMaya.MDagPath

bnFind	DAG path iterator.
bnGet	Retrieve a single DAG path.
__hash__	Hash value that can be relied on.
__str__	Full path name.
bnFindChildren	DAG path iterator over the children.
bnGetChild	Retrieve a single DAG path child.
bnGetParent	Retrieve the parent DAG path.

classmethod MDagPath.**bnFind**(*pattern=None*, *fnType=maya.OpenMaya.MFn.kInvalid*, *recursive=True*, *traverseUnderWorld=True*, *copy=True*)
DAG path iterator.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG paths to match. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.
- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.
- **copy** (*bool*) – `True` to copy each DAG path. It is useful when data persistence is required, such as when the DAG paths are to be stored into a list, otherwise it is faster to set it to `False`.

Yields *maya.OpenMaya.MDagPath* – The paths found.

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

---

classmethod `MDagPath.`**bnGet**(*pattern=None*, *fnType=maya.OpenMaya.MFn.kInvalid*, *recursive=True*, *traverseUnderWorld=True*)

Retrieve a single DAG path.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG path to match. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.
- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.

Returns The DAG path found. If none or many were found, `None` is returned.

Return type maya.OpenMaya.MDagPath

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

`MDagPath.`**__hash__**()

Hash value that can be relied on.

This is required because the original method returns different values for multiple instances pointing to a same object, thus making the `MDagPath` object not usable with hash-based containers such as dictionaries and sets.

Categories: *fix*.

Returns The hash value representing this object.

Return type int

`MDagPath.__str__()`

Full path name.

It is helpful when interacting with the commands layer by not having to manually call the `fullPathName()` method each time a `MDagPath` object needs to be passed to a command.

Categories: *fix*.

Returns The full path name.

Return type str

`MDagPath.bnFindChildren(pattern=None, fnType=maya.OpenMaya.MFn.kInvalid, recursive=True, traverseUnderWorld=True, copy=True)`

DAG path iterator over the children.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG paths to match, relative to the current DAG path. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.
- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.
- **copy** (*bool*) – `True` to copy each DAG path. It is useful when data persistence is required, such as when the DAG paths are to be stored into a list, otherwise it is faster to set it to `False`.

Yields *maya.OpenMaya.MDagPath* – The paths found.

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

`MDagPath.bnGetChild(pattern=None, fnType=maya.OpenMaya.MFn.kInvalid, recursive=True, traverseUnderWorld=True)`

Retrieve a single DAG path child.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG path to match, relative to the current DAG path. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.

- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.

Returns The path found.

Return type maya.OpenMaya.MDagPath

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

`MDagPath.`**`bnGetParent`**`()`

Retrieve the parent DAG path.

Categories: *foundation*.

Returns The parent DAG path, or `None` if this DAG path is directly parented under the world.

Return type maya.OpenMaya.MDagPath or None

OpenMaya.MFnBase

bnObject	Retrieve the object attached to this function set.

`MFnBase.`**`bnObject`**`()`

Retrieve the object attached to this function set.

Categories: *no throw*.

Returns The object or `None` if no valid object is attached to this function set.

Return type maya.OpenMaya.MObject or None

OpenMaya.MFnDagNode

bnFind	DAG node iterator.
bnGet	Retrieve a single DAG node.
__str__	Full path name.
bnFindChildren	DAG node iterator over the children.
bnGetChild	Retrieve a single DAG node child.

---

classmethod `MFnDagNode.`**`bnFind`**`(`*pattern=None*, *recursive=True*, *traverseUnderWorld=True*`)`

DAG node iterator.

The calling class defines the function set type for which the nodes need to be compatible with. It also represents the type of the object returned.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG nodes to match. Wildcards are allowed.
- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.

Yields *maya.OpenMaya.MDagNode* – The nodes found.

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

---

classmethod `MFnDagNode.bnGet` (*pattern=None*, *recursive=True*, *traverseUnderWorld=True*)

Retrieve a single DAG node.

The calling class defines the function set type for which the node needs to be compatible with. It also represents the type of the object returned.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG node to match. Wildcards are allowed.
- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.

Returns The DAG node found. If none or many were found, `None` is returned.

Return type maya.OpenMaya.MFnDagNode

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

`MFnDagNode.__str__()`

Full path name.

It is helpful when interacting with the commands layer by not having to manually call the `fullPathName()` method each time a `MFnDagNode` object needs to be passed to a command.

Categories: *fix*.

Returns The full path name.

Return type str

MFnDagNode.**bnFindChildren**(*pattern=None*, *fnType=maya.OpenMaya.MFn.kInvalid*, *recursive=True*, *traverseUnderWorld=True*)

DAG node iterator over the children.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG nodes to match, relative to the current node. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.
- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.

Yields *maya.OpenMaya.MDagNode* – The nodes found.

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

MFnDagNode.**bnGetChild**(*pattern=None*, *fnType=maya.OpenMaya.MFn.kInvalid*, *recursive=True*, *traverseUnderWorld=True*)

Retrieve a single DAG node child.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – Path or full path pattern of the DAG nodes to match, relative to the current node. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.
- **recursive** (*bool*) – `True` to search recursively.
- **traverseUnderWorld** (*bool*) – `True` to search within the underworld.

Returns The node found.

Return type maya.OpenMaya.MDagNode

Note: The pattern matching's global context is set to *full path* if the parameter `traverseUnderWorld` is `True`, and to *path* otherwise. See *Matching Rules*.

See also:

Pattern Matching, *Retrieving Nodes*

OpenMaya.MFnDependencyNode

bnFind	DG node iterator.
bnGet	Retrieve a single DG node.
__hash__	Hash value that can be relied on.
__str__	Name.

---

classmethod MFnDependencyNode.**bnFind**(*pattern=None*)

DG node iterator.

The calling class defines the function set type for which the nodes need to be compatible with. It also represents the type of the objects yielded.

Categories: *foundation*.

Parameters **pattern** (*str*) – Full name pattern of the DG nodes to match. Wildcards are allowed.

Yields *cls* – The DG nodes found.

See also:

Pattern Matching, *Retrieving Nodes*

---

classmethod MFnDependencyNode.**bnGet**(*pattern=None*)

Retrieve a single DG node.

The calling class defines the function set type for which the node needs to be compatible with. It also represents the type of the object returned.

Categories: *foundation*.

Parameters **pattern** (*str*) – Full name pattern of the DG node to match. Wildcards are allowed.

Returns The DG node found. If none or many were found, None is returned.

Return type cls

See also:

Pattern Matching, *Retrieving Nodes*

MFnDependencyNode.**__hash__**()

Hash value that can be relied on.

This is required because the original method returns different values for multiple instances pointing to a same object, thus making the MFnDependencyNode object not usable with hash-based containers such as dictionaries and sets.

Categories: *fix*.

Returns The hash value representing this object.

Return type int

MFnDependencyNode.**__str__**()
Name.

It is helpful when interacting with the commands layer by not having to manually call the `name()` method each time a `MFnDependencyNode` object needs to be passed to a command.

Categories: *fix*.

Returns The name.

Return type str

OpenMaya.MFnTransform

bnGetScale	Retrieve the scale component.
bnSetScale	Set the scale component.
bnScaleBy	Add to the scale component by scaling relatively.
bnGetShear	Retrieve the shear component.
bnSetShear	Set the shear component.
bnShearBy	Add to the shear component by shearing relatively.

MFnTransform.**bnGetScale**()
Retrieve the scale component.

Categories: *MScriptUtil*.

Returns The scale component.

Return type list [x, y, z]

MFnTransform.**bnSetScale**(*scale*)
Set the scale component.

Categories: *MScriptUtil*.

Parameters **scale** (*sequence of 3 floats*) – New scale component.

MFnTransform.**bnScaleBy**(*scale*)
Add to the scale component by scaling relatively.

Categories: *MScriptUtil*.

Parameters **scale** (*sequence of 3 floats*) – Relative value to scale by.

MFnTransform.**bnGetShear**()
Retrieve the shear component.

Categories: *MScriptUtil*.

Returns The shear component.

Return type list [x, y, z]

MFnTransform.**bnSetShear**(*shear*)
Set the shear component.

Categories: *MScriptUtil*.

Parameters **shear** (*sequence of 3 floats*) – New shear component.

MFnTransform.**bnShearBy**(*shear*)
Add to the shear component by shearing relatively.

Categories: *MScriptUtil*.

Parameters **shear** (*sequence of 3 floats*) – Relative value to shear by.

OpenMaya.MGlobal

bnIsValidName	Check if a *name* is strictly well-formed.
bnIsValidFullName	Check if a *full name* is strictly well-formed.
bnIsValidPath	Check if a *path* is strictly well-formed.
bnIsValidFullPath	Check if a *full path* is strictly well-formed.
bnMakeMatchNameFunction	Create a function to match *names* to a pattern.
bnMakeMatchFullNameFunction	Create a function to match *full names* to a pattern.
bnMakeMatchPathFunction	Create a function to match *paths* to a pattern.
bnMakeMatchFullPathFunction	Create a function to match *full paths* to a pattern.
bnMatchName	Check if a *name* matches a given pattern.
bnMatchFullName	Check if a *full name* matches a given pattern.
bnMatchPath	Check if a *path* matches a given pattern.
bnMatchFullPath	Check if a *full path* matches a given pattern.

---

classmethod MGlobal.**bnIsValidName**(*name*, *allowWildcards=False*)
Check if a *name* is strictly well-formed.

Names can identify DG nodes, excluding the ones carrying any namespace or hierarchy information. They are made of *character* elements, that is alphanumeric characters, underscores, and wildcards.

Categories: *explicit*.

Parameters

- **path** (*str*) – *Name* to check.
- **allowWildcards** (*bool*) – `True` to consider the wildcards as valid characters.

Returns `True` if the *name* is strictly well-formed.

Return type bool

See also:

Pattern Matching

---

classmethod MGlobal.**bnIsValidFullName**(*name*, *allowWildcards=False*, *matchRelative=False*)
Check if a *full name* is strictly well-formed.

Full names can fully identify any DG node. They are composed by one or more *name* elements, each separated by the namespace delimiter `:`.

Categories: *explicit*.

Parameters

- **path** (*str*) – *Full name* to check.
- **allowWildcards** (*bool*) – `True` to consider the wildcards as valid characters.
- **matchRelative** (*bool*) – `True` to allow matching relatively to a parent namespace. That is, *full names* starting with the namespace delimiter `:` are allowed.

Returns `True` if the *full name* is strictly well-formed.

Return type bool

See also:

Pattern Matching

---

classmethod `MGlobal.`**bnIsValidPath**(*path*, *allowWildcards=False*)

Check if a *path* is strictly well-formed.

Paths can identify DAG nodes, excluding the ones carrying any underworld information. They are composed by one or more *full name* elements, each starting with the hierarchy delimiter `|`.

Categories: *explicit*.

Parameters

- **path** (*str*) – *Path* to check.
- **allowWildcards** (*bool*) – `True` to consider the wildcards as valid characters.

Returns `True` if the *path* is strictly well-formed.

Return type bool

See also:

Pattern Matching

---

classmethod `MGlobal.`**bnIsValidFullPath**(*path*, *allowWildcards=False*, *matchRelative=False*)

Check if a *full path* is strictly well-formed.

Full paths can fully identify any DAG node. They are composed by one or more *path* elements, each separated by the underworld delimiter `->`.

Categories: *explicit*.

Parameters

- **path** (*str*) – *Full path* to check.
- **allowWildcards** (*bool*) – `True` to consider the wildcards as valid characters.
- **matchRelative** (*bool*) – `True` to allow matching relatively to a parent path. That is, *full paths* starting with the underworld delimiter `->` are allowed.

Returns `True` if the *full path* is strictly well-formed.

Return type bool

See also:

Pattern Matching

---

classmethod `MGlobal.`**bnMakeMatchNameFunction**(*pattern*)

Create a function to match *names* to a pattern.

Categories: *explicit*.

Parameters **pattern** (*str*) – *Name* pattern to build. Wildcards are allowed.

Returns A function expecting a single parameter, that is the *name* to check the pattern against. The return value of this function is a value that evaluates to `True` or `False` in a boolean operation. The value passed to the parameter of this function must be a strictly well-formed *name*. No check is done to ensure the validity of the input but this can be done manually using *MGlobal.bnIsValidName()*.

Return type function

Raises `ValueError` – The pattern is not well-formed.

Examples

```
>>> import bana
>>> bana.initialize()
>>> from maya import OpenMaya
>>> iterator = OpenMaya.MItDependencyNodes()
>>> match = OpenMaya.MGlobal.bnMakeMatchNameFunction('*Shape*')
>>> while not iterator.isDone():
...     obj = iterator.thisNode()
...     name = OpenMaya.MFnDependencyNode(obj).name()
...     if match(name):
...         print(name)
...     iterator.next()
```

See also:

Pattern Matching, *MGlobal.bnIsValidName()*

---

classmethod `MGlobal.`**bnMakeMatchFullNameFunction**(*pattern*, *matchRelative=False*)

Create a function to match *full names* to a pattern.

Categories: *explicit*.

Parameters

- **pattern** (*str*) – Full name pattern to build. Wildcards are allowed.
- **matchRelative** (*bool*) – `True` to allow matching relatively to a parent namespace. That is, *full names* starting with the namespace delimiter `:` are allowed.

Returns A function expecting a single parameter, that is the *full name* to check the pattern against. The return value of this function is a value that evaluates to `True` or `False` in a boolean operation. The value passed to the parameter of this function must be a strictly well-formed *full name*. No check is done to ensure the validity of the input but this can be done manually using *MGlobal.bnIsValidFullName()*.

Return type function

Raises `ValueError` – The pattern is not well-formed.

See also:

Pattern Matching, *MGlobal.bnIsValidFullName()*

---

classmethod `MGlobal.`**bnMakeMatchPathFunction**(*pattern*)

Create a function to match *paths* to a pattern.

Categories: *explicit*.

Parameters **pattern** (*str*) – *Path* pattern to build. Wildcards are allowed.

Returns A function expecting a single parameter, that is the *path* to check the pattern against. The return value of this function is a value that evaluates to `True` or `False` in a boolean operation. The value passed to the parameter of this function must be a strictly well-formed *path*. No check is done to ensure the validity of the input but this can be done manually using *MGlobal.bnIsValidPath()*.

Return type function

Raises `ValueError` – The pattern is not well-formed.

Examples

```
>>> import bana
>>> bana.initialize()
>>> from maya import OpenMaya
>>> iterator = OpenMaya.MItDag()
>>> match = OpenMaya.MGlobal.bnMakeMatchPathFunction('*|*Shape*')
>>> dagPath = OpenMaya.MDagPath()
>>> while not iterator.isDone():
...     iterator.getPath(dagPath)
...     path = dagPath.fullPathName()
...     if match(path):
...         print(path)
...     iterator.next()
```

See also:

Pattern Matching, *MGlobal.bnIsValidPath()*

---

classmethod `MGlobal.`**bnMakeMatchFullPathFunction**(*pattern*, *matchRelative=False*)

Create a function to match *full paths* to a pattern.

Categories: *explicit*.

Parameters

- **pattern** (*str*) – *Full path* pattern to build. Wildcards are allowed.
- **matchRelative** (*bool*) – `True` to allow matching relatively to a parent path. That is, *full paths* starting with the underworld delimiter `->` are allowed.

Returns A function expecting a single parameter, that is the *full path* to check the pattern against. The return value of this function is a value that evaluates to `True` or `False` in a boolean operation. The value passed to the parameter of this function must be a strictly well-formed *full path*. No check is done to ensure the validity of the input but this can be done manually using *MGlobal.bnIsValidFullPath()*.

Return type function

Raises `ValueError` – The pattern is not well-formed.

See also:

Pattern Matching, *MGlobal.bnIsValidFullPath()*

---

classmethod `MGlobal.`**`bnMatchName`**(*pattern*, *name*)

Check if a *name* matches a given pattern.

Both pattern and *name* must be strictly well-formed.

If the same pattern is to be matched several times, consider using *MGlobal.bnMakeMatchNameFunction()* instead.

Categories: *explicit*.

Parameters

- **`pattern`** (*str*) – Pattern to match to. Wildcards are allowed.
- **`path`** (*str*) – *Name* to check.

Returns `True` if the *name* matches the given pattern.

Return type bool

See also:

Pattern Matching, *MGlobal.bnIsValidName()*

---

classmethod `MGlobal.`**`bnMatchFullName`**(*pattern*, *name*, *matchRelative=False*)

Check if a *full name* matches a given pattern.

Both pattern and *full name* must be strictly well-formed.

If the same pattern is to be matched several times, consider using *MGlobal.bnMakeMatchFullNameFunction()* instead.

Categories: *explicit*.

Parameters

- **`pattern`** (*str*) – Pattern to match to. Wildcards are allowed.
- **`path`** (*str*) – *Full name* to check.
- **`matchRelative`** (*bool*) – `True` to allow matching relatively to a parent namespace. That is, *full names* starting with the namespace delimiter `:` are allowed.

Returns `True` if the *full name* matches the given pattern.

Return type bool

See also:

Pattern Matching, *MGlobal.bnIsValidFullName()*

---

classmethod MGlobal.**bnMatchPath**(*pattern*, *path*)
Check if a *path* matches a given pattern.

Both pattern and *path* must be strictly well-formed.

If the same pattern is to be matched several times, consider using *MGlobal.bnMakeMatchPathFunction()* instead.

Categories: *explicit*.

Parameters

- **pattern** (*str*) – Pattern to match to. Wildcards are allowed.
- **path** (*str*) – *Path* to check.

Returns True if the *path* matches the given pattern.

Return type bool

See also:

Pattern Matching, *MGlobal.bnIsValidPath()*

---

classmethod MGlobal.**bnMatchFullPath**(*pattern*, *path*, *matchRelative=False*)
Check if a *full path* matches a given pattern.

Both pattern and *full path* must be strictly well-formed.

If the same pattern is to be matched several times, consider using *MGlobal.bnMakeMatchFullPathFunction()* instead.

Categories: *explicit*.

Parameters

- **pattern** (*str*) – Pattern to match to. Wildcards are allowed.
- **path** (*str*) – *Full path* to check.
- **matchRelative** (*bool*) – True to allow matching relatively to a parent path. That is, *full paths* starting with the underworld delimiter -> are allowed.

Returns True if the *full path* matches the given pattern.

Return type bool

See also:

Pattern Matching, *MGlobal.bnIsValidFullPath()*

OpenMaya.MMatrix

__str__	Printable-friendly version of the values.
bnGet	Retrieve the values as a two-dimensional 4 x 4 list.

MMatrix.**__str__**()
Printable-friendly version of the values.

Categories: *fix*.

Returns A printable-friendly version of the values.

Return type str

MMatrix.**bnGet**()
Retrieve the values as a two-dimensional 4 x 4 list.

Categories: *MScriptUtil*.

Returns The two-dimensional 4 x 4 list of values.

Return type list of list of floats

OpenMaya.MObject

bnFind	DG node iterator.
bnGet	Retrieve a single DG node.
__hash__	Hash value that can be relied on.

---

classmethod MObject.**bnFind**(*pattern=None*, *fnType=maya.OpenMaya.MFn.kInvalid*)
DG node iterator.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – *Full name* pattern of the DG nodes to match. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.

Yields *maya.OpenMaya.MObject* – The DG nodes found.

See also:

Pattern Matching, *Retrieving Nodes*

---

classmethod MObject.**bnGet**(*pattern=None*, *fnType=maya.OpenMaya.MFn.kInvalid*)
Retrieve a single DG node.

Categories: *foundation*.

Parameters

- **pattern** (*str*) – *Full name* pattern of the DG node to match. Wildcards are allowed.
- **fnType** (*maya.OpenMaya.MFn.Type*) – Function set type to match.

Returns The DG node found. If none or many were found, None is returned.

Return type maya.OpenMaya.MObject

See also:

Pattern Matching, *Retrieving Nodes*

MObject.**__hash__**()
Hash value that can be relied on.

This is required because the original method returns different values for multiple instances pointing to a same object, thus making the MObject object not usable with hash-based containers such as dictionaries and sets.

Categories: *fix*.

Returns The hash value representing this object.

Return type int

OpenMaya.MPoint

__str__	Printable-friendly version of the values.
bnGet	Retrieve the values as a list.

MPoint.**__str__**()
Printable-friendly version of the values.

Categories: *fix*.

Returns A printable-friendly version of the values.

Return type str

MPoint.**bnGet**()
Retrieve the values as a list.

Categories: *MScriptUtil*.

Returns The values.

Return type list [x, y, z, w]

OpenMaya.MQuaternion

__str__	Printable-friendly version of the values.
bnGet	Retrieve the values as a list.

MQuaternion.**__str__**()
Printable-friendly version of the values.

Categories: *fix*.

Returns A printable-friendly version of the values.

Return type str

MQuaternion.**bnGet**()
Retrieve the values as a list.

Categories: *MScriptUtil*.

Returns The values.

Return type list [x, y, z, w]

OpenMaya.MTransformationMatrix

bnAddRotation	Not implemented.
bnGetRotation	Not implemented.
bnSetRotation	Not implemented.
bnAddRotationQuaternion	Not implemented.
bnGetRotationQuaternion	Not implemented.
bnSetRotationQuaternion	Not implemented.
bnAddScale	Add to the scale component by scaling relatively.
bnGetScale	Retrieve the scale component.
bnSetScale	Set the scale component.
bnAddShear	Add to the shear component by shearing relatively.
bnGetShear	Retrieve the shear component.
bnSetShear	Set the shear component.

MTransformationMatrix.**bnAddRotation**(*rotation*, *order=maya.OpenMaya.MTransformationMatrix.kXYZ*, *space=maya.OpenMaya.MSpace.kTransform*)

Not implemented. See the examples for an alternative approach.

Categories: *MScriptUtil*.

Examples

Alternative approach:

```
>>> from maya import OpenMaya
>>> transform = OpenMaya.MFnTransform()
>>> transform.create()
>>> xform = transform.transformation()
>>> rotation = OpenMaya.MEulerRotation(1.0, 2.0, 3.0,
...                                    OpenMaya.MEulerRotation.kXYZ)
>>> xform.rotateBy(rotation, OpenMaya.MSpace.kTransform)
>>> transform.set(xform)
```

MTransformationMatrix.**bnGetRotation**(*space=maya.OpenMaya.MSpace.kTransform*)

Not implemented. See the examples for an alternative approach.

Categories: *MScriptUtil*.

Examples

Alternative approach:

```
>>> from maya import OpenMaya
>>> transform = OpenMaya.MFnTransform()
>>> transform.create()
>>> rotation = transform.transformation().eulerRotation()
>>> [rotation.x, rotation.y, rotation.z]
[0.0, 0.0, 0.0]
>>> rotation.order
0
```

MTransformationMatrix.**bnSetRotation**(*rotation*, *order=maya.OpenMaya.MTransformationMatrix.kXYZ*, *space=maya.OpenMaya.MSpace.kTransform*)

Not implemented. See the examples for an alternative approach.

Categories: *MScriptUtil*.

Examples

Alternative approach:

```
>>> from maya import OpenMaya
>>> transform = OpenMaya.MFnTransform()
>>> transform.create()
>>> xform = transform.transformation()
>>> rotation = OpenMaya.MEulerRotation(1.0, 2.0, 3.0,
...                                    OpenMaya.MEulerRotation.kXYZ)
>>> xform.rotateTo(rotation)
>>> transform.set(xform)
```

MTransformationMatrix.**bnAddRotationQuaternion**(*rotation*, *space=maya.OpenMaya.MSpace.kTransform*)

Not implemented. See the examples for an alternative approach.

Categories: *MScriptUtil*.

Examples

Alternative approach:

```
>>> from maya import OpenMaya
>>> transform = OpenMaya.MFnTransform()
>>> transform.create()
>>> xform = transform.transformation()
>>> rotation = OpenMaya.MQuaternion(1.0, 2.0, 3.0, 4.0)
>>> xform.rotateBy(rotation, OpenMaya.MSpace.kTransform)
>>> transform.set(xform)
```

MTransformationMatrix.**bnGetRotationQuaternion**(*space=maya.OpenMaya.MSpace.kTransform*)

Not implemented. See the examples for an alternative approach.

Categories: *MScriptUtil*.

Examples

Alternative approach:

```
>>> from maya import OpenMaya
>>> transform = OpenMaya.MFnTransform()
>>> transform.create()
>>> rotation = transform.transformation().rotation()
>>> [rotation.x, rotation.y, rotation.z, rotation.w]
[0.0, 0.0, 0.0, 1.0]
```

MTransformationMatrix.**bnSetRotationQuaternion**(*rotation*, *order=maya.OpenMaya.MTransformationMatrix.kXYZ*, *space=maya.OpenMaya.MSpace.kTransform*)

Not implemented. See the examples for an alternative approach.

Categories: *MScriptUtil*.

Examples

Alternative approach:

```
>>> from maya import OpenMaya
>>> transform = OpenMaya.MFnTransform()
>>> transform.create()
>>> xform = transform.transformation()
>>> rotation = OpenMaya.MQuaternion(1.0, 2.0, 3.0, 4.0)
>>> xform.rotateTo(rotation)
>>> transform.set(xform)
```

MTransformationMatrix.**bnAddScale**(*scale*, *space=maya.OpenMaya.MSpace.kTransform*)

Add to the scale component by scaling relatively.

Categories: *MScriptUtil*.

Parameters

- **scale** (*sequence of 3 floats*) – Relative value to scale by.
- **space** (*maya.OpenMaya.MSpace.Space*) – Transform space in which to perform the scale.

MTransformationMatrix.**bnGetScale**(*space=maya.OpenMaya.MSpace.kTransform*)

Retrieve the scale component.

Categories: *MScriptUtil*.

Parameters **space** (*maya.OpenMaya.MSpace.Space*) – Transform space in which to get the scale.

Returns The scale component.

Return type list [x, y, z]

MTransformationMatrix.**bnSetScale**(*scale*, *space=maya.OpenMaya.MSpace.kTransform*)
Set the scale component.

Categories: *MScriptUtil*.

Parameters

- **scale** (*sequence of 3 floats*) – New scale component.
- **space** (*maya.OpenMaya.MSpace.Space*) – Transform space in which to set the scale.

MTransformationMatrix.**bnAddShear**(*shear*, *space=maya.OpenMaya.MSpace.kTransform*)
Add to the shear component by shearing relatively.

Categories: *MScriptUtil*.

Parameters

- **shear** (*sequence of 3 floats*) – Relative value to shear by.
- **space** (*maya.OpenMaya.MSpace.Space*) – Transform space in which to perform the shear.

MTransformationMatrix.**bnGetShear**(*space=maya.OpenMaya.MSpace.kTransform*)
Retrieve the shear component.

Categories: *MScriptUtil*.

Parameters **space** (*maya.OpenMaya.MSpace.Space*) – Transform space in which to get the shear.

Returns The shear component.

Return type list [x, y, z]

MTransformationMatrix.**bnSetShear**(*shear*, *space=maya.OpenMaya.MSpace.kTransform*)
Set the shear component.

Categories: *MScriptUtil*.

Parameters

- **shear** (*sequence of 3 floats*) – New shear component.
- **space** (*maya.OpenMaya.MSpace.Space*) – Transform space in which to set the shear.

OpenMaya.MVector

__str__	Printable-friendly version of the values.
bnGet	Retrieve the values as a list.
bnRotateBy	Rotate the vector.

MVector.**__str__**()
Printable-friendly version of the values.

Categories: *fix*.

Returns A printable-friendly version of the values.

Return type str

MVector.**bnGet**()
Retrieve the values as a list.

Categories: *MScriptUtil*.

Returns The values.

Return type list [x, y, z]

MVector.**bnRotateBy**(*rotation*, *order=maya.OpenMaya.MTransformationMatrix.kXYZ*)
Rotate the vector.

Categories: *MScriptUtil*.

Parameters

- **rotation** (*sequence of 3 floats*) – Values in radian to rotate by.
- **order** (*maya.OpenMaya.MTransformationMatrix.RotationOrder*) – Rotation order.

Returns The new vector.

Return type maya.OpenMaya.MVector

CHAPTER 2

Developer's Guide

Running the Tests

After making any code change in Bana, tests need to be evaluated to ensure that the library still behaves as expected.

Note: Some of the commands below are wrapped into `make` targets for convenience, see the file `Makefile`.

unittest

The tests are written using Python's built-in `unittest` module. They are available in the `tests` directory and can be fired through the `tests/run.py` file:

```
$ mayapy tests/run.py
```

It is possible to run specific tests by passing a space-separated list of partial names to match:

```
$ mayapy tests/run.py ThisTestClass and_that_function
```

The `unittest`'s command line interface is also supported:

```
$ mayapy -m unittest discover -s tests -v
```

Finally, each test file is a **standalone** and can be directly executed.

coverage

The package `coverage` is used to help localize code snippets that could benefit from having some more testing:

```
$ mayapy -m coverage run --source bana -m unittest discover -s tests
$ coverage report
$ coverage html
```

In no way should `coverage` be a race to the 100% mark since it is *not* always meaningful to cover each single line of code. Furthermore, **having some code fully covered isn't synonym to having quality tests**. This is our responsability, as developers, to write each test properly regardless of the coverage status.

Benchmarks

A set of benchmarks are also available to keep the running performances in check. They are to be found in the `benchmarks` folder and can be run in a similar fashion to the tests through the `benchmarks/run.py` file:

```
$ mayapy benchmarks/run.py
```

Or for more specificity:

```
$ mayapy benchmarks/run.py ThisBenchClass and_that_function
```

Here again, each benchmark file is a **standalone** and can be directly executed.

Note: The command line interface `mayapy -m unittest discover` is not supported for the benchmarks.

CHAPTER 3

Additional Information

Changelog

Version numbers comply with the Sementic Versioning Specification (SemVer).

Unreleased

- Make minor tweaks to the code.

v0.1.0 (2017-01-11)

Changed

- Rewrite everything from scratch. Changes are not backwards compatible.

v0.0.3 (2014-12-07)

Added

- Add a `__hash__()` method for the `MDagPath`, `MFnDependencyNode`, and `MObject` classes.

Changed

- Make minor tweaks to the code.

v0.0.2 (2014-06-22)

Added

- Add a `bnn_asFunctionSet()` method to the `MObject` and `MDagPath` classes.
- Add a `bnn_getFunctionSet()` method to the `MObject` class.
- Add a new internal module for data caching.

Changed

- Prepend the missing `bnn` identifiers for the test routines.
- Make minor tweaks to the code.

v0.0.1 (2014-06-21)

- Initial release.

Versioning

Version numbers comply with the Sementic Versioning Specification (SemVer).

In summary, version numbers are written in the form `MAJOR.MINOR.PATCH` where:

- incompatible API changes increment the MAJOR version.
- functionalities added in a backwards-compatible manner increment the MINOR version.
- backwards-compatible bug fixes increment the PATCH version.

Major version zero (0.y.z) is considered a special case denoting an initial development phase. Anything may change at any time without the MAJOR version being incremented.

License

Python Module Index

b

Index

Symbols

B

E

F

I

M

N