
NodeCalculator Documentation

Release 2.0.0

Mischa Kolbe

Aug 19, 2020

Contents:

1	About	3
2	Download	5
3	Tutorials	7
3.1	Tutorial: Introduction	7
3.2	Tutorial: Basics	8
3.3	Tutorial: Math and Operators	10
3.4	Tutorial: Convenient Extras	11
3.5	Tutorial: Tracer	13
3.6	Tutorial: Customize It	14
3.6.1	Main	14
3.6.2	Appendix	15
3.7	Tutorial: Under the Hood	16
3.8	Tutorial: Examples	17
3.8.1	Example: Soft Approach Value	17
3.8.2	Example: Simple cogs	19
3.8.3	Example: Stepping cogs	20
3.8.4	Example: Dynamic colors	21
4	Code	25
4.1	Config	25
4.2	Core	26
4.3	Operators	57
4.4	Extension	79
4.5	NcValue	81
4.6	Tracer	84
4.7	OmUtil	85
4.8	Logging	92
5	Changes	95
5.1	Release 2.1.5	95
5.1.1	Features added	95
5.1.2	Bugs fixed	95
5.2	Release 2.1.4	96
5.2.1	Bugs fixed	96
5.3	Release 2.1.3	96

5.3.1	Features added	96
5.3.2	Bugs fixed	96
5.4	Release 2.1.2	96
5.4.1	Features added	96
5.4.2	Bugs fixed	97
5.5	Release 2.1.1	97
5.5.1	Bugs fixed	97
5.6	Release 2.1.0	97
5.6.1	Incompatible changes	97
5.6.2	Features added	97
5.6.3	Bugs fixed	98
5.7	Release 2.0.1	98
5.7.1	Bugs fixed	98
5.8	Release 2.0.0	98
5.8.1	Dependencies	98
5.8.2	Incompatible changes	98
5.8.3	Deprecated	98
5.8.4	Features added	98
5.8.5	Bugs fixed	99
5.8.6	Testing	99
5.8.7	Features removed	99
5.9	Release 1.0.0	99
6	Overview	101
	Python Module Index	103
	Index	105

The NodeCalculator is an OpenSource Python module that allows you to create node networks in Autodesk Maya by writing math formulas.

CHAPTER 1

About

The NodeCalculator is an OpenSource Python module that allows you to *create node networks in Autodesk Maya by writing math formulas*.

The idea for this tool originated from the incredibly tedious process of translating a math formula into Maya nodes. Often times the mathematical logic is simple and could be written down in a few seconds. But then you find yourself spending the next 30 minutes translating that formula into Maya commands. The NodeCalculator tries to speed up and ease that process.

I hope this tool brings the fun & beauty of math back to the Maya TDs out there ;)

Cheers, [Mischa](#).

CHAPTER 2

Download

Get the Python scripts from GitHub: [NodeCalculator](#)

Get the NodeCalculator cheat sheet: `NodeCalculator cheat sheet`

The following videos should give you a good overview how to use the NodeCalculator.

Note: If you're unsure whether this tool is interesting for you: Watch an *example video* first. You might not understand the details, but you'll see what the NodeCalculator allows you to do.

I recommend to watch the videos in order. They go into more and more detail, so if you feel you know enough to work with the NodeCalculator: Feel free to stop. When you need to understand the tool better, you can come back for the remaining, more in-depth videos.

3.1 Tutorial: Introduction

Listing 1: Code used in video:

```
# INTRODUCTION

import node_calculator.core as noca

# Initiate Node-objects for all test-geos
a_geo = noca.Node("A_geo")
b_geo = noca.Node("B_geo")
c_geo = noca.Node("C_geo")

# Average all directions of the A-translation
translate_a_average = noca.Op.average(a_geo.tx, a_geo.ty, a_geo.tz)
# Create a "floor collider" based on the height of B
b_height_condition = noca.Op.condition(b_geo.ty > 0, b_geo.ty, 0)

# Drive C-translation by different example-formula for each direction
c_geo.translate = [b_geo.tx / 2 - 2, b_height_condition * 2, translate_a_average]
```

(continues on next page)

(continued from previous page)

```

# ~~~ VS ~~~

from maya import cmds

# Average all directions of the A-translation
var1 = cmds.createNode("plusMinusAverage", name="A_translate_average")
cmds.setAttr(var1 + ".operation", 3)
cmds.connectAttr("A_geo.tx", var1 + ".input3D[0].input3Dx", force=True)
cmds.connectAttr("A_geo.ty", var1 + ".input3D[1].input3Dx", force=True)
cmds.connectAttr("A_geo.tz", var1 + ".input3D[2].input3Dx", force=True)

# Create a "floor collider" based on the height of B
var2 = cmds.createNode("condition", name="height_condition")
cmds.setAttr(var2 + ".operation", 2)
cmds.connectAttr("B_geo.ty", var2 + ".firstTerm", force=True)
cmds.setAttr(var2 + ".secondTerm", 0)
cmds.connectAttr("B_geo.ty", var2 + ".colorIfTrueR", force=True)
cmds.setAttr(var2 + ".colorIfFalseR", 0)

# Drive C-translation by different example-formula for each direction
var3 = cmds.createNode("multiplyDivide", name="half_B_tx")
cmds.setAttr(var3 + ".operation", 2)
cmds.connectAttr("B_geo.tx", var3 + ".input1X", force=True)
cmds.setAttr(var3 + ".input2X", 2)
var4 = cmds.createNode("plusMinusAverage", name="offset_half_b_tx_by_2")
cmds.setAttr(var4 + ".operation", 2)
cmds.connectAttr(var3 + ".outputX", var4 + ".input3D[0].input3Dx", force=True)
cmds.setAttr(var4 + ".input3D[1].input3Dx", 2)
var5 = cmds.createNode("multiplyDivide", name="double_height_condition")
cmds.setAttr(var5 + ".operation", 1)
cmds.connectAttr(var2 + ".outColorR", var5 + ".input1X", force=True)
cmds.setAttr(var5 + ".input2X", 2)
cmds.connectAttr(var4 + ".output3Dx", "C_geo.translateX", force=True)
cmds.connectAttr(var5 + ".outputX", "C_geo.translateY", force=True)
cmds.connectAttr(var1 + ".output3Dx", "C_geo.translateZ", force=True)

```

3.2 Tutorial: Basics

Listing 2: Code used in video:

```

# Tab1
# BASICS

import node_calculator.core as noca

# Valid Node instantiations
a = noca.Node("A_geo")
a = noca.Node("A_geo.tx")
a = noca.Node("A_geo", ["ty", "tz", "tx"])
a = noca.Node("A_geo", attrs=["ty", "tz", "tx"])
a = noca.Node(["A_geo.ty", "B_geo.tz", "A_geo.tx"])

```

(continues on next page)

(continued from previous page)

```

# Numbers and lists work as well
a = noca.Node(7)
a = noca.Node([1, 2, 3])

# Created NcNode has a node-& attrs-part
a = noca.Node("A_geo.tx")
print(a)

# Tab2
# BASICS

import node_calculator.core as noca

# Valid Node instantiations
a = noca.Node("A_geo")

# Attribute setting
a.tx = 7
a.translateX = 6
a.tx.set(3)
a.t = 5
a.t = [1, 2, 3]

# Even if an attribute was specified at initialization...
a = noca.Node("A_geo.ty")
# ...any attribute can be set!
a.sx = 2

# Any attribute works
a.tx = 12
a.translateX = 12
a.visibility = 0
a.thisIsMyAttr = 6

# Tab3
# BASICS

import node_calculator.core as noca

# Caution!
bad = noca.Node("A_geo.tx")
bad = 2.5
good = noca.Node("A_geo")
good.tx = 2.5

a = noca.Node("A_geo.tx")
a.attrs = 3.5
a = noca.Node("A_geo", ["tx", "ty"])
a.attrs = 1
a.attrs = [2, 3]

# Tab4
# BASICS

```

(continues on next page)

(continued from previous page)

```
import node_calculator.core as noca

# Attribute query
print(a.ty.get())
print(a.t.get())

# Tab5
# BASICS

import node_calculator.core as noca

# Attribute connection
b = noca.Node("B_geo")
a.translateX = b.scaleY
```

3.3 Tutorial: Math and Operators

Listing 3: Code used in video:

```
# Tab1
# MATH

import node_calculator.core as noca

a = noca.Node("A_geo")
b = noca.Node("B_geo")
c = noca.Node("C_geo")

# Simple math operation
a.tx + 2

# Nodes are setup automatically
a.tx - 2

# Multiple math operations
a.tx * 2 + 3

# Assignment of math operation result
c.ty = a.tx * 2 + 3

# Tab2
# MATH

import node_calculator.core as noca

a = noca.Node("A_geo")
b = noca.Node("B_geo")
c = noca.Node("C_geo")

# Correct order of operations. Thanks, Python!
c.ty = a.tx + b.ty / 2
c.ty = (a.tx + b.ty) / 2
```

(continues on next page)

(continued from previous page)

```

# Works with 1D, 2D & 3D. Mixes well with 1D attrs!
c.t = b.t * a.t - [1, 2, 3]
c.t = b.t * a.ty

# Intermediate results ("bad" isn't necessarily bad)
a_third = a.t / 3
c.t = a_third

# Tab3
# MATH

import node_calculator.core as noca

a = noca.Node("A_geo")
b = noca.Node("B_geo")
c = noca.Node("C_geo")

# Additional operations via Op-class
c.tx = noca.Op.length(a.t, b.t)

# Available Operators
help(noca.Op)
noc.a.Op.available(full=False)
help(noca.Op.length)

# Conditionals are fun
c.t = noca.Op.condition(2 - b.ty > 3, b.t, [0, 3, 0])

# Warning: It's Python!
# Or: "The reason for Nodes of values/lists".
c.t = [a.ty, a.tx, 2] - 5
c.t = [a.ty, a.tx, 2] * [1, 4, b.tx]

```

3.4 Tutorial: Convenient Extras

Listing 4: Code used in video:

```

# Tab1
# CONVENIENT EXTRAS

import node_calculator.core as noca

a = noca.Node("A_geo")
b = noca.Node("B_geo")
c = noca.Node("C_geo")

# Keywords (see cheatSheet!)
add_result = a.t + [1, 2, 3]
print(add_result.node)
print(add_result.attrs)
print(add_result.attrs_list)
print(add_result.plugs)

```

(continues on next page)

(continued from previous page)

```

noca_list = noca.Node([a.tx, c.tx, b.tz])
print(noca_list.nodes)

# Tab2
# CONVENIENT EXTRAS

import node_calculator.core as noca

# Create nodes as NcNode instances
my_transform = noca.transform("myTransform")
my_locator = noca.locator("myLocator")
my_xy = noca.create_node("nurbsCurve", "myCurve")

# Add attributes
my_transform.add_separator()
offset = my_transform.add_float("offsetValue", min=0, max=3)
space_switch = my_transform.add_enum("spaceSwitch", cases=["local", "world"])
my_locator.t = noca.Op.condition(
    space_switch == 0,
    my_transform.ty * 2 - offset,
    0
)

# Tab3
# CONVENIENT EXTRAS

import node_calculator.core as noca

# NcNodes as iterables
some_node = noca.Node("A_geo", ["tx", "ty"])
for index, item in enumerate(some_node):
    print(type(item), item)
    item.attrs = index

# Attributes can be accessed via index
some_node[1].attrs = 7

# Work around issue of array-attributes
plus_minus = noca.create_node(
    "plusMinusAverage",
    name="test",
    attrs=["input3D[0].input3Dx"]
)
plus_minus.attrs = 3

# Tab4
# CONVENIENT EXTRAS

import node_calculator.core as noca

# Convert to PyNode
my_locator = noca.locator("myLocator", attrs=["tx", "ty", "tz"])
pm_my_locator = my_locator.to_py_node()

```

(continues on next page)

(continued from previous page)

```

pm_my_locator = my_locator.to_py_node(ignore_attrs=True)
pm_my_locator = my_locator[1].to_py_node()

# Tab5
# CONVENIENT EXTRAS

import node_calculator.core as noca

a = noca.Node("A_geo")
b = noca.Node("B_geo")

# auto_unravel & auto_consolidate
a = noca.Node("A_geo", auto_unravel=False, auto_consolidate=False)
b = noca.Node("B_geo", auto_unravel=True, auto_consolidate=True)
a.t = b.tx

nocas.set_global_auto_unravel(False)
nocas.set_global_auto_consolidate(False)

```

3.5 Tutorial: Tracer

Listing 5: Code used in video:

```

# Tab1
# TRACER

import node_calculator.core as noca

a = noca.Node("A_geo")
b = noca.Node("B_geo")
c = noca.Node("C_geo")

# This is our lovely formula, but it needs to be faster!
c.ty = a.tx + b.ty / 2

# Tab2
# TRACER
with noca.Tracer() as trace:
    c.ty = a.tx + b.ty / 2
print(trace)

with noca.Tracer(pprint_trace=True):
    current_val = a.tx.get()
    offset_val = (current_val - 1) / 3
    c.ty = a.tx + b.ty / offset_val

```

3.6 Tutorial: Customize It

3.6.1 Main

Listing 6: Code used in video:

```
# CUSTOMIZE IT

# example_extension.py:
from node_calculator.core import noca_op
from node_calculator.core import _create_operation_node

REQUIRED_EXTENSION_PLUGINS = ["lookdevKit"]

EXTENSION_OPERATORS = {
    "color_math_min": {
        "node": "colorMath",
        "inputs": [
            ["colorAR", "colorAG", "colorAB"],
            ["alphaA"],
            ["colorBR", "colorBG", "colorBB"],
            ["alphaB"],
        ],
        "outputs": [
            ["outColorR", "outColorG", "outColorB"],
            # ["outAlpha"], ?
        ],
        "operation": 4,
    },
}

@noc_a_op
def color_math_min(color_a, alpha_a=0.25, color_b=(0, 0, 1), alpha_b=0.75):
    created_node = _create_operation_node(
        'color_math_min', color_a, alpha_a, color_b, alpha_b
    )
    return created_node

# In Maya:

# CUSTOMIZE IT
import node_calculator.core as noca

# Initiate Node-objects for all test-geos
a_geo = noca.Node("A_geo")
b_geo = noca.Node("B_geo")
c_geo = noca.Node("C_geo")

c_geo.tx = noca.Op.color_math_min(
    color_a=a_geo.tx,
    alpha_a=0.5,
    color_b=b_geo.ty,
    alpha_b=0.5,
)
```

3.6.2 Appendix

Listing 7: Code used in video:

```
# CUSTOMIZE IT

# example_extension.py:
from node_calculator.core import noca_op
from node_calculator.core import _create_operation_node

REQUIRED_EXTENSION_PLUGINS = ["lookdevKit"]

EXTENSION_OPERATORS = {
    "color_math_min": {
        "node": "colorMath",
        "inputs": [
            ["colorAR", "colorAG", "colorAB"],
            ["alphaA"],
            ["colorBR", "colorBG", "colorBB"],
            ["alphaB"],
        ],
        "outputs": [
            ["outColorR", "outColorG", "outColorB"],
            # ["outAlpha"], ?
        ],
        "operation": 4,
    },
}

@noca_op
def color_math_min(color_a, alpha_a=0.25, color_b=(0, 0, 1), alpha_b=0.75):
    created_node = _create_operation_node(
        'color_math_min', color_a, alpha_a, color_b, alpha_b
    )
    return created_node

# In Maya:

# CUSTOMIZE IT
import node_calculator.core as noca

# Initiate Node-objects for all test-geos
a_geo = noca.Node("A_geo")
b_geo = noca.Node("B_geo")
c_geo = noca.Node("C_geo")

c_geo.tx = noca.Op.color_math_min(
    color_a=a_geo.tx,
    alpha_a=0.5,
    color_b=b_geo.ty,
    alpha_b=0.5,
```

(continues on next page)

(continued from previous page)

)

3.7 Tutorial: Under the Hood

Listing 8: Code used in video:

```

# Tab1
# UNDER THE HOOD

import node_calculator.core as noca

_node = noca.Node("A_geo")
print('Type of noca.Node("A_geo"):', type(_node))

_attr = noca.Node("A_geo").tx
print('Type of noca.Node("A_geo").tx:', type(_attr))

_node_list = noca.Node(["A_geo", "B_geo"])
print('Type of noca.Node(["A_geo", "B_geo"]):', type(_node_list))

_int = noca.Node(7)
print('Type of noca.Node(7):', type(_int))

_float = noca.Node(1.2)
print('Type of noca.Node(1.2):', type(_float))

_list = noca.Node([1, 2, 3])
print('Type of noca.Node([1, 2, 3]):', type(_list))

# Tab2
# UNDER THE HOOD

# NcNode vs NcAttrs
nc_node = noca.Node("A_geo.translateX")

nc_attr = nc_node.scaleY
nc_attr.extra
# or:
nc_node.scaleY.extra

plus_minus = noca.create_node("plusMinusAverage", "test")
plus_minus = noca.Node(plus_minus, "input3D[0]")
plus_minus.attrs.input3Dx = 8

# Tab3
# UNDER THE HOOD

# NcValues
# This does NOT work:
normal_int = 1

```

(continues on next page)

(continued from previous page)

```

normal_int.marco = "polo"

# This works:
noca_int = noca.Node(1)
noca_int.marco = "polo"
noca_int.metadata
noca_int.created_by_user

# Keeping track of origin of values:
from maya import cmds
scale = cmds.getAttr("A_geo.scaleX")
translate = cmds.getAttr("A_geo.translateX")
print(scale.metadata)
print(translate.metadata)
# vs
a_geo = noca.Node("A_geo")
with noca.Tracer():
    scale = a_geo.scaleX.get()
    translate = a_geo.translateX.get()

    a_geo.tx = scale + translate

print(scale.metadata)
print(translate.metadata)

```

3.8 Tutorial: Examples

3.8.1 Example: Soft Approach Value

Listing 9: Code used in video:

```

# EXAMPLE: soft_approach_value

import node_calculator.core as noca

"""Task:
Approach a target value slowly, once the input value is getting close to it.
"""

"""
# Sudo-Code based on Harry Houghton's video: youtube.com/watch?v=xS1LpHE14Uk
in_value = <inputValue>
fade_in_range = <fadeInRange>
target_value = <targetValue>

if (in_value > (target_value - fade_in_range)):
    if (fade_in_range > 0):
        exponent = -(in_value - (target_value - fade_in_range)) / fade_in_range
        result = target_value - fade_in_range * exp(exponent)
    else:
        result = target_value
else:
    result = in_value

```

(continues on next page)

(continued from previous page)

```

driven.attr = result
"""

import math

driver = noca.Node("driver")
in_value = driver.tx
driven = noca.Node("driven.tx")
fade_in_range = driver.add_float("transitionRange", value=1)
target_value = driver.add_float("targetValue", value=5)

# Note: Factoring the leading minus sign into the parenthesis requires one node
# less. I didn't do so to maintain the similarity to Harry's example.
# However; I'm using the optimized version in noca.Op.soft_approach()
exponent = -(in_value - (target_value - fade_in_range)) / fade_in_range
soft_approach_value = target_value - fade_in_range * math.e ** exponent

is_range_valid_condition = noca.Op.condition(
    fade_in_range > 0,
    soft_approach_value,
    target_value
)

is_in_range_condition = noca.Op.condition(
    in_value > (target_value - fade_in_range),
    is_range_valid_condition,
    in_value
)

driven.attrs = is_in_range_condition

# NOTE: This setup is now a standard operator: noca.Op.soft_approach()

```

Listing 10: Code used in video (example extension):

```

# DON'T import node_calculator.core as noca! It's a cyclical import that fails!
from node_calculator.core import noca_op
from node_calculator.core import Op

# ~~~~~ STEP 1: REQUIRED PLUGINS ~~~~~
REQUIRED_EXTENSION_PLUGINS = []

# ~~~~~ STEP 2: OPERATORS DICTIONARY ~~~~~
EXTENSION_OPERATORS = {}

# ~~~~~ STEP 3: OPERATOR FUNCTION ~~~~~
@noca_op
def soft_approach(in_value, fade_in_range=0.5, target_value=1):
    """Follow in_value, but approach the target_value slowly.

    Note:
        Only works for 1D inputs!
    """

```

(continues on next page)

(continued from previous page)

```

Args:
    in_value (NcNode or NcAttrs or str or int or float): Value or attr
    fade_in_range (NcNode or NcAttrs or str or int or float): Value or
        attr. This defines a range over which the target_value will be
        approached. Before the in_value is within this range the output
        of this and the in_value will be equal.
    target_value (NcNode or NcAttrs or str or int or float): Value or
        attr. This is the value that will be approached slowly.

Returns:
    NcNode: Instance with node and output-attr.

Example:
    ::

        in_attr = Node("pCube.tx")
        Op.soft_approach(in_attr, fade_in_range=2, target_value=5)
        # Starting at the value 3 (because 5-2=3), the output of this
        # will slowly approach the target_value 5.
    """
    start_val = target_value - fade_in_range

    exponent = ((start_val) - in_value) / fade_in_range
    soft_approach_value = target_value - fade_in_range * Op.exp(exponent)

    is_range_valid_condition = Op.condition(
        fade_in_range > 0,
        soft_approach_value,
        target_value
    )

    is_in_range_condition = Op.condition(
        in_value > start_val,
        is_range_valid_condition,
        in_value
    )

    return is_in_range_condition

```

3.8.2 Example: Simple cogs

Listing 11: Code used in video:

```

# EXAMPLE: cogs_simple

import node_calculator.core as noca

"""Task:
Drive all cogs by an attribute on the ctrl.
"""

```

(continues on next page)

(continued from previous page)

```
# Solution 1:
# Direct drive rotation
ctrl = noca.Node("ctrl")
gear_5 = noca.Node("gear_5_geo")
gear_7 = noca.Node("gear_7_geo")
gear_8 = noca.Node("gear_8_geo")
gear_17 = noca.Node("gear_17_geo")
gear_22 = noca.Node("gear_22_geo")

driver = ctrl.add_float("cogRotation") * 10

gear_22.ry = driver / 22.0
gear_5.ry = driver / -5.0
gear_7.ry = driver / 7.0
gear_8.ry = driver / -8.0
gear_17.ry = driver / 17.0

# Solution 2:
# Chained rotation:
ctrl = noca.Node("ctrl")
gear_5 = noca.Node("gear_5_geo")
gear_7 = noca.Node("gear_7_geo")
gear_8 = noca.Node("gear_8_geo")
gear_17 = noca.Node("gear_17_geo")
gear_22 = noca.Node("gear_22_geo")

driver = ctrl.add_float("cogRotation", min=0)

gear_22.ry = driver
gear_8.ry = gear_22.ry * (-22/8.0)
gear_7.ry = gear_8.ry * (-8/7.0)
gear_5.ry = gear_7.ry * (-7/5.0)
gear_17.ry = gear_5.ry * (-5/17.0)
```

3.8.3 Example: Stepping cogs

Listing 12: Code used in video:

```
# EXAMPLE: cogs_stepping

import node_calculator.core as noca

"""Task:
Drive all cogs by an attribute on the ctrl. All teeth but one were removed from
one of the cogs(!)

Uses maya_math_nodes extension!
```

(continues on next page)

(continued from previous page)

```

"""

# Initialize nodes.
main_rot = noca.Node("ctrl.cogRot") * 100
cog_5 = noca.Node("gear_5_geo")
cog_7 = noca.Node("gear_7_geo")
cog_8 = noca.Node("gear_8_geo")
cog_22 = noca.Node("gear_22_geo")
cog_17 = noca.Node("gear_17_geo")

# Basic cog-rotation of small and big cog.
cog_5.ry = main_rot / 5.0
cog_7.ry = main_rot / -7.0
cog_22.ry = main_rot / -22.0

# Make rotation positive (cog_22 increases in negative direction).
single_tooth_rot = - cog_22.ry

# Dividing single_tooth_rot by 360deg and ceiling gives number of steps.
step_count = noca.Op.floor(noca.Op.divide(single_tooth_rot, 360))

single_tooth_degrees_per_step = 360 / 8.0
receiving_degrees_per_step = single_tooth_degrees_per_step / 17.0 * 8
stepped_receiving_rot = step_count * receiving_degrees_per_step
single_tooth_live_step_rotation = noca.Op.modulus_int(single_tooth_rot, single_tooth_
↳degrees_per_step)
receiving_live_step_rotation = single_tooth_live_step_rotation / 17.0 * 8
rot_offset = 1

cog_17.ry = noca.Op.condition(
    # At every turn of the single tooth gear: Actively rotate the receiving gear_
↳during degrees_per_step degrees (45deg).
    noca.Op.modulus_int(single_tooth_rot, 360) < single_tooth_degrees_per_step,
    # When live rotation is happening the current step rotation is added to the_
↳accumulated stepped rotation.
    stepped_receiving_rot + receiving_live_step_rotation + rot_offset,
    # Static rotation if single tooth gear isn't driving. Needs an extra step since_
↳step_count is floored.
    stepped_receiving_rot + receiving_degrees_per_step + rot_offset
)

cog_8.ry = cog_17.ry / -8.0 * 17

```

3.8.4 Example: Dynamic colors

Note: No video (yet)!

Listing 13: Code used in video:

```
# EXAMPLE: Dynamic colors

import node_calculator.core as noca

"""Task:
Drive color based on translate x, y, z values:
The further into the x/y/z plane: The more r/g/b.
Values below zero/threshold should default to black.
"""

# Easy, but incomplete, due to: minus * minus = positive
b = noca.Node("geo")
b_mat = noca.Node("mat")
multiplier = b.add_float("multiplier", value=0.25, max=0.5)
r_value = b.ty * b.tz * multiplier
g_value = b.tx * b.tz * multiplier
b_value = b.tx * b.ty * multiplier
b_mat.color = [r_value, g_value, b_value]

#####

# Prototype red first.
b = noca.Node("geo")
b_mat = noca.Node("mat")
multiplier = b.add_float("multiplier", value=0.25, max=0.5)
r_value = b.ty * b.tz * multiplier
b_mat.colorR = noca.Op.condition(b.ty > 0, noca.Op.condition(b.tz > 0, r_value, 0), 0)

#####

# Doesn't display correctly in viewport!
b = noca.Node("geo")
b_mat = noca.Node("mat")

multiplier = b.add_float("multiplier", value=0.25, max=0.5)
threshold = 1

tx_zeroed = b.tx - threshold
ty_zeroed = b.ty - threshold
tz_zeroed = b.tz - threshold

r_value = ty_zeroed * tz_zeroed * multiplier
g_value = tx_zeroed * tz_zeroed * multiplier
b_value = tx_zeroed * ty_zeroed * multiplier

black = 0
with noca.Tracer(pprint_trace=True):
    b_mat.color = [
        noca.Op.condition(b.ty > threshold, noca.Op.condition(b.tz > threshold, r_
↪value, black), black),
        noca.Op.condition(b.tz > threshold, noca.Op.condition(b.tx > threshold, g_
↪value, black), black),
        noca.Op.condition(b.tx > threshold, noca.Op.condition(b.ty > threshold, b_
↪value, black), black),
```

(continues on next page)

(continued from previous page)

1

These pages are auto-created from the docStrings of the NodeCalculator files.

Note: For completeness ALL functions & methods are listed in this documentation. When using the NodeCalculator in Maya you should only use the *public* functions & methods!

You won't have to deal with functions & methods in Maya that are...

- *protected*. They start with one underscore: `_some_protected_function()`.
- *private*. They start and end with two underscores: `__some_private_function__()`.

4.1 Config

The config.py file allows you to globally change the default behaviour of the NodeCalculator.

```
"""Basic NodeCalculator settings."""

# Node preferences ---
NODE_PREFIX = "nc" # Name prefix for all nodes created by the NodeCalculator.

# Attribute preferences ---
DEFAULT_SEPARATOR_NAME = "_____" # Default NiceName for separator-attributes.
DEFAULT_SEPARATOR_VALUE = "_____" # Default value for separator-attributes.
DEFAULT_ATTR_FLAGS = { # Defaults for add_float(), add_enum(), ... attribute_
    ↪creation.
    "keyable": True,
}
```

(continues on next page)

(continued from previous page)

```
# Connection preferences ---
GLOBAL_AUTO_CONSOLIDATE = True # Reduce plugs to parent plug, if possible.
GLOBAL_AUTO_UNRAVEL = True # Expand plugs into their child components. I recommend_
↳ using True!

# Tracer preferences ---
VARIABLE_PREFIX = "var" # Prefix for variables in the Tracer-stack (created nodes).
VALUE_PREFIX = "val" # Prefix for values in the Tracer-stack (queried values).

# Extension preferences ---
EXTENSION_PATH = "" # Without a path the NodeCalculator will check for the_
↳ extension(s) locally.
# All extension files must live in the same location!
EXTENSION_NAMES = [] # Names of the extension python files (without .py).
```

4.2 Core

This is the main NodeCalculator file!

Create a node-network by entering a math-formula.

author Mischa Kolbe <mischakolbe@gmail.com>

credits Mischa Kolbe, Steven Bills, Marco D’Ambros, Benoit Gielly, Adam Vanner, Niels Kleinheinz,
Andres Weber

version 2.1.2

Note: In any comment/docString of the NodeCalculator I use this convention:

- node: Name of a Maya node in the scene (dagPath if name isn’t unique)
- attr/attribute: Attribute on a Maya node in the scene
- plug: Combination of node and attribute; node.attr

NcNode and NcAttrs instances provide these keywords:

- attrs: Returns currently stored NcAttrs of this NcNode instance.
- attrs_list: Returns list of stored attrs: [attr, ...] (list of strings).
- node: Returns name of Maya node in scene (str).
- plugs: Returns list of stored plugs: [node.attr, ...] (list of strings).

NcList instances provide these keywords:

- nodes: Returns Maya nodes inside NcList: [node, ...] (list of strings)

Supported operations:

```
# Basic math
+, -, *, /, **
```

(continues on next page)

(continued from previous page)

```
# To see the available Operators, use:
Op.available()
# Or to see all Operators and their full docString:
Op.available(full=True)
```

Example

```
import node_calculator.core as noca

a = noca.Node("pCube1")
b = noca.Node("pCube2")
c = noca.Node("pCube3")

with noca.Tracer(pprint_trace=True):
    e = b.add_float("someAttr", value=c.tx)
    a.s = noca.Op.condition(b.ty - 2 > c.tz, e, [1, 2, 3])
```

class `core.NcAttrs` (*holder_node*, *attrs*)

NcAttrs are linked to an NcNode instance & represent attrs on Maya node.

Note: Getting attr X from an NcAttrs that holds attr Y returns: NcAttrs.Y.X In contrast; NcNode instances do NOT “concatenate” attrs: Getting attr X from an NcNode that holds attr Y only returns: NcNode.X

__getattr__ (*name*)

Get a new NcAttrs instance with the requested attribute.

Note: The requested attr gets “concatenated” onto the existing attr(s)!

There are certain keywords that will NOT return a new NcAttrs:

- `attrs`: Returns this NcAttrs instance (self).
 - `attrs_list`: Returns stored attrs: [attr, ...] (list of strings).
 - `node`: Returns name of Maya node in scene (str).
 - `nodes`: Returns name of Maya node in scene in a list ([str]).
 - `plugins`: Returns stored plugs: [node.attr, ...] (list of strings).
-

Parameters `name` (*str*) – Name of requested attribute

Returns New NcAttrs instance OR self, if keyword “attrs” was used!

Return type *NcAttrs*

Example

```
a = Node("pCube1") # Create new NcNode-object
a.tx.ty # invokes __getattr__ on NcNode "a" first, which
         returns an NcAttrs instance with node: "a" & attrs:
         "tx". The __getattr__ described here then acts on
```

(continues on next page)

(continued from previous page)

```
the retrieved NcAttrs instance and returns a new
NcAttrs instance. This time with node: "a" & attrs:
"tx.ty"!
```

__getitem__ (*index*)

Get stored attribute at given index.

Note: This looks through the list of stored attributes.

Parameters **index** (*int*) – Index of desired item**Returns** New NcNode instance, solely with attribute at index.**Return type** *NcNode*

__init__ (*holder_node, attrs*)

Initialize NcAttrs-class instance.

Note: `__setattr__` is altered. The usual “self.node=node” results in loop! Therefore attributes need to be set a bit awkwardly via `__dict__`!

Parameters

- **holder_node** (*NcNode*) – Represents a Maya node
- **attrs** (*str or list or NcAttrs*) – Represents attrs on the Maya node

__holder_node

NcNode instance this NcAttrs belongs to.

Type *NcNode***__held_attrs_list**

Strings that represent attrs on Maya node.

Type list**Raises** `TypeError` – If the holder_node isn't of type NcNode.

__auto_consolidateGet `__auto_consolidate` attribute of `__holder_node`.**Returns** Whether auto consolidating is allowed**Return type** bool

__auto_unravelGet `__auto_unravel` attribute of `__holder_node`.**Returns** Whether auto unravelling is allowed**Return type** bool

__node_mobj

Get the MObject this NcAttrs instance refers to.

Note: MObject is stored on the NcNode this NcAttrs instance refers to!

Returns MObject instance of Maya node in the scene

Return type MObject

attrs

Get this NcAttrs instance.

Returns NcAttrs instance that represents Maya attributes.

Return type *NcAttrs*

attrs_list

Get list of stored attributes of this NcAttrs instance.

Returns List of strings that represent Maya attributes.

Return type list

node

Get name of the Maya node this NcAttrs is linked to.

Returns Name of Maya node in the scene.

Return type str

class core.NcBaseClass

Base class for NcLists & NcBaseNode (hence indirectly NcNode & NcAttrs).

Note: NcNode, NcAttrs and NcList are the “building blocks” of NodeCalculator calculations. Having NcBaseClass as their common parent class makes sure the overloaded operators apply to each of these “building blocks”.

__add__ (other)

Regular addition operator for NodeCalculator objects.

Example

```
Node("pCube1.ty") + 4
```

__div__ (other)

Regular division operator for NodeCalculator objects.

Example

```
Node("pCube1.ty") / 4
```

__eq__ (other)

Equality operator for NodeCalculator objects.

Returns Instance of a newly created Maya condition-node

Return type *NcNode*

Example

```
Node("pCube1.ty") == 4
```

`__ge__` (*other*)

Greater equal operator for NodeCalculator objects.

Returns Instance of a newly created Maya condition-node

Return type *NcNode*

Example

```
Node("pCube1.ty") >= 4
```

`__gt__` (*other*)

Greater than operator for NodeCalculator objects.

Returns Instance of a newly created Maya condition-node

Return type *NcNode*

Example

```
Node("pCube1.ty") > 4
```

`__init__` ()

Initialize NcBaseClass instance.

`__le__` (*other*)

Less equal operator for NodeCalculator objects.

Returns Instance of a newly created Maya condition-node

Return type *NcNode*

Example

```
Node("pCube1.ty") <= 4
```

`__lt__` (*other*)

Less than operator for NodeCalculator objects.

Returns Instance of a newly created Maya condition-node

Return type *NcNode*

Example

```
Node("pCube1.ty") < 4
```

`__mul__` (*other*)

Regular multiplication operator for NodeCalculator objects.

Example

```
Node("pCube1.ty") * 4
```

__ne__ (*other*)

Inequality operator for NodeCalculator objects.

Returns Instance of a newly created Maya condition-node

Return type *NcNode*

Example

```
Node("pCube1.ty") != 4
```

__neg__ ()

Leading minus sign multiplies by -1.

Example

```
- Node("pCube1.ty")
```

__pos__ ()

Leading plus signs are ignored, since they are redundant.

Example

```
+ Node("pCube1.ty")
```

__pow__ (*other*)

Regular power operator for NodeCalculator objects.

Example

```
Node("pCube1.ty") ** 4
```

__radd__ (*other*)

Reflected addition operator for NodeCalculator objects.

Note: Fall-back method if regular addition is not defined/fails.

Example

```
4 + Node("pCube1.ty")
```

__rdiv__ (*other*)

Reflected division operator for NodeCalculator objects.

Note: Fall-back method if regular division is not defined/fails.

Example

```
4 / Node ("pCube1.ty")
```

`__rmul__` (*other*)

Reflected multiplication operator for NodeCalculator objects.

Note: Fall-back method if regular multiplication is not defined/fails.

Example

```
4 * Node ("pCube1.ty")
```

`__rpow__` (*other*)

Reflected power operator for NodeCalculator objects.

Example

```
4 ** Node ("pCube1.ty")
```

`__rsub__` (*other*)

Reflected subtraction operator for NodeCalculator objects.

Note: Fall-back method if regular subtraction is not defined/fails.

Example

```
4 - Node ("pCube1.ty")
```

`__sub__` (*other*)

Regular subtraction operator for NodeCalculator objects.

Example

```
Node ("pCube1.ty") - 4
```

`__weakref__`

list of weak references to the object (if defined)

classmethod `_add_to_command_stack` (*command*)

Add a command to the class-variable `_executed_commands_stack`.

Parameters `command` (*str or list*) – String or list of strings of Maya command(s)

classmethod `_add_to_traced_nodes` (*node*)

Add a node to the class-variable `_traced_nodes`.

Parameters *node* (`TracerMObject`) – MObject with metadata. Check docString of TracerMObject for more detail!

classmethod `_add_to_traced_values` (*value*)

Add a value to the class-variable `_traced_values`.

Parameters *value* (`NcValue`) – Value with metadata. Check docString of NcValue.

__compare (*other, operator*)

Create a Maya condition node, set to the correct operation-type.

Parameters

- **other** (`NcNode` or `int` or `float`) – Compare self-attrs with other
- **operator** (`string`) – Operation type available in Maya condition-nodes

Returns Instance of a newly created Maya condition-node

Return type `NcNode`

classmethod `_flush_command_stack` ()

Reset class-variable `_executed_commands_stack` to an empty list.

classmethod `_flush_traced_nodes` ()

Reset class-variable `_traced_nodes` to an empty list.

classmethod `_flush_traced_values` ()

Reset class-variable `_traced_values` to an empty list.

classmethod `_get_next_value_name` ()

Return the next available value name.

Note: When Tracer is active, queried values get a value name assigned.

Returns Next available value name.

Return type `str`

classmethod `_get_next_variable_name` ()

Return the next available variable name.

Note: When Tracer is active, created nodes get a variable name assigned.

Returns Next available variable name.

Return type `str`

classmethod `_get_tracer_variable_for_node` (*node*)

Try to find and return traced variable for given node.

Parameters *node* (`str`) – Name of Maya node

Returns

If there is a traced variable for this node: Return the variable, otherwise return `None`

Return type str or None

classmethod `_initialize_trace_variables()`

Reset all class variables used for tracing.

class `core.NcBaseNode`

Base class for NcNode and NcAttrs.

Note: This class will have access to the `.node` and `.attrs` attributes, once it is instantiated in the form of a NcNode or NcAttrs instance.

__init__()

Initialize of NcBaseNode class, which is used for NcNode & NcAttrs.

Note: For more detail about `auto_unravel` & `auto_consolidate` check `docString` of `set_global_auto_consolidate` & `set_global_auto_unravel!`

Parameters

- **auto_unravel** (*bool*) – Should attrs of this instance be unravelled.
- **auto_consolidate** (*bool*) – Should instance-attrs be consolidated.

__iter__()

Iterate over list of attributes.

Yields *NcNode* – Next item in list of attributes.

Raises `StopIteration` – If end of `.attrs_list` is reached.

__len__()

Return the length of the stored attributes list.

Returns Length of stored NcAttrs list. 0 if no Attrs are defined.

Return type int

__repr__()

Print unambiguous format of NcBaseNode instance.

Note: For example invoked by running highlighted code in Maya.

Returns String of concatenated class-type, node and attrs.

Return type str

__setattr__ (*name, value*)

Set or connect attribute to the given value.

Note: Attribute setting works the same way for NcNode and NcAttrs instances. Their difference lies within the `__getattr__` method.

`setattr` is invoked by equal-sign. Does NOT work without attr:

`a = Node("pCube1.ty")` # Initialize Node-object with attr given

a.ty = 7 # Works fine if attribute is specifically called

a = 7 # Does NOT work!

It looks like the same operation as above, but here Python calls the assignment operation, NOT setattr. The assignment operation can't be overridden.

Parameters

- **name** (*str*) – Name of the attribute to be set
- **value** (*NcNode or NcAttrs or str or int or float or list or tuple*) – Connect attr to this object or set attr to this value/array

Example

```
a = Node("pCube1") # Create new NcNode-object
a.tx = 7 # Set pCube1.tx to the value 7
a.t = [1, 2, 3] # Set pCube1.tx/ty/tz to 1/2/3 respectively
a.tx = Node("pCube2").ty # Connect pCube2.ty to pCube1.tx
```

__getitem__ (*index, value*)

Set or connect attribute at index to the given value.

Note: Item setting works the same way for NcNode and NcAttrs instances. Their difference lies within the `__getitem__` method.

This looks at the list of attrs stored inside NcAttrs.

Parameters

- **index** (*int*) – Index of item to be set
- **value** (*NcNode or NcAttrs or str or int or float*) – Set/connect item at index to this.

__str__ ()

Print readable format of NcBaseNode instance.

Note: For example invoked by using `print()` in Maya.

Returns String of concatenated node and attrs.

Return type str

__add_all_add_attr_methods ()

Add all possible attribute types for `add_XYZ()` methods via closure.

Note: Allows to add attributes, similar to `addAttr`-command.

Example

```
Node("pCube1").add_float("my_float_attr", defaultValue=1.1)
Node("pCube1").add_short("my_int_attr", keyable=False)
```

`_add_traced_attr` (*attr_name*, ***kwargs*)

Create a Maya-attribute on the Maya-node this NcBaseNode refers to.

Parameters

- **`attr_name`** (*str*) – Name of new attribute.
- **`kwargs`** (*dict*) – Any user specified flags & their values. Gets combined with values in `DEFAULT_ATTR_FLAGS`!

Returns NcNode instance with the newly created attribute.

Return type *NcNode*

`_define_add_attr_method` (*attr_type*, *default_data_type*)

Closure to add `add_XYZ()` methods.

Note: Check `docString` of `_add_all_add_attr_methods`.

Parameters

- **`attr_type`** (*str*) – Name of data type of this attr: `bool`, `long`, ...
- **`default_data_type`** (*str*) – Either “attributeType” or “dataType”. See Maya docs for more info.

Returns Function that will be added to class methods.

Return type `executable`

`add_enum` (*name*, *enum_name*=”, *cases*=None, ***kwargs*)

Create an enum-attribute with given name and kwargs.

Note: kwargs are exactly the same as in `cmds.addAttr()`!

Parameters

- **`name`** (*str*) – Name for the new attribute to be created.
- **`enum_name`** (*list or str*) – User-choices for the resulting enum-attr.
- **`cases`** (*list or str*) – Overrides `enum_name`, which is a horrific name.
- **`kwargs`** (*dict*) – User specified flags to be set for the new attr.

Returns NcNode-instance with the node and new attribute.

Return type *NcNode*

Example

```
Node("pCube1").add_enum(cases=["A", "B", "C"], value=2)
```


add_int (*args, **kwargs)

Create an integer-attribute on the node associated with this NcNode.

Note: This function simply redirects to add_long, but most people will probably expect an “int” data type.

Parameters

- **args** (*list*) – Arguments that will be passed on to add_long()
- **kwargs** (*dict*) – Key/value pairs that will be passed on to add_long()

Returns NcNode-instance with the node and new attribute.

Return type *NcNode*

add_separator (name=<sphinx.ext.autodoc.importer._MockObject object>, enum_name=<sphinx.ext.autodoc.importer._MockObject object>, cases=None, **kwargs)

Create a separator-attribute.

Note: Default name and enum_name are defined by the globals DEFAULT_SEPARATOR_NAME and DEFAULT_SEPARATOR_VALUE! kwargs are exactly the same as in cmds.addAttr()!

Parameters

- **name** (*str*) – Name for the new separator to be created.
- **enum_name** (*list or str*) – User-choices for the resulting enum-attr.
- **cases** (*list or str*) – Overrides enum_name, which is a horrific name.
- **kwargs** (*dict*) – User specified flags to be set for the new attr.

Returns NcNode-instance with the node and new attribute.

Return type *NcNode*

Example

```
Node("pCube1").add_separator()
```

attr (attr=None)

Get new NcNode instance with given attr (using keywords is allowed).

Note: It is pretty difficult to get an NcNode instance with any of the NodeCalculator keywords (node, attr, attrs, ...), except for when they are initialized. This method helps for those special cases.

Parameters **attr** (*str*) – Attribute on the Maya node this instance refers to.

Returns

Instance with the given attr in its Attrs, or None if no attr was specified.

Return type *NcNode* or None

auto_state()

Print the status of `_auto_unravel` and `_auto_consolidate`.

get()

Get the value of a `NcNode/NcAttrs`-attribute.

Note: Works similar to a `cmds.getAttr()`.

Returns Value of the queried attribute.

Return type int or float or list

get_shapes (*full=False*)

Get shape nodes of `self.node`.

Parameters **full** (*bool*) – Return full or shortest dag path

Returns List of `MObjects` of shapes.

Return type list

nodes

Property that returns node within list.

Note: This property mostly exists to maintain consistency with `NcList`. Even though nodes of a `NcNode/NcAttrs` instance will always be a list of length 1 it might come in handy to match the property of `NcLists`!

Returns Name of Maya node this instance refers to, in a list.

Return type list

plugins

Property to allow easy access to the Node-plugins.

Note: A “plug” stands for “node.attr”!

Returns List of plugins. Empty list if no attributes are defined!

Return type list

set (*value*)

Set or connect the value of a `NcNode/NcAttrs`-attribute.

Note: Similar to a `cmds.setAttr()`.

Parameters **value** (`NcNode` or `NcAttrs` or *str* or *int* or *float* or *list* or *tuple*) – Connect attribute to this value (=plug) or set attribute to this value/array.

set_auto_consolidate (*state*)

Change the auto consolidating state.

Note: Check docString of `set_global_auto_consolidate` for more info!

Parameters `state` (*bool*) – Desired auto consolidate state: On/Off

set_auto_unravel (*state*)

Change the auto unravelling state.

Note: Check docString of `set_global_auto_unravel` for more info!

Parameters `state` (*bool*) – Desired auto unravel state: On/Off

to_py_node (*ignore_attrs=False*)

Get a PyNode from a NcNode/NcAttrs instance.

Parameters `ignore_attrs` (*bool*) – Don't use attrs when creating PyNode instance. When set to True only the node will be used for PyNode instantiation. Defaults to False.

Returns PyNode-instance of this node or plug

Return type `pm.PyNode`

Raises `RuntimeError` – If the user requested a PyNode of an NcNode/NcAttrs with multiple attrs. PyNodes can only contain one attr max.

class `core.NcList` (*args)

NcList is a list with overloaded operators (inherited from NcBaseClass).

Note: NcList has the following keywords:

- `nodes`: Returns Maya nodes in NcList: [node, ...] (list of strings)

NcList inherits from list, for things like `isinstance(NcList, list)`.

__copy__ ()

Behavior for `copy.copy()`.

Returns Shallow copy of this NcList instance.

Return type *NcList*

__deepcopy__ (*memo=None*)

Behavior for `copy.deepcopy()`.

Parameters `memo` (*dict*) – Memo-dictionary to be passed to `deepcopy`.

Returns Deep copy of this NcList instance.

Return type *NcList*

__delitem__ (*index*)

Delete the item at the given index from this NcList instance.

Parameters `index` (*int*) – Index of the item to be deleted.

__getattr__ (*name*)

Get a list of NcAttrs instances, all with the requested attribute.

Note: There are certain keywords that will NOT return a new NcAttrs:

- `attrs`: Returns currently stored NcAttrs of this NcNode instance.
 - `attrs_list`: Returns stored attrs: `[attr, ...]` (list of strings).
 - `node`: Returns name of Maya node in scene (str).
 - `nodes`: Returns name of Maya node in scene in a list (`[str]`).
 - `plugins`: Returns stored plugs: `[node.attr, ...]` (list of strings).
-

Parameters `name` (*str*) – Name of requested attribute

Returns New NcList with requested NcAttrs.

Return type *NcList*

Example

```
# getattr is invoked by .attribute:
a = Node(["pCube1.ty", "pSphere1.tx"]) # Initialize NcList.
Op.average(a.attrs) # Average .ty on first with .tx on second.
Op.average(a.tz) # Average .tz on both nodes.
```

`__getitem__` (*index*)

Get stored item at given index.

Note: This looks through the `_items` list of this NcList instance.

Parameters `index` (*int*) – Index of desired item

Returns Stored item at index.

Return type *NcNode* or *NcValue*

`__init__` (**args*)

Initialize new NcList-instance.

Parameters `args` (*NcNode* or *NcAttrs* or *NcValue* or *str* or *list* or *tuple*) – Any number of values that should be stored as an array of values.

`__iter__` ()

Iterate over items stored in this NcList instance.

Yields *NcNode* or *NcAttrs* or *NcValue* – Next item in list of attributes.

Raises `StopIteration` – If end of `NcList._items` is reached.

`__len__` ()

Return the length of the NcList.

Returns Number of items stored in this NcList instance.

Return type `int`

`__repr__()`
Unambiguous format of NcList instance.

Note: For example invoked by running highlighted NcList instance in Maya

Returns String of concatenated class-type, node and attrs.

Return type `str`

`__reversed__()`
Reverse the list of stored items on this NcList instance.

Returns New instance with reversed list of items.

Return type `NcList`

`__setattr__(name, value)`
Set or connect list items to the given value.

Note: Attribute setting works similar to NcNode and NcAttrs instances, in order to provide a (hopefully) seamless workflow, whether using NcNodes, NcAttrs or NcLists.

Parameters

- **name** (`str`) – Name of the attribute to be set. “attrs” is keyword!
- **value** (`NcNode` or `NcAttrs` or `str` or `int` or `float` or `list` or `tuple`) – Connect attr to this object or set attr to this value/array

Example

```
setattr is invoked by equal-sign. Does NOT work without attr:
a = Node(["pCube1.ty", "pSphere1.tx"]) # Initialize NcList.
a.attrs = 7 # Set list items to 7; .ty on first, .tx on second.
a.tz = 7 # Set the tz-attr on all items in the NcList to 7.
a = 7 # Does NOT work! It looks like same operation as above,
      # but here Python calls the assignment operation, NOT
      # setattr. The assignment-operation can't be overridden.
```

`__setitem__(index, value)`
Set or connect attribute at index to the given value.

Note: This looks at the `_items` list of this NcList instance

Parameters

- **index** (`int`) – Index of item to be set
- **value** (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Set/connect item at index to this.

__str__()
Readable format of NcList instance.

Note: For example invoked by using `print(NcList instance)` in Maya

Returns String of all NcList `_items`.

Return type `str`

__weakref__
list of weak references to the object (if defined)

static _convert_item_to_nc_instance(*item*)
Convert given item into a NodeCalculator friendly class instance.

Parameters *item* (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Item to be converted into either an `NcNode` or an `NcValue`.

Returns Given item in the appropriate format.

Return type `NcNode` or `NcValue`

Raises `RuntimeError` – If the given item cannot be converted into an `NcNode` or `NcValue`.

append(*value*)
Append value to list of items.

Note: Given value will be converted automatically to appropriate NodeCalculator type before being appended!

Parameters *value* (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Value to append.

attr(*attr=None*)
Get new NcList instance with given attribute (keywords are allowed).

Note: Basically a new NcList with `.attr()` run on all its items.

Parameters *attr* (`str`) – Attribute on the Maya nodes.

Returns

Instance containing `NcAttrs` or an empty NcList when no `attr` was specified.

Return type `NcList`

extend(*other*)
Extend NcList with another list.

Parameters *other* (`NcList` or `list`) – List to be added to the end of this NcList.

get()
Get current value of all items within this NcList instance.

Note: NcNode & NcAttrs instances in list are queried. NcValues are added to return list unaltered.

Returns

List of queried values. Can be list of (int, float, list), depending on “queried” attributes!

Return type list

insert (*index*, *value*)

Insert value to list of items at the given index.

Note: Given value will be converted automatically to appropriate NodeCalculator type before being inserted!

Parameters

- **index** (*int*) – Index at which the value should be inserted.
- **value** (*NcNode or NcAttrs or str or int or float*) – Value to insert.

node

Property to warn user about inappropriate access.

Note: Only NcNode & NcAttrs allow to access their node via node-property. Since user might not be aware of creating NcList instance: Give a hint that NcList instances have a nodes-property instead.

nodes

Sparse list of all nodes within NcList instance.

Note: Only names of Maya nodes are in return_list. Furthermore: It is a sparse list without any duplicate names.

This can be useful for example for `cmds.hide(my_collection.nodes)`

Returns List of names of Maya nodes stored in this NcList instance.

Return type list

set (*value*)

Set or connect the value of all NcNode/NcAttrs-attributes in NcList.

Note: Similar to a `cmds.setAttr()` on a list of plugs.

Parameters value (*NcNode or NcAttrs or str or int or float or list or tuple*) – Connect attribute to this value (=plug) or set attribute to this value/array.

class `core.NcNode` (*node*, *attrs=None*, *auto_unravel=None*, *auto_consolidate=None*)

NcNodes are linked to Maya nodes & can hold attrs in a NcAttrs-instance.

Note: Getting attr X from an NcNode that holds attr Y only returns: NcNode.X In contrast; NcAttrs instances “concatenate” attrs: Getting attr X from an NcAttrs that holds attr Y returns: NcAttrs.Y.X

__getattr__ (*name*)

Get a new NcAttrs instance with the requested attribute.

Note: There are certain keywords that will NOT return a new NcAttrs:

- **attrs:** Returns currently stored NcAttrs of this NcNode instance.
 - **attrs_list:** Returns stored attrs: [attr, ...] (list of strings).
 - **node:** Returns name of Maya node in scene (str).
 - **nodes:** Returns name of Maya node in scene in a list ([str]).
 - **plugins:** Returns stored plugs: [node.attr, ...] (list of strings).
-

Parameters **name** (*str*) – Name of requested attribute

Returns New OR stored instance, if keyword “attrs” was used!

Return type *NcAttrs*

Example

```
a = Node("pCube1") # Create new Node-object
a.tx # invokes __getattr__ and returns a new Node-object.
      It's the same as typing Node("a.tx")
```

__getitem__ (*index*)

Get stored attribute at given index.

Note: Looks through list of attrs stored in the NcAttrs of this NcNode.

Parameters **index** (*int*) – Index of desired item

Returns New NcNode instance, only with attr at index.

Return type *NcNode*

__init__ (*node, attrs=None, auto_unravel=None, auto_consolidate=None*)

Initialize NcNode-class instance.

Note: **__setattr__** is altered. The usual “self.node=node” results in loop! Therefore attributes need to be set a bit awkwardly via **__dict__**!

NcNode uses an MObject as its reference to the Maya node it belongs to. Maya node MUST therefore exist at instantiation time!

Parameters

- **node** (*str or NcNode or NcAttrs or MObject*) – Represents a Maya node

- **attrs** (*str or list or NcAttrs*) – Represents Maya attrs on the node
- **auto_unravel** (*bool*) – Should attrs be auto-unravelling? Check `set_global_auto_unravel` docString for more details.
- **auto_consolidate** (*bool*) – Should attrs be auto-consolidated? Check `set_global_auto_consolidate` docString for more details.

`_node_mobj`

Reference to Maya node.

Type `MObject`

`_held_attrs`

`NcAttrs` instance that defines the attrs.

Type `NcAttrs`

Raises

- `RuntimeError` – If number was given to initialize an `NcNode` with.
- `RuntimeError` – If list/tuple was given to initialize an `NcNode` with.
- `RuntimeError` – If the given string doesn't represent a unique, existing Maya node in the scene.

Example

```
a = Node("pCube1") # Node invokes NcNode instantiation!
b = Node("pCube2.ty")
b = Node("pCube3", ["ty", "tz", "tx"])
```

`attrs`

Get currently stored `NcAttrs` instance of this `NcNode`.

Returns `NcAttrs` instance that represents Maya attributes.

Return type `NcAttrs`

`attrs_list`

Get list of stored attributes of this `NcNode` instance.

Returns List of strings that represent Maya attributes.

Return type `list`

`node`

Get the name of Maya node this `NcNode` refers to.

Returns Name of Maya node in the scene.

Return type `str`

class `core.Node(*args, **kwargs)`

Return instance of appropriate type, based on given args

Note: `Node` is an abstract class that returns components of appropriate type that can then be involved in a `NodeCalculator` calculation.

Parameters

- **item** (*bool or int or float or str or list or tuple*) – Maya node, value, list of nodes, etc.
- **attrs** (*str or list or tuple*) – String or list of strings that are an attribute on this node. Defaults to None.
- **auto_unravel** (*bool*) – Should attrs automatically be unravelled into child attrs when operations are performed on this Node? Defaults to None, which means GLOBAL_AUTO_UNRAVEL is used. NodeCalculator works best if this is left unchanged!
- **auto_consolidate** (*bool*) – Should attrs automatically be consolidated into parent attrs when operations are performed on this Node, to reduce the amount of connections? Defaults to None, which means GLOBAL_AUTO_UNRAVEL is used. Sometimes parent plugs don't update/evaluate reliably. If that's the case; use this flag or `noca.set_global_auto_consolidate(False)`.

Returns Instance with given args.

Return type *NcNode* or *NcList* or *NcValue*

Example

```
# NcNode instance with pCube1 as node and tx as attr
Node("pCube.tx")
# NcNode instance with pCube1 as node and tx as attr
Node("pCube", "tx")
# NcNode instance with pCube1 as node and tx as attr
Node("pCube", ["tx"])

# NcList instance with value 1 and NcNode with pCube1
Node([1, "pCube"])

# NcIntValue instance with value 1
Node(1)
```

__init__ (**args, **kwargs*)
Pass this init.

The Node-class only serves to redirect to the appropriate type based on the given args! Therefore the init must not do anything.

Parameters

- **args** (*list*) – This dummy-init accepts any arguments.
- **kwargs** (*dict*) – This dummy-init accepts any keyword arguments.

__weakref__
list of weak references to the object (if defined)

class `core.OperatorMetaClass` (*name, bases, body*)
MetaClass for NodeCalculator operators that go beyond basic math.

Note: A meta-class was used to ensure the “Op”-class to be a singleton class. Some methods are created on the fly in the `__init__` method.

__init__ (*name, bases, body*)
OperatorMetaClass-class constructor

Note: name, bases, body are necessary for `__metaclass__` to work properly

__weakref__

list of weak references to the object (if defined)

available (*full=False*)

Print all available operators.

Parameters **full** (*bool*) – If False only the operator-names are printed. If True the docString of all operators is printed. Defaults to False.

class `core.Tracer` (*trace=True, print_trace=False, pprint_trace=False, cheers_love=False*)

Class that returns all Maya commands executed by NodeCalculator formula.

Note: Any NodeCalculator formula enclosed in a with-statement will be logged.

Example

```
with Tracer(pprint_trace=True) as s:
    a.tx = b.ty - 2 * c.tz
print(s)
```

__enter__ ()

Set up NcBaseClass class-variables for tracing.

Note: The returned variable is what X in “with noca.Tracer() as X” will be.

Returns List of all executed commands.

Return type list

__exit__ (*exc_type, value, traceback*)

Print executed commands at the end of the with-statement.

__init__ (*trace=True, print_trace=False, pprint_trace=False, cheers_love=False*)

Tracer-class constructor.

Parameters

- **trace** (*bool*) – Enables/disables tracing.
- **print_trace** (*bool*) – Print command stack as a list.
- **pprint_trace** (*bool*) – Print command stack as a multi-line string.
- **cheers_love** (*bool*) – ;)

__weakref__

list of weak references to the object (if defined)

`core.__load_extension` (*noca_extension*)

Load the given extension in the correct way for the NodeCalculator.

Note: Check the tutorials and example extension files to see how you can create your own extensions.

Parameters `noca_extension` (*module*) – Extension Python module to be loaded.

`core.__load_extensions()`

Import the potential NodeCalculator extensions.

`core.__add_to_node_bin(node)`

Add a node to NODE_BIN to keep track of created nodes for easy cleanup.

Note: Nodes are stored in NODE_BIN by name, NOT MPlug! Therefore, if a node was renamed it will not be deleted by cleanup().

Parameters `node` (*str*) – Name of Maya node to be added to the NODE_BIN.

`core.__check_for_parent_attribute(plug_list)`

Reduce the given list of plugs to a single parent attribute.

Parameters `plug_list` (*list*) – List of plugs: ["node.attribute", ...]

Returns

If parent attribute was found it is returned as an MPlug instance, otherwise None is returned

Return type MPlug or None

`core.__consolidate_plugs_to_min_dimension(*plugs)`

Try to consolidate the given input plugs.

Note: A full set of child attributes can be reduced to their parent attr: ["tx", "ty", "tz"] becomes ["t"]

A 3D to 3D connection can be 1 connection if both plugs have a parent attr! However, a 1D attr can not connect to a 3D attr and must NOT be consolidated!

Parameters `plugs` (*list(NcNode or NcAttrs or str or int or float or list or tuple)*) – Plugs to check.

Returns

Consolidated plugs, if consolidation was successful. Otherwise given inputs are returned unaltered.

Return type list

`core.__create_node_name(operation, *args)`

Create a procedural Maya node name that is as descriptive as possible.

Parameters

- **operation** (*str*) – Operation the new node has to perform
- **args** (*MPlug or NcNode or NcAttrs or list or numbers or str*) – Attributes connecting into the newly created node.

Returns Generated name for the given node operation and args.

Return type str

`core._create_operation_node(operation, *args)`

Create & connect adequately named Maya nodes for the given operation.

Parameters

- **operation** (*str*) – Operation the new node has to perform
- **args** (*NcNode or NcAttrs or str*) – Attrs connecting into created node

Returns

Either new **NcNode** instance with the newly created Maya-node of type **OPERATORS[operation][“node”]** and with attributes stored in **OPERATORS[operation][“outputs”]**. If the outputs are multidimensional (for example “translateXYZ” & “rotateXYZ”) a new **NcList** instance is returned with **NcNodes** for each of the outputs.

Return type *NcNode* or *NcList*

`core._create_traced_operation_node(operation, attrs)`

Create named Maya node for the given operation & add cmds to `_command_stack` if Tracer is active.

Parameters

- **operation** (*str*) – Operation the new node has to perform
- **attrs** (*MPlug or NcNode or NcAttrs or list or numbers or str*) – Attrs that will be connecting into the newly created node.

Returns Name of newly created Maya node.

Return type str

`core._format_docstring(*args, **kwargs)`

Format docString of a function: Substitute placeholders with (kw)args.

Note: Formatting your docString directly won’t work! It won’t be a string literal anymore and Python won’t consider it a docString! Replacing the docString (`__doc__`) via this closure circumvents this issue.

Parameters

- **args** (*list*) – Arguments for the string formatting: `.format()`
- **kwargs** (*list*) – Keyword arguments for the string formatting: `.format()`

Returns The function with formatted docString.

Return type executable

`core._get_node_inputs(operation, new_node, args_list)`

Get node-inputs based on operation-type and involved arguments.

Note: To anyone delving deep enough into the NodeCalculator to reach this point; I apologize. This function in particular is difficult to grasp. The main idea is to find which node-inputs (defined in the **OPERATORS**-dictionary) are needed for the given args. Dealing with array-inputs and often 3D-inputs is the difficult part. I hope the following explanation will make it easier to understand what is happening.

Within this function we deal a lot with different levels of arguments:

args_list (*list*)

> **arg_element** (list)

> **arg_item** (list or MPlug/value/...) > **arg_axis** (MPlug/value/...)

The **arg_item**-level might seem redundant. The reason for its existence are array-input attributes (`input[0]`, etc.). They need to be a list of items under one **arg_element**. That way one can loop over all array-input **arg_items** in an **arg_element** and get the correct indices, even if there is a mix of non-array input attrs and array-input attrs. Without this extra layer an input before the array-input would throw off the indices by 1!

The **ARGS_LIST** is made up of the various arguments that will connect into the node.

> [array-values, translation-values, rotation-values, ...]

The **ARG_ELEMENT** is what will set/connect into an attribute “section” of a node. For array-inputs **THIS** is what matters(!), because one attribute section (`input[{array}]`) will actually be made up of many inputs.

> [array-values]

The **ARG_ITEM** is one particular **arg_element**. For **arg_elements** that are array-input the **arg_item** is a specific input of a array-input. For non-array-inputs the **arg_elements** & the **arg_item** are equivalent!

> [array-input-value[0]]

The **ARG_AXIS** is the most granular item, referring to a particular Maya node attribute.

> [array-value[0].valueX]

Parameters

- **operation** (*str*) – Operation the new node has to perform.
- **new_node** (*str*) – Name of newly created Maya node.
- **args_list** (*NcNode* or *NcAttrs* or *NcValue*) – Attrs/Values the node attrs will be connected/set to.

Raises `RuntimeError` – If trying to connect a multi-dimensional attr into a 1D attr. This is an ambiguous connection that can’t be resolved.

Returns

(**clean_inputs_list**, **clean_args_list**, **max_arg_element_len**, **max_arg_axis_len**) >
clean_inputs_list holds all necessary node inputs for given args. > clean_args_list holds args that were adjusted to match clean_inputs_list. > max_arg_element_len holds the highest dimension of array attrs. > max_arg_axis_len holds highest number of attribute axis involved.

Return type tuple

Example

```
These are examples of how the different "levels" of the args_list look like, described in the Note-section. Notice how the args_list is made up of arg_elements, which are made up of arg_items, which in turn are composed of arg_axis.
```

```
args_list = [  
    [  
        [<OpenMaya.MPlug X>, <OpenMaya.MPlug Y>, <OpenMaya.MPlug Z>],
```

(continues on next page)

(continued from previous page)

```

        <OpenMaya.MPlug A>,
        2
    ]
]

# Note: This example would be for an array-input attribute of a node!
arg_elements = [
    [<OpenMaya.MPlug X>, <OpenMaya.MPlug Y>, <OpenMaya.MPlug Z>],
    <OpenMaya.MPlug A>,
    2
]

arg_item = [<OpenMaya.MPlug X>, <OpenMaya.MPlug Y>, <OpenMaya.MPlug Z>]

arg_axis = <OpenMaya.MPlug X>

```

`core._get_node_outputs` (*operation*, *new_node*, *max_array_len*, *max_axis_len*)

Get node-outputs based on operation-type and involved arguments.

Note: See docString of `_get_node_inputs` for origin of `max_array_len` and `max_axis_len`, as well as what `output_element` or `output_axis` means.

Parameters

- **operation** (*str*) – Operation the new node has to perform.
- **new_node** (*str*) – Name of newly created Maya node.
- **max_array_len** (*int* or *None*) – Highest dimension of arrays.
- **max_axis_len** (*int*) – Highest dimension of attribute axis.

Returns

List of `NcNode` instances that hold an attribute according to the outputs defined in the OPERATORS dictionary.

Return type list

`core._is_consolidation_allowed` (*inputs*)

Check for any `NcBaseNode`-instance that is NOT set to auto consolidate.

Parameters **inputs** (`NcNode` or `NcAttrs` or *str* or *int* or *float* or *list* or *tuple*) – Items to check for a turned off auto-consolidation.

Returns True, if all given items allow for consolidation.

Return type bool

`core._is_valid_maya_attr` (*plug*)

Check if given plug is of an existing Maya attribute.

Parameters **plug** (*str*) – String of a Maya plug in the scene (`node.attr`).

Returns Whether the given plug is an existing plug in the scene.

Return type bool

`core._join_cmds_kwargs (**kwargs)`

Concatenates Maya command kwargs for Tracer.

Parameters `kwargs` (*dict*) – Key/value-pairs that should be converted to a string.

Returns String of kwargs&values for the command in the Tracer-stack.

Return type `str`

`core._set_or_connect_a_to_b (obj_a_list, obj_b_list, **kwargs)`

Set or connect the first list of inputs to the second list of inputs.

Parameters

- **obj_a_list** (*list*) – List of MPlugs to be set or connected into.
- **obj_b_list** (*list*) – List of MPlugs, int, float, etc. which obj_a_list items will be set or connected to.
- **kwargs** (*dict*) – Arguments used in `_traced_set_attr` (~ `cmds.setAttr`)

Returns Returns False, if setting/connecting was not possible.

Return type `bool`

Raises

- `RuntimeError` – If an item of the obj_a_list isn't a Maya attribute.
- `RuntimeError` – If an item of the obj_b_list can't be set/connected due to unsupported type.

`core._split_plug_into_node_and_attr (plug)`

Split given plug into its node and attribute part.

Parameters `plug` (*MPlug or str*) – Plug of a Maya node/attribute combination.

Returns

Strings of separated node and attribute part or **None** if separation was not possible.

Return type tuple or None

Raises `RuntimeError` – If the given plug could not be split into node & attr.

`core._traced_add_attr (node, **kwargs)`

Add attr to Maya node & add cmds to `_command_stack` if Tracer is active.

Note: This is simply an overloaded `cmds.addAttr(node, **kwargs)`.

Parameters

- **node** (*str*) – Maya node the attribute should be added to.
- **kwargs** (*dict*) – `cmds.addAttr`-flags

`core._traced_connect_attr (plug_a, plug_b)`

Connect 2 plugs & add command to `_command_stack` if Tracer is active.

Note: This is `cmds.connectAttr(plug_a, plug_b, force=True)` with Tracer-stuff.

Parameters

- **plug_a** (*MPlug or str*) – Source plug
- **plug_b** (*MPlug or str*) – Destination plug

`core._traced_create_node` (*node_type, **kwargs*)

Create a Maya node and add it to the `_traced_nodes` if Tracer is active.

Note: This is simply an overloaded `cmds.createNode(node_type, **kwargs)`. It includes the `cmds.parent` command if parenting flags are given.

If Tracer is active: Created nodes are associated with a variable. If they are referred to later on in the NodeCalculator statement, the variable name will be used instead of their node-name.

Parameters

- **node_type** (*str*) – Type of the Maya node that should be created.
- **kwargs** (*dict*) – `cmds.createNode` & `cmds.parent` flags

Returns Name of newly created Maya node.

Return type `str`

`core._traced_get_attr` (*plug*)

Get attr of Maya node & add cmds to `_command_stack` if Tracer is active.

Note: This is a tweaked & overloaded `cmds.getAttr(plug)`: Awkward return values of 3D-attribs are converted from `tuple(list())` to a simple `list()`.

Parameters **plug** (*MPlug or str*) – Plug of Maya node, whose value should be queried.

Returns Queried value of Maya node plug.

Return type `list` or `numbers` or `bool` or `str`

`core._traced_set_attr` (*plug, value=None, **kwargs*)

Set attr on Maya node & add cmds to `_command_stack` if Tracer is active.

Note: This is simply an overloaded `cmds.setAttr(plug, value, **kwargs)`.

Parameters

- **plug** (*MPlug or str*) – Plug of a Maya node that should be set.
- **value** (*list or numbers or bool*) – Value the given plug should be set to.
- **kwargs** (*dict*) – `cmds.setAttr`-flags

`core._unravel_and_set_or_connect_a_to_b` (*obj_a, obj_b, **kwargs*)

Set `obj_a` to value of `obj_b` OR connect `obj_b` into `obj_a`.

Note: Allowed assignments are: (1-D stands for 1-dimensional, X-D for multi-dim; 2-D, 3-D, ...)

> Setting 1-D attribute to a 1-D value/attr # `pCube1.tx = 7`

- > Setting X-D attribute to a 1-D value/attr # pCube1.t = 7 # equal to [7]*3
 - > Setting X-D attribute to a X-D value/attr # pCube1.t = [1, 2, 3]
 - > Setting 1-D attribute to a X-D value/attr # Error: Ambiguous connection!
 - > Setting X-D attribute to a Y-D value/attr # Error: Dimension mismatch that can't be resolved!
-

Parameters

- **obj_a** (*NcNode* or *NcAttrs* or *str*) – Needs to be a plug. Either as a NodeCalculator-object or as a string (“node.attr”)
- **obj_b** (*NcNode* or *NcAttrs* or *int* or *float* or *list* or *tuple* or *string*) – Can be a numeric value, a list of values or another plug either in the form of a NodeCalculator-object or as a string (“node.attr”)
- **kwargs** (*dict*) – Arguments used in `_traced_set_attr` (~ `cmds.setAttr`)

Raises

- `RuntimeError` – If trying to connect a multi-dimensional attr into a 1D attr. This is an ambiguous connection that can't be resolved.
- `RuntimeError` – If trying to connect a multi-dimensional attr into a multi-dimensional attr with different dimensionality. This is a dimension mismatch that can't be resolved!

`core._unravel_base_node_instance` (*base_node_instance*)

Unravel NcBaseNode instance.

Get name of Maya node or MPlug of Maya attribute the NcBaseNode refers to.

Parameters `base_node_instance` (*NcNode* or *NcAttrs*) – Instance to find Mplug for.

Returns

MPlug of the Maya attribute the given NcNode/NcAttrs refers to or name of node, if no attrs are defined.

Return type MPlug or str

`core._unravel_item` (*item*)

Turn input into MPlugs or values that can be set/connected by Maya.

Note: The items of a list are all unravelled as well! Parent plug becomes list of child plugs: “t” -> [“tx”, “ty”, “tz”]

Parameters (`MPlug`, `NcList` or `NcNode` or `NcAttrs` or `NcValue` or `list` or `tuple` or (*item*) – str or numbers): input to be unravelled/cleaned.

Returns MPlug or value

Return type MPlug or *NcValue* or int or float or list

Raises `TypeError` – If given item is of an unsupported type.

`core._unravel_item_as_list` (*item*)

Convert input into clean list of values or MPlugs.

Parameters `item` (*NcNode* or *NcAttrs* or *NcList* or *int* or *float* or *list* or *str*) – input to be unravelled and returned as list.

Returns List consistent of values or MPlugs

Return type list

`core._unravel_list(list_instance)`

Unravel list instance; get value or MPlug of its items.

Parameters `list_instance` (*list or tuple*) – list to be unravelled.

Returns List of unravelled items.

Return type list

`core._unravel_nc_list(nc_list)`

Unravel NcList instance; get value or MPlug of its NcList-items.

Parameters `nc_list` (`NcList`) – NcList to be unravelled.

Returns List of unravelled NcList-items.

Return type list

`core._unravel_plug(node, attr)`

Convert Maya node/attribute combination into an MPlug.

Note: Tries to break up a parent attribute into its child attributes: `.t -> [tx, ty, tz]`

Parameters

- **node** (*str*) – Name of the Maya node
- **attr** (*str*) – Name of the attribute on the Maya node

Returns

MPlug of the Maya attribute, list of MPlugs if a parent attribute was unravelled to its child attributes.

Return type MPlug or list

`core._unravel_str(str_instance)`

Convert name of a Maya plug into an MPlug.

Parameters `str_instance` (*str*) – Name of the plug; “node.attr”

Returns

MPlug of the Maya attribute, None if given string doesn’t refer to a valid Maya plug in the scene.

Return type MPlug or None

`core.cleanup(keep_selected=False)`

Remove all nodes created by the NodeCalculator, based on node names.

Note: Nodes are stored in `NODE_BIN` by name, NOT MPlug! Therefore, if a node was renamed it will not be deleted by this function. This is intentional; cleanup is for cases of fast iteration, where a lot of nodes can accumulate fast. It should interfere with anything the user wants to keep as little as possible!

Parameters **keep_selected** (*bool*) – Prevent selected nodes from being deleted. Defaults to False.

`core.create_node (node_type, name=None, **kwargs)`

Create a new node of given type as an NcNode.

Parameters

- **node_type** (*str*) – Type of Maya node to be created
- **name** (*str*) – Name for new Maya-node
- **kwargs** (*dict*) – arguments that are passed to Maya createNode function

Returns Instance that is linked to the newly created transform

Return type *NcNode*

Example

```
a = noca.create_node("transform", "myTransform")
a.t = [1, 2, 3]
```

`core.locator (name=None, **kwargs)`

Create a Maya locator node as an NcNode.

Parameters

- **name** (*str*) – Name of locator instance that will be created
- **kwargs** (*dict*) – keyword arguments given to create_node function

Returns Instance that is linked to the newly created locator

Return type *NcNode*

Example

```
a = noca.locator("myLoc")
a.t = [1, 2, 3]
```

`core.noca_op (func)`

Add given function to the Op-class.

Note: This is a decorator used in NodeCalculator extensions! It makes it easy for the user to add additional operators to the Op-class.

Check the tutorials and example extension files to see how you can create your own extensions.

Parameters **func** (*executable*) – Function to be added to Op as a method.

`core.reset_cleanup ()`

Empty the cleanup queue without deleting the nodes.

`core.set_global_auto_consolidate (state)`

Set the global auto consolidate state.

Note: Auto consolidate combines full set of child attrs to their parent attr: ["translateX", "translateY", "translateZ"] becomes "translate".

Consolidating plugs is preferable: it will make your node graph cleaner and easier to read. However: Using parent plugs can sometimes cause update issues on attrs!

Parameters `state` (*bool*) – State auto consolidate should be set to

`core.set_global_auto_unravel` (*state*)
Set the global auto unravel state.

Note: Auto unravel breaks up a parent attr into its child attrs: “translate” becomes [“translateX”, “translateY”, “translateZ”].

This behaviour is desired in most cases for the NodeCalculator to work. But in some cases the user might want to prevent this. For example: When using the choice-node the user probably wants the inputs to be exactly the ones chosen (not broken up into child-attributes and those connected to the choice node).

Parameters `state` (*bool*) – State auto unravel should be set to

`core.transform` (*name=None, **kwargs*)
Create a Maya transform node as an NcNode.

Parameters

- **name** (*str*) – Name of transform instance that will be created
- **kwargs** (*dict*) – keyword arguments given to `create_node` function

Returns Instance that is linked to the newly created transform

Return type *NcNode*

Example

```
a = noca.transform("myTransform")
a.t = [1, 2, 3]
```

4.3 Operators

These are the available default operators in the NodeCalculator!

Attention: These operators must be accessed via the **Op**-class! For example:

```
import node_calculator.core as noca
node = noca.Node("pCube1")
nocl.Op.decompose_matrix(node.worldMatrix)
```

Basic NodeCalculator operators.

This is an extension that is loaded by default.

The main difference to the `base_functions` is that operators are stand-alone functions that create a Maya node.

`base_operators._extension_operators_init()`
Fill EXTENSION_OPERATORS-dictionary with all available operations.

Note: EXTENSION_OPERATORS holds the data for each available operation: the necessary node-type, its inputs, outputs, etc. This unified data enables to abstract node creation, connection, ..

possible flags: - node: Type of Maya node necessary - inputs: input attributes (list of lists) - outputs: output attributes (list) - operation: set operation-attr for different modes of a node - output_is_predetermined: should always ALL output attrs be added?

Use "{array}" in inputs or outputs to denote an array-attribute!

`base_operators.angle_between(vector_a, vector_b=(1, 0, 0))`
Create angleBetween-node to find the angle between 2 vectors.

Parameters

- **vector_a** (`NcNode` or `NcAttrs` or `int` or `float` or `list`) – Vector to consider for angle between.
- **vector_b** (`NcNode` or `NcAttrs` or `int` or `float` or `list` or `tuple`) – Vector to consider for angle between. Defaults to (1, 0, 0).

Returns Instance with angleBetween-node and output-attribute(s)

Return type *NcNode*

Example

```
matrix = Node("pCube1").worldMatrix
pt = Op.point_matrix_mult(
    [1, 0, 0], matrix, vector_multiply=True
)
Op.angle_between(pt, [1, 0, 0])
```

`base_operators.average(*attrs)`
Create plusMinusAverage-node for averaging input attrs.

Parameters **attrs** (`NcNode` or `NcAttrs` or `NcList` or `string` or `list` or `tuple`) – Inputs to be averaged.

Returns Instance with plusMinusAverage-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.average(Node("pCube.t"), [1, 2, 3])
```

`base_operators.blend(attr_a, attr_b, blend_value=0.5)`
Create blendColor-node.

Parameters

- **attr_a** (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Plug or value to blend from

- **attr_b** (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – Plug or value to blend to
- **blend_value** (*NcNode* or *str* or *int* or *float*) – Plug or value defining blend-amount. Defaults to 0.5.

Returns Instance with blend-node and output-attributes

Return type *NcNode*

Example

```
Op.blend(1, Node("pCube.tx"), Node("pCube.customBlendAttr"))
```

`base_operators.choice` (*inputs*, *selector=0*)

Create choice-node to switch between various input attributes.

Note: Multi index input seems to also require one “selector” per index. So we package a copy of the same selector for each input.

Parameters

- **inputs** (*NcList* or *NcAttrs* or *list*) – Any number of input values or plugs
- **selector** (*NcNode* or *NcAttrs* or *int*) – Selector-attr on choice node to select one of the inputs based on their index. Defaults to 0.

Returns Instance with choice-node and output-attribute(s)

Return type *NcNode*

Example

```
option_a = Node("pCube1.tx")
option_b = Node("pCube2.tx")
switch = Node("pSphere1").add_bool("optionSwitch")
choice_node = Op.choice([option_a, option_b], selector=switch)
Node("pTorus1").tx = choice_node
```

`base_operators.clamp` (*attr_a*, *min_value=0*, *max_value=1*)

Create clamp-node.

Parameters

- **attr_a** (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – Input value
- **min_value** (*NcNode* or *NcAttrs* or *int* or *float* or *list*) – min-value for clamp-operation. Defaults to 0.
- **max_value** (*NcNode* or *NcAttrs* or *int* or *float* or *list*) – max-value for clamp-operation. Defaults to 1.

Returns Instance with clamp-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.clamp(Node("pCube.t"), [1, 2, 3], 5)
```

`base_operators.closest_point_on_mesh(mesh, position=(0, 0, 0), return_all_outputs=False)`

Get the closest point on a mesh, from the given position.

Parameters

- **mesh** (`NcNode` or `NcAttrs` or *str*) – Mesh node.
- **position** (`NcNode` or `NcAttrs` or *int* or *float* or *list*) – Find closest point on mesh to this position. Defaults to (0, 0, 0).
- **return_all_outputs** (*bool*) – Return all outputs as an `NcList`. Defaults to `False`.

Returns

If **return_all_outputs** is set to **True**, an **NcList** is returned with all outputs. Otherwise only the first output (position) is returned as an `NcNode` instance.

Return type *NcNode* or *NcList*

Example

```
cube = Node("pCube1")
Op.closest_point_on_mesh(cube.outMesh, [1, 0, 0])
```

`base_operators.closest_point_on_surface(surface, position=(0, 0, 0), return_all_outputs=False)`

Get the closest point on a surface, from the given position.

Parameters

- **surface** (`NcNode` or `NcAttrs` or *str*) – NURBS surface node.
- **position** (`NcNode` or `NcAttrs` or *int* or *float* or *list*) – Find closest point on surface to this position. Defaults to (0, 0, 0).
- **return_all_outputs** (*bool*) – Return all outputs as an `NcList`. Defaults to `False`.

Returns

If **return_all_outputs** is set to **True**, an **NcList** is returned with all outputs. Otherwise only the first output (position) is returned as an `NcNode` instance.

Return type *NcNode* or *NcList*

Example

```
sphere = Node("nurbsSphere1")
Op.closest_point_on_surface(sphere.local, [1, 0, 0])
```

`base_operators.compose_matrix(translate=None, rotate=None, scale=None, shear=None, rotate_order=None, euler_rotation=None, **kwargs)`

Create composeMatrix-node to assemble matrix from transforms.

Parameters

- **translate** (*NcNode or NcAttrs or str or int or float*) – translate [t] Defaults to None, which corresponds to value 0.
- **rotate** (*NcNode or NcAttrs or str or int or float*) – rotate [r] Defaults to None, which corresponds to value 0.
- **scale** (*NcNode or NcAttrs or str or int or float*) – scale [s] Defaults to None, which corresponds to value 1.
- **shear** (*NcNode or NcAttrs or str or int or float*) – shear [sh] Defaults to None, which corresponds to value 0.
- **rotate_order** (*NcNode or NcAttrs or str or int*) – rot-order [ro] Defaults to None, which corresponds to value 0.
- **euler_rotation** (*NcNode or NcAttrs or bool*) – Euler or quaternion [uer] Defaults to None, which corresponds to True.
- **kwargs** (*dict*) – Short flags, see in [brackets] for each arg above. Long names take precedence!

Returns Instance with composeMatrix-node and output-attribute(s)

Return type *NcNode*

Example

```
in_a = Node("pCube1")
in_b = Node("pCube2")
decomp_a = Op.decompose_matrix(in_a.worldMatrix)
decomp_b = Op.decompose_matrix(in_b.worldMatrix)
Op.compose_matrix(r=decomp_a.outputRotate, s=decomp_b.outputScale)
```

`base_operators.condition(condition_node, if_part=False, else_part=True)`
Set up condition-node.

Note: `condition_node` can be a *NcNode*-instance of a Maya condition node. An appropriate *NcNode*-object gets automatically created when NodeCalculator objects are used in comparisons (`==`, `>`, `>=`, `<`, `<=`). Simply use comparison operators in the first argument. See example.

Parameters

- **condition_node** (*NcNode or bool or int or float*) – Condition-statement. See note and example.
- **if_part** (*NcNode or NcAttrs or str or int or float*) – Value/plug that is returned if the condition evaluates to true. Defaults to False.
- **else_part** (*NcNode or NcAttrs or str or int or float*) – Value/plug that is returned if the condition evaluates to false. Defaults to True.

Returns Instance with condition-node and outColor-attributes

Return type *NcNode*

Example

```
condition_node = Node("pCube1.tx") >= 2
pass_on_if_true = Node("pCube2.ty") + 2
pass_on_if_false = 5 - Node("pCube2.tz").get()
# Op.condition(condition-part, "if true"-part, "if false"-part)
Op.condition(condition_node, pass_on_if_true, pass_on_if_false)
```

`base_operators.cross(attr_a, attr_b=0, normalize=False)`
Create vectorProduct-node for vector cross-multiplication.

Parameters

- **attr_a** (`NcNode` or `NcAttrs` or `str` or `int` or `float` or `list`) – Vector A.
- **attr_b** (`NcNode` or `NcAttrs` or `str` or `int` or `float` or `list`) – Vector B. Defaults to 0.
- **normalize** (`NcNode` or `NcAttrs` or `bool`) – Whether resulting vector should be normalized. Defaults to False.

Returns Instance with vectorProduct-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.cross(Node("pCube.t"), [1, 2, 3], True)
```

`base_operators.curve_info(curve)`
Measure the length of a curve.

Parameters **curve** (`NcNode`, `NcAttrs` or `string`) – The curve to be measured.

Returns Instance with vectorProduct-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.curve_info(Node("nurbsCurve.local"))
```

`base_operators.decompose_matrix(in_matrix, return_all_outputs=False)`
Create decomposeMatrix-node to disassemble matrix into transforms.

Parameters

- **in_matrix** (`NcNode` or `NcAttrs` or `string`) – matrix attr to decompose
- **return_all_outputs** (`bool`) – Return all outputs, as an `NcList`. Defaults to False.

Returns

If **return_all_outputs** is set to **True**, an `NcList` is returned with all outputs. Otherwise only the first output (translate) is returned as an `NcNode` instance.

Return type `NcNode` or `NcList`

Example

```
driver = Node("pCube1")
driven = Node("pSphere1")
decomp = Op.decompose_matrix(driver.worldMatrix)
driven.t = decomp.outputTranslate
driven.r = decomp.outputRotate
driven.s = decomp.outputScale
```

`base_operators.dot(attr_a, attr_b=0, normalize=False)`

Create vectorProduct-node for vector dot-multiplication.

Parameters

- **attr_a** (`NcNode` or `NcAttrs` or `str` or `int` or `float` or `list`) – Vector A.
- **attr_b** (`NcNode` or `NcAttrs` or `str` or `int` or `float` or `list`) – Vector B. Defaults to 0.
- **normalize** (`NcNode` or `NcAttrs` or `bool`) – Whether resulting vector should be normalized. Defaults to False.

Returns Instance with vectorProduct-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.dot(Node("pCube.t"), [1, 2, 3], True)
```

`base_operators.euler_to_quat(angle, rotate_order=0)`

Create eulerToQuat-node to add two quaternions together.

Parameters

- **angle** (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Euler angles to convert into a quaternion.
- **rotate_order** (`NcNode` or `NcAttrs` or `int`) – Order of rotation. Defaults to 0, which represents rotate order “xyz”.

Returns Instance with eulerToQuat-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.euler_to_quat(Node("pCube").rotate, 2)
```

`base_operators.exp(attr_a)`

Raise attr_a to the base of natural logarithms.

Parameters **attr_a** (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Value or attr

Returns Instance with multiplyDivide-node and output-attr(s)

Return type `NcNode`

Example

```
Op.exp(Node("pCube.t"))
```

`base_operators.four_by_four_matrix`(*vector_a=None, vector_b=None, vector_c=None, translate=None*)

Create a four by four matrix out of its components.

Parameters

- **vector_a** (*NcNode or NcAttrs or str or list or tuple or int or float*) – First vector of the matrix; the “x-axis”. Or can contain all 16 elements that make up the 4x4 matrix. Defaults to None, which means the identity matrix will be used.
- **vector_b** (*NcNode or NcAttrs or str or list or tuple or int or float*) – Second vector of the matrix; the “y-axis”. Defaults to None, which means the vector (0, 1, 0) will be used, if matrix is not defined solely by vector_a.
- **vector_c** (*NcNode or NcAttrs or str or list or tuple or int or float*) – Third vector of the matrix; the “z-axis”. Defaults to None, which means the vector (0, 0, 1) will be used, if matrix is not defined solely by vector_a.
- **translate** (*NcNode or NcAttrs or str or list or tuple or int or float*) – Translate-elements of the matrix. Defaults to None, which means the vector (0, 0, 0) will be used, if matrix is not defined solely by vector_a.

Returns Instance with fourByFourMatrix-node and output-attr(s)

Return type *NcNode*

Example

```
cube = Node("pCube1")
vec_a = Op.point_matrix_mult(
    [1, 0, 0],
    cube.worldMatrix,
    vector_multiply=True
)
vec_b = Op.point_matrix_mult(
    [0, 1, 0],
    cube.worldMatrix,
    vector_multiply=True
)
vec_c = Op.point_matrix_mult(
    [0, 0, 1],
    cube.worldMatrix,
    vector_multiply=True
)
out = Op.four_by_four_matrix(
    vector_a=vec_a,
    vector_b=vec_b,
    vector_c=vec_c,
    translate=[cube.tx, cube.ty, cube.tz]
)
```

`base_operators.hold_matrix`(*matrix*)

Create holdMatrix-node for storing a matrix.

Parameters **matrix** (*NcNode or NcAttrs or string or list*) – Matrix to store.

Returns Instance with holdMatrix-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.hold_matrix(Node("pCube1.worldMatrix"))
```

`base_operators.inverse_matrix(in_matrix)`

Create inverseMatrix-node to invert the given matrix.

Parameters `in_matrix` (*NcNode* or *NcAttrs* or *str*) – Matrix to invert

Returns Instance with inverseMatrix-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.inverse_matrix(Node("pCube.worldMatrix"))
```

`base_operators.length(attr_a, attr_b=0)`

Create distanceBetween-node to measure length between given points.

Parameters

- **attr_a** (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – Start point.
- **attr_b** (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – End point. Defaults to 0.

Returns Instance with distanceBetween-node and distance-attribute

Return type *NcNode*

Example

```
Op.len(Node("pCube.t"), [1, 2, 3])
```

`base_operators.matrix_distance(matrix_a, matrix_b=None)`

Create distanceBetween-node to measure distance between matrices.

Parameters

- **matrix_a** (*NcNode* or *NcAttrs* or *str*) – Matrix defining start point.
- **matrix_b** (*NcNode* or *NcAttrs* or *str*) – Matrix defining end point. Defaults to None, which gives the length between the origin and the point described by matrix_a.

Returns Instance with distanceBetween-node and distance-attribute

Return type *NcNode*

Example

```
Op.len(Node("pCube.worldMatrix"), Node("pCube2.worldMatrix"))
```

`base_operators.mult_matrix(*attrs)`

Create multMatrix-node for multiplying matrices.

Parameters `attrs` (`NcNode` or `NcAttrs` or `NcList` or *string* or *list* or *tuple*) – Matrices to multiply together.

Returns Instance with multMatrix-node and output-attribute(s)

Return type *NcNode*

Example

```
matrix_mult = Op.mult_matrix(
    Node("pCube1.worldMatrix"), Node("pCube2").worldMatrix
)
decomp = Op.decompose_matrix(matrix_mult)
out = Node("pSphere")
out.translate = decomp.outputTranslate
out.rotate = decomp.outputRotate
out.scale = decomp.outputScale
```

`base_operators.nearest_point_on_curve(curve, position=(0, 0, 0), return_all_outputs=False)`

Get curve data from a particular point on a curve.

Parameters

- **curve** (`NcNode` or `NcAttrs` or *str*) – Curve node.
- **position** (`NcNode` or `NcAttrs` or *int* or *float* or *list*) – Find closest point on curve to this position. Defaults to (0, 0, 0).
- **return_all_outputs** (*bool*) – Return all outputs as an `NcList`. Defaults to False.

Returns

If `return_all_outputs` is set to **True**, an `NcList` is returned with all outputs. Otherwise only the first output (position) is returned as an `NcNode` instance.

Return type *NcNode* or *NcList*

Example

```
curve = Node("curve1")
Op.nearest_point_on_curve(curve.local, [1, 0, 0])
```

`base_operators.normalize_vector(in_vector, normalize=True)`

Create vectorProduct-node to normalize the given vector.

Parameters

- **in_vector** (`NcNode` or `NcAttrs` or *str* or *int* or *float* or *list*) – Vect.
- **normalize** (`NcNode` or `NcAttrs` or *bool*) – Turn normalize on/off. Defaults to True.

Returns Instance with vectorProduct-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.normalize_vector(Node("pCube.t"))
```

```
base_operators.pair_blend(translate_a=0, rotate_a=0, translate_b=0, rotate_b=0, weight=1,
                           quat_interpolation=False, return_all_outputs=False)
```

Create pairBlend-node to blend between two transforms.

Parameters

- **translate_a** (*NcNode* or *NcAttrs* or *str* or *int* or *float* or *list*) – Translate value of first transform. Defaults to 0.
- **rotate_a** (*NcNode* or *NcAttrs* or *str* or *int* or *float* or *list*) – Rotate value of first transform. Defaults to 0.
- **translate_b** (*NcNode* or *NcAttrs* or *str* or *int* or *float* or *list*) – Translate value of second transform. Defaults to 0.
- **rotate_b** (*NcNode* or *NcAttrs* or *str* or *int* or *float* or *list*) – Rotate value of second transform. Defaults to 0.
- **weight** (*NcNode* or *NcAttrs* or *str* or *int* or *float* or *list*) – Bias towards first or second transform. Defaults to 1.
- **quat_interpolation** (*NcNode* or *NcAttrs* or *bool*) – Use euler (False) or quaternions (True) to interpolate rotation Defaults to False.
- **return_all_outputs** (*bool*) – Return all outputs, as an NcList. Defaults to False.

Returns

If **return_all_outputs** is set to **True**, an **NcList** is returned with all outputs. Otherwise only the first output (translate) is returned as an *NcNode* instance.

Return type *NcNode* or *NcList*

Example

```
a = Node("pCube1")
b = Node("pSphere1")
blend_attr = a.add_float("blend")
Op.pair_blend(a.t, a.r, b.t, b.r, blend_attr)
```

```
base_operators.pass_matrix(matrix, scale=1)
```

Create passMatrix-node for passing and optionally scaling a matrix.

Parameters

- **matrix** (*NcNode* or *NcAttrs* or *string* or *list*) – Matrix to store.
- **scale** (*NcNode* or *NcAttrs* or *int* or *float*) – Scale to be applied to matrix. Defaults to 1.

Returns Instance with passMatrix-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.pass_matrix(Node("pCube1.worldMatrix"))
```

`base_operators.point_matrix_mult` (*in_vector*, *in_matrix*, *vector_multiply=False*)

Create pointMatrixMult-node to transpose the given matrix.

Parameters

- **in_vector** (*NcNode* or *NcAttrs* or *str* or *int* or *float* or *list*) – Vect.
- **in_matrix** (*NcNode* or *NcAttrs* or *str*) – Matrix
- **vector_multiply** (*NcNode* or *NcAttrs* or *str* or *int* or *bool*) – Whether vector multiplication should be performed. Defaults to False.

Returns Instance with pointMatrixMult-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.point_matrix_mult(  
    Node("pSphere.t"),  
    Node("pCube.worldMatrix"),  
    vector_multiply=True  
)
```

`base_operators.point_on_curve_info` (*curve*, *parameter=0*, *as_percentage=False*, *return_all_outputs=False*)

Get curve data from a particular point on a curve.

Parameters

- **curve** (*NcNode* or *NcAttrs* or *str*) – Curve node.
- **parameter** (*NcNode* or *NcAttrs* or *int* or *float* or *list*) – Get curve data at the position on the curve specified by this parameter. Defaults to 0.
- **as_percentage** (*NcNode* or *NcAttrs* or *int* or *float* or *bool*) – Use 0-1 values for parameter. Defaults to False.
- **return_all_outputs** (*bool*) – Return all outputs as an NcList. Defaults to False.

Returns

If **return_all_outputs** is set to **True**, an **NcList** is returned with all outputs. Otherwise only the first output (position) is returned as an *NcNode* instance.

Return type *NcNode* or *NcList*

Example

```
curve = Node("curve1")  
Op.point_on_curve_info(curve.local, 0.5)
```

`base_operators.point_on_surface_info` (*surface*, *parameter=(0, 0)*, *as_percentage=False*, *return_all_outputs=False*)

Get surface data from a particular point on a NURBS surface.

Parameters

- **surface** (*NcNode or NcAttrs or str*) – NURBS surface node.
- **parameter** (*NcNode or NcAttrs or int or float or list*) – UV values that define point on NURBS surface. Defaults to (0, 0).
- **as_percentage** (*NcNode or NcAttrs or int or float or bool*) – Use 0-1 values for parameters. Defaults to False.
- **return_all_outputs** (*bool*) – Return all outputs as an NcList. Defaults to False.

Returns

If **return_all_outputs** is set to **True**, an **NcList** is returned with all outputs. Otherwise only the first output (position) is returned as an **NcNode** instance.

Return type *NcNode or NcList*

Example

```
sphere = Node("nurbsSphere1")
Op.point_on_surface_info(sphere.local, [0.5, 0.5])
```

`base_operators.pow(attr_a, attr_b=2)`
Raise `attr_a` to the power of `attr_b`.

Parameters

- **attr_a** (*NcNode or NcAttrs or str or int or float*) – Value or attr.
- **attr_b** (*NcNode or NcAttrs or str or int or float*) – Value or attr. Defaults to 2.

Returns Instance with multiplyDivide-node and output-attr(s)

Return type *NcNode*

Example

```
Op.pow(Node("pCube.t"), 2.5)
```

`base_operators.quat_add(quat_a, quat_b=(0, 0, 0, 1))`
Create quatAdd-node to add two quaternions together.

Parameters

- **quat_a** (*NcNode or NcAttrs or str or list or tuple*) – First quaternion.
- **quat_b** (*NcNode or NcAttrs or str or list or tuple*) – Second quaternion. Defaults to (0, 0, 0, 1).

Returns Instance with quatAdd-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.quat_add(  
    create_node("decomposeMatrix").outputQuat,  
    create_node("decomposeMatrix").outputQuat,  
)
```

`base_operators.quat_conjugate(quat_a)`

Create quatConjugate-node to conjugate a quaternion.

Parameters `quat_a` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Quaternion to conjugate.

Returns Instance with quatConjugate-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.quat_conjugate(create_node("decomposeMatrix").outputQuat)
```

`base_operators.quat_invert(quat_a)`

Create quatInvert-node to invert a quaternion.

Parameters `quat_a` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Quaternion to invert.

Returns Instance with quatInvert-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.quat_invert(create_node("decomposeMatrix").outputQuat)
```

`base_operators.quat_mul(quat_a, quat_b=(0, 0, 0, 1))`

Create quatProd-node to multiply two quaternions together.

Parameters

- `quat_a` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – First quaternion.
- `quat_b` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Second quaternion. Defaults to (0, 0, 0, 1).

Returns Instance with quatProd-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.quat_mul(  
    create_node("decomposeMatrix").outputQuat,  
    create_node("decomposeMatrix").outputQuat,  
)
```

`base_operators.quat_negate(quat_a)`

Create quatNegate-node to negate a quaternion.

Parameters `quat_a` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Quaternion to negate.

Returns Instance with quatNegate-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.quat_negate(create_node("decomposeMatrix").outputQuat)
```

`base_operators.quat_normalize(quat_a)`

Create quatNormalize-node to normalize a quaternion.

Parameters `quat_a` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Quaternion to normalize.

Returns Instance with quatNormalize-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.quat_normalize(create_node("decomposeMatrix").outputQuat)
```

`base_operators.quat_sub(quat_a, quat_b=(0, 0, 0, 1))`

Create quatSub-node to subtract two quaternions from each other.

Parameters

- `quat_a` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – First quaternion.
- `quat_b` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Second quaternion that will be subtracted from the first. Defaults to (0, 0, 0, 1).

Returns Instance with quatSub-node and output-attribute(s)

Return type `NcNode`

Example

```
Op.quat_sub(
    create_node("decomposeMatrix").outputQuat,
    create_node("decomposeMatrix").outputQuat,
)
```

`base_operators.quat_to_euler(quat_a, rotate_order=0)`

Create quatToEuler-node to convert a quaternion into an euler angle.

Parameters

- `quat_a` (`NcNode` or `NcAttrs` or `str` or `list` or `tuple`) – Quaternion to convert into Euler angles.

- **rotate_order** (*NcNode or NcAttrs or or int*) – Order of rotation. Defaults to 0, which represents rotate order “xyz”.

Returns Instance with quatToEuler-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.quat_to_euler(create_node("decomposeMatrix").outputQuat, 2)
```

```
base_operators.remap_color(attr_a, output_min=0, output_max=1, input_min=0, input_max=1,  
                           values_red=None, values_green=None, values_blue=None)
```

Create remapColor-node to remap the given input.

Parameters

- **attr_a** (*NcNode or NcAttrs or str or int or float*) – Input color.
- **output_min** (*NcNode or NcAttrs or int or float or list*) – min-Value. Defaults to 0.
- **output_max** (*NcNode or NcAttrs or int or float or list*) – max-Value. Defaults to 1.
- **input_min** (*NcNode or NcAttrs or int or float or list*) – old min-Value. Defaults to 0.
- **input_max** (*NcNode or NcAttrs or int or float or list*) – old max-Value. Defaults to 1.
- **values_red** (*list*) – List of tuples for red-graph in the form; (value_Position, value_FloatValue, value_Interp) The value interpolation element is optional (default: linear) Defaults to None.
- **values_green** (*list*) – List of tuples for green-graph in the form; (value_Position, value_FloatValue, value_Interp) The value interpolation element is optional (default: linear) Defaults to None.
- **values_blue** (*list*) – List of tuples for blue-graph in the form; (value_Position, value_FloatValue, value_Interp) The value interpolation element is optional (default: linear) Defaults to None.

Returns Instance with remapColor-node and output-attribute(s)

Return type *NcNode*

Raises

- **TypeError** – If given values isn’t a list of either lists or tuples.
- **RuntimeError** – If given values isn’t a list of lists/tuples of length 2 or 3.

Example

```
Op.remap_color(  
    Node("blinn1.outColor"),  
    values_red=[(0.1, .2, 0), (0.4, 0.3)]  
)
```

`base_operators.remap_hsv(attr_a, output_min=0, output_max=1, input_min=0, input_max=1, values_hue=None, values_saturation=None, values_value=None)`

Create remapHsv-node to remap the given input.

Parameters

- **attr_a** (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Input color.
- **output_min** (`NcNode` or `NcAttrs` or `int` or `float` or `list`) – min-Value. Defaults to 0.
- **output_max** (`NcNode` or `NcAttrs` or `int` or `float` or `list`) – max-Value. Defaults to 1.
- **input_min** (`NcNode` or `NcAttrs` or `int` or `float` or `list`) – old min-Value. Defaults to 0.
- **input_max** (`NcNode` or `NcAttrs` or `int` or `float` or `list`) – old max-Value. Defaults to 1.
- **values_hue** (`list`) – List of tuples for hue-graph in the form; (value_Position, value_FloatValue, value_Interp) The value interpolation element is optional (default: linear) Defaults to None.
- **values_saturation** (`list`) – List of tuples for saturation-graph in form; (value_Position, value_FloatValue, value_Interp) The value interpolation element is optional (default: linear) Defaults to None.
- **values_value** (`list`) – List of tuples for value-graph in the form; (value_Position, value_FloatValue, value_Interp) The value interpolation element is optional (default: linear) Defaults to None.

Returns Instance with remapHsv-node and output-attribute(s)

Return type `NcNode`

Raises

- `TypeError` – If given values isn't a list of either lists or tuples.
- `RuntimeError` – If given values isn't a list of lists/tuples of length 2 or 3.

Example

```
Op.remap_hsv(
    Node("blinn1.outColor"),
    values_saturation=[(0.1, .2, 0), (0.4, 0.3)]
)
```

`base_operators.remap_value(attr_a, output_min=0, output_max=1, input_min=0, input_max=1, values=None)`

Create remapValue-node to remap the given input.

Parameters

- **attr_a** (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Input value
- **output_min** (`NcNode` or `NcAttrs` or `int` or `float` or `list`) – min-Value. Defaults to 0.
- **output_max** (`NcNode` or `NcAttrs` or `int` or `float` or `list`) – max-Value. Defaults to 1.

- **input_min** (*NcNode or NcAttrs or int or float or list*) – old min-Value. Defaults to 0.
- **input_max** (*NcNode or NcAttrs or int or float or list*) – old max-Value. Defaults to 1.
- **values** (*list*) – List of tuples in the following form; (value_Position, value_FloatValue, value_Interp) The value interpolation element is optional (default: linear) Defaults to None.

Returns Instance with remapValue-node and output-attribute(s)

Return type *NcNode*

Raises

- `TypeError` – If given values isn't a list of either lists or tuples.
- `RuntimeError` – If given values isn't a list of lists/tuples of length 2 or 3.

Example

```
Op.remap_value(  
    Node("pCube.t"),  
    values=[(0.1, .2, 0), (0.4, 0.3)]  
)
```

`base_operators.reverse(attr_a)`

Create reverse-node to get 1 minus the input.

Parameters **attr_a** (*NcNode or NcAttrs or str or int or float*) – Input value

Returns Instance with reverse-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.reverse(Node("pCube.visibility"))
```

`base_operators.rgb_to_hsv(rgb_color)`

Create rgbToHsv-node to get RGB color in HSV representation.

Parameters **rgb_color** (*NcNode or NcAttrs or str or int or float*) – Input RGB color.

Returns Instance with rgbToHsv-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.rgb_to_hsv(Node("blinn1.outColor"))
```

`base_operators.set_range(attr_a, min_value=0, max_value=1, old_min_value=0, old_max_value=1)`

Create setRange-node to remap the given input attr to a new min/max.

Parameters

- **attr_a** (*NcNode or NcAttrs or str or int or float*) – Input value.
- **min_value** (*NcNode or NcAttrs or int or float or list*) – New min. Defaults to 0.
- **max_value** (*NcNode or NcAttrs or int or float or list*) – New max. Defaults to 1.
- **old_min_value** (*NcNode or NcAttrs or int or float or list*) – Old min. Defaults to 0.
- **old_max_value** (*NcNode or NcAttrs or int or float or list*) – Old max. Defaults to 1.

Returns Instance with setRange-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.set_range(Node("pCube.t"), [1, 2, 3], 4, [-1, 0, -2])
```

`base_operators.sqrt(attr_a)`

Get the square root of attr_a.

Parameters **attr_a** (*NcNode or NcAttrs or str or int or float*) – Value or attr

Returns Instance with multiplyDivide-node and output-attr(s)

Return type *NcNode*

Example

```
Op.sqrt(Node("pCube.tx"))
```

`base_operators.sum(*attrs)`

Create plusMinusAverage-node for averaging input attrs.

Parameters **attrs** (*NcNode or NcAttrs or NcList or string or list or tuple*) – Inputs to be added up.

Returns Instance with plusMinusAverage-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.average(Node("pCube.t"), [1, 2, 3])
```

`base_operators.transpose_matrix(in_matrix)`

Create transposeMatrix-node to transpose the given matrix.

Parameters **in_matrix** (*NcNode or NcAttrs or str*) – Plug or value for in_matrix

Returns Instance with transposeMatrix-node and output-attribute(s)

Return type *NcNode*

Example

```
Op.transpose_matrix(Node("pCube.worldMatrix"))
```

`base_operators.weighted_add_matrix(*matrices)`

Add matrices with a weight-bias.

Parameters `matrices` (`NcNode` or `NcAttrs` or *list* or *tuple*) – Any number of matrices. Can be a list of tuples; (matrix, weight) or simply a list of matrices. In that case the weight will be evenly distributed between all given matrices.

Returns Instance with `wtAddMatrix`-node and output-attribute(s)

Return type `NcNode`

Example

```
:: cube_a      =      Node("pCube1.worldMatrix")      cube_b      =      Node("pCube2.worldMatrix")
   Op.weighted_add_matrix(cube_a, cube_b)
```

Basic NodeCalculator functions.

This is an extension that is loaded by default.

The main difference to the `base_operators` is that functions rely on operators! They combine existing operators to create more complex setups.

`base_functions.acos(attr_a)`

Arccosine of `attr_a`.

Note: Only works for 1D inputs!

Parameters `attr_a` (`NcNode` or `NcAttrs` or *str* or *int* or *float*) – Value or attr

Returns Instance with node and output-attr.

Return type `NcNode`

Example

```
in_attr = Node("pCube.tx")
Op.acos(in_attr)
```

`base_functions.asin(attr_a)`

Arcsine of `attr_a`.

Note: Only works for 1D inputs!

Parameters `attr_a` (`NcNode` or `NcAttrs` or *str* or *int* or *float*) – Value or attr

Returns Instance with node and output-attr.

Return type `NcNode`

Example

```
in_attr = Node("pCube.tx")
Op.asin(in_attr)
```

`base_functions.atan(attr_a)`
Arctangent of attr_a, which calculates only quadrant 1 and 4.

Note: Only works for 1D inputs!

Parameters `attr_a` (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Value or attr

Returns Instance with node and output-attr.

Return type `NcNode`

Example

```
in_attr = Node("pCube.tx")
Op.atan(in_attr)
```

`base_functions.atan2(attr_a, attr_b)`
Arctangent2 of attr_b/attr_a, which calculates all four quadrants.

Note: The arguments mimic the behaviour of `math.atan2(y, x)`! Make sure you pass them in the right order.

Parameters

- `attr_a` (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Value or attr
- `attr_b` (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Value or attr

Returns Instance with node and output-attr.

Return type `NcNode`

Example

```
x = Node("pCube.tx")
y = Node("pCube.ty")
Op.atan2(y, x)
```

`base_functions.cos(attr_a)`
Cosine of attr_a.

Note: Only works for 1D inputs!

The idea how to set this up with native Maya nodes is from Chad Vernon: <https://www.chadvernon.com/blog/trig-maya/>

Parameters `attr_a` (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Value or attr

Returns Instance with node and output-attr.

Return type *NcNode*

Example

```
in_attr = Node("pCube.tx")
Op.cos(in_attr)
```

`base_functions.sin(attr_a)`
Sine of attr_a.

Note: Only works for 1D inputs!

The idea how to set this up with native Maya nodes is from Chad Vernon: <https://www.chadvernon.com/blog/trig-maya/>

Parameters `attr_a` (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – Value or attr

Returns Instance with node and output-attr.

Return type *NcNode*

Example

```
in_attr = Node("pCube.tx")
Op.sin(in_attr)
```

`base_functions.soft_approach(attr_a, fade_in_range=0.5, target_value=1)`
Follow attr_a, but approach the target_value slowly.

Note: Only works for 1D inputs!

Parameters

- **attr_a** (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – Value or attr
- **fade_in_range** (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – Value or attr. This defines a range over which the target_value will be approached. Before the attr_a is within this range the output of this and the attr_a will be equal. Defaults to 0.5.
- **target_value** (*NcNode* or *NcAttrs* or *str* or *int* or *float*) – Value or attr. This is the value that will be approached slowly. Defaults to 1.

Returns Instance with node and output-attr.

Return type *NcNode*

Example

```
in_attr = Node("pCube.tx")
Op.soft_approach(in_attr, fade_in_range=2, target_value=5)
# Starting at the value 3 (because 5-2=3), the output of this
# will slowly approach the target_value 5.
```

`base_functions.tan(attr_a)`
Tangent of attr_a.

Note: Only works for 1D inputs!

The idea how to set this up with native Maya nodes is from Chad Vernon: <https://www.chadvernon.com/blog/trig-maya/>

Parameters `attr_a` (`NcNode` or `NcAttrs` or `str` or `int` or `float`) – Value or attr

Returns Instance with node and output-attr.

Return type `NcNode`

Example

```
in_attr = Node("pCube.tx")
Op.tan(in_attr)
```

4.4 Extension

NodeCalculator extensions allow you to add your own Operators and therefore tailor it to your specific needs.

```
"""Additional operators for the NodeCalculator; proprietary or custom nodes.

Note:
    If you want to separate out this extension file from the core functionality
    of the NodeCalculator (to maintain your proprietary additions in a separate
    repo or so) you simply have to add the folder where this noca_extension
    module will live to the __init__.py of the node_calculator-module!

    Check noca_extension_maya_math_nodes.py to see an example of how to write a
    NodeCalculator extension.

:author: You ;)
"""

# DON'T import node_calculator.core as noca! It's a cyclical import that fails!
# Most likely the only two things needed from the node_calculator:
from node_calculator.core import noca_op
from node_calculator.core import _create_operation_node

# ~~~~~
# ~~~~~ STEP 1: REQUIRED PLUGINS ~~~~~
# ~~~~~
```

(continues on next page)

(continued from previous page)

```

"""
If your operators require certain Maya plugins to be loaded: Add the name(s) of
those plugin(s) to this list.

You can use this script to find out what plugin a certain node type lives in:

node_type = "" # Enter node type here!
for plugin in cmds.pluginInfo(query=True, listPlugins=True):
    plugin_types = cmds.pluginInfo(plugin, query=True, dependNode=True) or []
    for plugin_type in plugin_types:
        if plugin_type == node_type:
            print "\n>>> {} is part of the plugin {}".format(node_type, plugin)
"""
REQUIRED_EXTENSION_PLUGINS = []

# ~~~~~
# ~~~~~ STEP 2: OPERATORS DICTIONARY ~~~~~
# ~~~~~
"""
EXTENSION_OPERATORS holds the data for each available operation:
the necessary node-type, its inputs, outputs, etc.
This unified data enables to abstract node creation, connection, etc.

Mandatory flags:

- node: Type of Maya node necessary
- inputs: input attributes (list of lists)
- outputs: output attributes (list)

Optional flags:

- operation: many Maya nodes have an "operation" attribute that sets the
operation mode of the node. Use this flag to set this attribute.
- output_is_predetermined: should outputs be truncated to dimensionality of
inputs or should they always stay exactly as specified?

Check here to see lots of examples for the EXTENSION_OPERATORS-dictionary:
_operator_lookup_table_init (in lookup_table.py)
"""
EXTENSION_OPERATORS = {
    # "example_operation": {
    #     "node": "mayaNodeForThisOperation",
    #     "inputs": [
    #         ["singleInputParam"],
    #         ["input1X", "input1Y", "input1Z"],
    #         ["input2X", "input2Y", "input2Z"],
    #         ["input[{array}].inX", "input[{array}].inY", "input[{array}].inZ"],
    #     ],
    #     "outputs": [
    #         ["outputX", "outputY", "outputZ"],
    #     ],
    #     "operation": 3,
    #     "output_is_predetermined": False,
    # }
}

```

(continues on next page)

(continued from previous page)

```
# ~~~~~
# ~~~~~ STEP 3: OPERATOR FUNCTION ~~~~~
# ~~~~~
"""
```

Add a function for every operation that should be accessible via `noca.Op`!

Let's look at this example::

```
@noca_op
def example_operation(attr_a, attr_b=(0, 1, 2), attr_c=False):
    created_node = _create_operation_node(
        'example_operation', attr_a, attr_b, attr_c
    )
    return created_node
```

The decorator `@noca_op` is mandatory! It will take care of adding the function to `noca.Op`!

I recommend using the same name for the function as the one you chose for the corresponding `EXTENSION_OPERATORS`-key to avoid confusion what belongs together!

The function arguments will be what the user can specify when calling this operation via `noca.Op.example_operation()`. Feel free to use default values.

Inside the function you can do whatever is necessary to make your operator work. Most likely you will want to use the `_create_operation_node`-function at some point. It works like this:

- + The first argument must be the name of the operation. It's used as a key to look up the data you stored in `EXTENSION_OPERATORS`! As mentioned: I suggest you use the same name for the `EXTENSION_OPERATORS`-key and the function name to prevent any confusion what belongs together!
- + The following arguments will be used (in order!) to connect or set the inputs specified for this operation in the `EXTENSION_OPERATORS` dictionary.
- + The created node is returned as a `noca.NcNode`-instance. It has the outputs specified in the `EXTENSION_OPERATORS` associated with it.

To properly integrate your additional operation into the NodeCalculator you must return the returned `NcNode` instance of `_create_operation_node`! That way the newly created node and its outputs can be used in further operations.

Check here to see lots of examples for the operator functions:

`OperatorMetaClass` (in `core.py`)

(Again: the `@noca_op` decorator takes care of integrating your functions into this class. No need to add the argument `"self"`.)

```
"""
```

4.5 NcValue

Module for values of base types (int, float, ...) that can store metadata.

author Mischa Kolbe <mischakolbe@gmail.com>

Note: The stored metadata on NcValues is essential to keep track of where values came from when the NodeCalculator is tracing.

When a value is queried in a NodeCalculator formula it returns an NcValue instance, which has the value-variable attached to it. For example:

```
a = noca.Node("pCube1")

with noca.Tracer(pprint_trace=True):
    a.tx.get()

# Printout:
# val1 = cmds.getAttr('pCube1.tx')
```

“val1” is the stored variable name of this queried value. When it is used in a calculation later in the formula the variable name is used instead of the value itself. For example:

```
a = noca.Node("pCube1")
b = noca.Node("pSphere1")

with noca.Tracer(pprint_trace=True):
    curr_tx = a.tx.get()
    b.ty = curr_tx

# Printout:
# val1 = cmds.getAttr('pCube1.tx')
# cmds.setAttr('pSphere1.translateY', val1)

# Rather than plugging in the queried value (making it very unclear
# where that value came from), value-variable "val1" is used instead.
```

Furthermore: Basic math operations performed on NcValues are stored, too! This allows to keep track of where values came from as much as possible:

```
a = noca.Node("pCube1")
b = noca.Node("pSphere1")

with noca.Tracer(pprint_trace=True):
    curr_tx = a.tx.get()
    b.ty = curr_tx + 2 # Adding 2 doesn't break the origin of curr_tx!

# Printout:
# val1 = cmds.getAttr('pCube1.tx')
# cmds.setAttr('pSphere1.translateY', val1 + 2) # <-- !!!

# Note that the printed trace contains the correct calculation
# including the value-variable "val1".
```

Example

```
# This works:
a = value(1, "my_metadata")

# This does NOT work:
```

(continues on next page)

(continued from previous page)

```
a = 1
a.metadata = "my_metadata"
# >>> AttributeError: 'int' object has no attribute 'metadata'
```

class nc_value.NcValue

BaseClass inherited by all NcValue-classes that are created on the fly.

Note: Only exists for inheritance check: isinstance(XYZ, NcValue) NcIntValue, NcFloatValue, etc. are otherwise hard to identify.

__weakref__

list of weak references to the object (if defined)

nc_value._concatenate_metadata(*operator*, *input_a*, *input_b*)

Concatenate the metadata for the given NcValue(s).

Note: Check docString of this module for more detail why this is important.

Parameters

- **operator** (*str*) – Name of the operator sign to be used for concatenation
- **input_a** (*NcValue* or *int* or *float* or *bool*) – First part of the operation
- **input_b** (*NcValue* or *int* or *float* or *bool*) – Second part of the operation

Returns Concatenated metadata for performed operation.**Return type** *str***Examples**

```
a = noca.Node("pCube1")
b = noca.Node("pSphere1")

with noca.Tracer(pprint_trace=True):
    curr_tx = a.tx.get()

    b.ty = curr_tx + 2

# >>> val1 = cmds.getAttr('pCube1.tx')
# >>> cmds.setAttr('pSphere1.translateY', val1 + 2) # <-- !!!
```

nc_value._create_metadata_val_class(*class_type*)

Closure to create value class of any type.

Note: Check docString of value function for more details.

Parameters **class_type** (*any builtin-type*) – Type to create a new NcValue-class for.**Returns**

New class constructor for a NcValue class of appropriate type to match given class_type

Return type NcValueClass

`nc_value.value(in_val, metadata=None, created_by_user=True)`
Create a new value with metadata of appropriate NcValue-type.

Note: For clarity: The given `in_val` is of a certain type & an appropriate type of NcValue must be used. For example: - A value of type `<int>` will become a `<NcIntValue>` - A value of type `<float>` will become a `<NcFloatValue>` - A value of type `<list>` will become a `<NcListValue>` The first time a certain NcValue class is required (meaning: if it's not in the globals yet) the function `_create_metadata_val_class` is called to create and add the necessary class to the globals. Any subsequent time that particular NcValue class is needed, the existing class constructor in the globals is used.

The reason for all this is that each created NcValue class is an instance of the appropriate base type. For example: - An instance of `<NcIntValue>` inherits from `<int>` - An instance of `<NcFloatValue>` inherits from `<float>` - An instance of `<NcListValue>` inherits from `<list>`

Parameters

- **in_val** (*any type*) – Value of any type
- **metadata** (*any type*) – Any data that should be attached to this value
- **created_by_user** (*bool*) – Whether this value was created manually

Returns

New instance of appropriate NcValue-class Read Note for details.

Return type class-instance

Examples

```
a = value(1, "some metadata")
print(a)
# >>> 1
print(a.metadata)
# >>> "some metadata"
print(a.basetype)
# >>> <type 'int'>
a.maya_node = "pCube1"  # Not only .metadata can be stored!
print(a.maya_node)
# >>> pCube1

b = value([1, 2, 3], "some other metadata")
print(b)
# >>> [1, 2, 3]
print(b.metadata)
# >>> "some other metadata"
print(b.basetype)
# >>> <type 'list'>
```

4.6 Tracer

Trace Maya commands executed by the NodeCalculator to return to the user.

Note: The actual Tracer class is in `core.py` because it is mutually linked to other classes in `core.py` and would cause cyclic references or tedious workarounds to fix.

author Mischa Kolbe <mischakolbe@gmail.com>

class `tracer.TracerMObject` (*node*, *tracer_variable*)

Class that allows to store metadata with MObjects, used for the Tracer.

Note: The Tracer uses variable names for created nodes. This class is an easy and convenient way to store these variable names with the MObject.

__init__ (*node*, *tracer_variable*)

TracerMObject-class constructor.

Parameters

- **node** (*MObject*) – Maya MObject
- **tracer_variable** (*str*) – Variable name for this MObject.

__weakref__

list of weak references to the object (if defined)

node

Get name of Maya node this TracerMObject refers to.

Returns Name of Maya node in the scene.

Return type `str`

tracer_variable

Get variable name of this TracerMObject.

Returns Variable name the NodeCalculator associated with this MObject.

Return type `str`

4.7 OmUtil

Maya utility functions. Using almost exclusively OpenMaya commands.

author Mischa Kolbe <mischakolbe@gmail.com>

Note: I am using this terminology when talking about plugs:

- **Array plug:** A plug that allows any number of connections. Example: “input3D” is the array plug of the plugs “input3D[i]”.
- **Array plug element:** A specific plug of an array plug. Example: “input3D[7]” is an array plug element of “input3D”.
- **Parent plug:** A plug that can be split into child plugs associated with it. Example: “translate” is the parent plug of [“tx”, “ty”, “tz”]
- **Child plug:** A plug that makes up part of a parent plug. Example: “translateX” is a child plug of “translate”
- **Foster child plug:** A plug that is a child of a child plug.

Example: “`ramp1.colorEntryList[0].colorR`” -> `colorR` is a foster child plug of `colorEntryList[0]`, since `color` is the child plug of `colorEntryList[0]` and `colorR` the child of `color`.

`om_util.get_all_mobjs_of_type` (*dependency_node_type*)

Get all MObjects in the current Maya scene of the given OpenMaya-type.

Parameters `dependency_node_type` (*OpenMaya.MFn*) – OpenMaya.MFn-type.

Returns List of MObjects of matching type.

Return type list

Example

```
get_all_mobjs_of_type (OpenMaya.MFn.kDependencyNode)
```

`om_util.get_array_mplug_by_index` (*mplug, index, physical=True*)

Get array element MPlug of given mplug by its physical or logical index.

Note: `.input3D` -> `.input3D[3]`

PHYSICAL INDEX: This function will NOT create a plug if it doesn’t exist! Therefore this function is particularly useful for iteration through the element plugs of an array plug.

The index can range from 0 to `numElements()` - 1.

LOGICAL INDEX: Maya will create a plug at the requested index if it doesn’t exist. This function is therefore very useful to reliably get an array element MPlug, even if that particular index doesn’t necessarily already exist.

The logical index is the sparse array index used in MEL scripts.

Parameters

- **mplug** (*MPlug*) – MPlug whose array element MPlug to get.
- **index** (*int*) – Index of array element plug.
- **physical** (*bool*) – Look for element at physical index. If set to False it will look for the logical index!

Returns MPlug at the requested index.

Return type MPlug or None

`om_util.get_array_mplug_elements` (*mplug*)

Get all array element MPlugs of the given mplug.

Note: `.input3D` -> `.input3D[0]`, `.input3D[1]`, ...

Parameters **mplug** (*MPlug*) – MPlug whose array element MPlugs to get.

Returns List of array element MPlugs.

Return type list

`om_util.get_attr_of_mobj(mobj, attr)`
Get value of attribute on MObject.

Note: Basically `cmds.getAttr()` that works with MObjects.

Parameters

- **mobj** (*MObject or MDagPath or str*) – Node whose attr should be queried.
- **attr** (*str*) – Name of attribute.

Returns Value of queried plug.

Return type list or tuple or int or float or bool or str

`om_util.get_child_mplug(mplug, child)`
Get the child MPlug of the given mplug.

Note: `.t -> .tx`

Parameters

- **mplug** (*MPlug*) – MPlug whose child MPlug to get.
- **child** (*str*) – Name of the child plug

Returns Child MPlug or None if that doesn't exist.

Return type MPlug or None

`om_util.get_child_mplugins(mplug)`
Get all child MPlugs of the given mplug.

Note: `.t -> [.tx, .ty, .tz]`

Parameters **mplug** (*MPlug*) – MPlug whose child MPlugs to get.

Returns List of child MPlugs.

Return type list

`om_util.get_dag_path_of_mobj(mobj, full=False)`
Get the dag path of an MObject. Either partial or full.

Note: The flag “full” defines whether a full or partial DAG path should be returned. “Partial” simply means the shortest unique DAG path to describe the given MObject. “Full” always gives the entire DAG path.

Parameters

- **mobj** (*MObject or MDagPath or str*) – Node whose long name is requested.
- **full** (*bool*) – Return either the entire or partial DAG path. See Note.

Returns DAG path to the given MObject.

Return type str

`om_util.get_mdag_path(mobj)`

Get an MDagPath from the given mobj.

Parameters `mobj` (*MObject or MDagPath or str*) – MObject to get MDagPath for.

Returns MDagPath of the given mobj.

Return type MDagPath

`om_util.get_mfn_dag_node(node)`

Get an MFnDagNode of the given node.

Parameters `node` (*MObject or MDagPath or str*) – Node to get MFnDagNode for.

Returns MFnDagNode of the given node.

Return type MFnDagNode

`om_util.get_mobj(node)`

Get the MObject of the given node.

Parameters `node` (*MObject or MDagPath or str*) – Maya node requested as an MObject.

Raises `RuntimeError` – If the given string doesn't represent a unique, existing Maya node in the scene.

Returns MObject instance that is a reference to the given node.

Return type MObject

`om_util.get_mplug_of_mobj(mobj, attr)`

Get MPlug from the given MObject and attribute combination.

Parameters

- `mobj` (*MObject*) – MObject of node.
- `attr` (*str*) – Name of attribute on node.

Returns MPlug of the given node/attr combination.

Return type MPlug

`om_util.get_mplug_of_node_and_attr(node, attr_str, expand_to_shape=True, __shape_lookup=False)`

Get an MPlug to the given node & attr combination.

Parameters

- `node` (*MObject or MDagPath or str*) – Node whose attr should be queried.
- `attr_str` (*str*) – Name of attribute.
- `expand_to_shape` (*bool*) – If the given node is a transform with shape nodes underneath it; check for the attribute on the shape node, if it can't be found on the transform. Defaults to True.
- `__shape_lookup` (*bool*) – Flag to specify that an automatic lookup due to `expand_to_shape` is taking place. This must never be set by the user! Defaults to False.

Returns MPlug of given node.attr or None if that doesn't exist.

Return type MPlug or None

Raises `RuntimeError` – If the desired mplug does not exist.

`om_util.get_mplug_of_plug(plug)`

Get the MPlug to any given plug.

Parameters `plug` (*MPlug or str*) – Name of plug; “name.attr”

Returns MPlug of the requested plug.

Return type MPlug

`om_util.get_name_of_mobj(mobj)`

Get the name of an MObject.

Parameters `mobj` (*MObject*) – MObject whose name is requested.

Returns Name of given MObject.

Return type str

`om_util.get_node_type(node, api_type=False)`

Get the type of the given Maya node.

Note: More versatile version of `cmds.nodeType()`

Parameters

- **node** (*MObject or MDagPath or str*) – Node whose type should be queried.
- **api_type** (*bool*) – Return Maya API type.

Returns Type of Maya node

Return type str

`om_util.get_parent(node)`

Get parent of the given node.

Note: More versatile version of `cmds.listRelatives(node, parent=True)[0]`

Parameters `node` (*MObject or MDagPath or str*) – Node to get parent of.

Returns Name of node’s parent.

Return type str

`om_util.get_parent_mplug(mplug)`

Get the parent MPlug of the given mplug.

Note: `.tx -> .t`

Parameters `mplug` (*MPlug*) – MPlug whose parent MPlug to get.

Returns Parent MPlug or None if that doesn’t exist.

Return type MPlug or None

`om_util.get_parents(node)`

Get parents of the given node.

Parameters `node` (*MObject* or *MDagPath* or *str*) – Node whose parents are queried.

Returns Name of parents in an ascending list: First parent first.

Return type list

`om_util.get_selected_nodes_as_mobjs()`

Get all currently selected nodes in the scene as MObjects.

Returns List of MObjects of the selected nodes in the Maya scene.

Return type list

`om_util.get_shape_mobjs(mobj)`

Get the shape MObjects of a given MObject.

Parameters `mobj` (*MObject* or *MDagPath* or *str*) – MObject whose shapes are requested.

Returns List of MObjects of the shapes of given MObject.

Return type list

`om_util.get_unique_mplug_path(mplug)`

Get a unique path to the given MPlug.

Note: MPlug instances don't return unique paths by default. Therefore they don't support non unique items. This is a work-around for that issue.

Parameters `mplug` (*str* or *MPlug*) – Name or MPlug of attribute.

Returns Unique path to attribute: `node.attribute`

Return type str

Raises `ValueError` – If given object is neither a string or an MPlug instance.

`om_util.is_instanced(node)`

Check if a Maya node is instantiated.

Parameters `node` (*MObject* or *MDagPath* or *str*) – Node to check if it's instantiated.

Returns Whether given node is instantiated.

Return type bool

`om_util.is_valid_mplug(mplug)`

Check whether given mplug is a valid MPlug.

Parameters `mplug` (*MObject* or *MPlug* or *MDagPath* or *str*) – Potential MPlug.

Returns True if given mplug actually is an MPlug instance.

Return type bool

`om_util.rename_mobj(mobj, name)`

Rename the given MObject.

Note: This is currently NOT undoable!!! Therefore be careful!

Parameters

- **mobj** (*MObject*) – Node to be renamed.
- **name** (*str*) – New name for the node.

Returns New name of renamed mobj

Return type *str*

`om_util.select_mobjjs` (*mobjjs*)

Select the given MObjects in the Maya scene.

Parameters **mobjjs** (*list*) – List of MObjects to be selected.

Todo:

For some reason the version below (utilizing OpenMaya-methods) only selects the nodes, but they don't get selected in the outliner or viewport! Therefore using the cmds-version for now.

```
m_selection_list = OpenMaya.MSelectionList()
for mobj in mobjjs:
    m_selection_list.add(mobj)
OpenMaya.MGlobal.setActiveSelectionList(
    m_selection_list,
    OpenMaya.MGlobal.kReplaceList
)
return m_selection_list
```

`om_util.set_mobj_attribute` (*mobj*, *attr*, *value*)

Set attribute on given MObject to the given value.

Note: Basically `cmds.setAttr()` that works with MObjects.

Parameters

- **mobj** (*MObject* or *MDagPath* or *str*) – Node whose attribute should be set.
- **attr** (*str*) – Name of attribute.
- **value** (*int* or *float* or *bool* or *str*) – Value plug should be set to.

Todo: Use OpenMaya API only! Check: austinjbaker.com/mplugins-setting-values

`om_util.split_attr_string` (*attr*)

Split string referring to an attr on a Maya node into its elements.

Note: “`attr_a[attr_a_index].attr_b[attr_b_index]. ...`” -> [(`attr_a`, `attr_a_index`), (`attr_b`, `attr_b_index`), ...]

The index for an attribute that has no index will be `None`.

Parameters **attr** (*str*) – Name of attribute.

Returns List of tuples of the form (attribute, attribute-index)

Return type list

Raises `ValueError` – If given string is not in the pattern described in Note.

`om_util.split_node_string(node)`

Split string referring to a Maya node name into its separate elements.

Note: “namespace1:namespace2:someldag|path|node” -> (namespaces, dag_path, node)

Parameters `node` (*str*) – Name of Maya node, potentially with namespaces & dagPath.

Returns Tuple of the form (namespaces, dag_path, node)

Return type tuple

Raises

- `ValueError` – If given string is not in the pattern described in Note.
- `RuntimeError` – If the given string looks like it stands for multiple Maya-nodes; a string that yields multiple regex-matches.

`om_util.split_plug_string(plug)`

Split string referring to a plug into its separate elements.

Note: “namespace:someldag|path|node.attr_a[attr_a_index].attr_b” -> (namespace, dag_path, node, [(attr_a, attr_a_index), (attr_b, None)])

Parameters `plug` (*str*) – Name of plug; “name.attr”

Returns Tuple of elements that make up plug (namespace, dag_path, node, attrs)

Return type tuple

4.8 Logging

Module for logging.

author Mischa Kolbe <mik@dneg.com>

credits Steven Bills, Mischa Kolbe

class `logger.NullHandler` (*level=0*)

Basic custom logging handler.

emit (*record*)

Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a `NotImplementedError`.

`logger.clear_handlers()`

Reset handlers of logger.

Note: This prevents creating multiple handler copies when using `reload(logger)`.

`logger.setup_file_handler(file_path, max_bytes=104857600, level=20)`

Creates a rotating file handler for logging.

Default level is info.

Parameters

- **file_path** (*str*) – Path where to save the log to.
- **max_bytes** (*int*) – Maximum size of output file.
- **level** (*int*) – Desired logging level. Default is logging.INFO.

max_bytes: $x \ll y$ Returns x with the bits shifted to the left by y places. $100 \ll 20 == 100 * 2^{20}$

`logger.setup_stream_handler(level=20)`

Create a stream handler for logging.

Note: Logging levels are: DEBUG, INFO, WARN, ERROR, CRITICAL

Parameters **level** (*int*) – Desired logging level. Default is logging.INFO.

5.1 Release 2.1.5

5.1.1 Features added

- Added getattr, attr and set methods to NcList class. > [node_calculator/issues/93](#)
- **Added trigonometry operators:** > [node_calculator/issues/94](#)
 - Thanks for the basic idea, Chad Vernon! <https://www.chadvernon.com/blog/trig-maya/>
 - sin
 - cos
 - tan
 - asin
 - acos
 - atan
 - atan2

5.1.2 Bugs fixed

- Fixed MPlug retrieval of indexed AND aliased attrs (such as blendshape target weights) > [node_calculator/issues/91](#)
- Including foster children in MPlug search (such as ramp_node.colorEntryList[0].colorR) > [node_calculator/issues/92](#)

5.2 Release 2.1.4

5.2.1 Bugs fixed

- remap_value now accepts NoCa nodes for value-arg > [node_calculator/issues/95](#)

5.3 Release 2.1.3

5.3.1 Features added

- added curve_info Operator. > [node_calculator/issues/82](#) & [87](#)
- added reset_cleanup function to reset the cleanup-stack. > [node_calculator/issues/83](#)

5.3.2 Bugs fixed

- More robust shape node creation and naming. > [node_calculator/issues/86](#)
- More descriptive error message when node doesn't exist or isn't unique. > [node_calculator/issues/84](#)
- PyMel is only loaded when necessary. > [node_calculator/issues/85](#)

5.4 Release 2.1.2

5.4.1 Features added

- Added the following operators: > [node_calculator/issues/80](#)
 - sum
 - quatAdd
 - quatConjugate
 - quatInvert
 - quatNegate
 - quatNormalize
 - quatProd
 - quatSub
 - quatToEuler
 - eulerToQuat
 - holdMatrix
 - reverse
 - passMatrix
 - remapColor
 - remapHsv

- rgbToHsv
 - wtAddMatrix
 - closestPointOnMesh
 - closestPointOnSurface
 - pointOnSurfaceInfo
 - pointOnCurveInfo
 - nearestPointOnCurve
 - fourByFourMatrix
- Operator unittests are more generic now: A dictionary contains which inputs/outputs to use for each Operators test.
 - Added more unittests for some issues that came up: non-unique node names, aliased attributes, accessing shape-attributes through the transform (see Features added in Release 2.1.1). > [node_calculator/issues/76](#)

5.4.2 Bugs fixed

- `sum()`, `average()` and `mult_matrix()` operators now work correctly when given lists/tuples/NcLists as args.

5.5 Release 2.1.1

5.5.1 Bugs fixed

- Now supports non-unique names > [node_calculator/issues/74](#)
- Catch error when user sets a non-existent attribute on an NcList item (now only throws a warning) > [node_calculator/issues/73](#)

5.6 Release 2.1.0

5.6.1 Incompatible changes

- Careful: The actual NodeCalculator now lives in the `node_calculator` INSIDE the main repo! It's not at the top level anymore.
- The `decompose_matrix` and `pair_blend` Operators now have a “`return_all_outputs`”-flag. By default they return an NcNode now, not all outputs in an NcList! > [node_calculator/issues/67](#)

5.6.2 Features added

- Tests are now standalone (not dependent on CMT anymore) and can be run from a console! Major kudos to Andres Weber!
- CircleCi integration to auto-run checks whenever repo is updated. Again: Major kudos to Andres Weber!
- The default Operators are now factored out into their own files: `base_functions.py` & `base_operators.py` > [node_calculator/issues/59](#)

- It's now possible to set attributes on the shape from the transform (mimicking Maya behaviour). Sudo example: `pCube1.outMesh` (instead of requiring `pCube1Shape.outMesh`) > [node_calculator/issues/69](#)
- The `noca.cleanup(keep_selected=False)` function allows to delete all nodes created by the NodeCalculator to unclutter heavy prototyping scenes. > [node_calculator/issues/63](#)

5.6.3 Bugs fixed

- The dot-Operator now correctly returns a 1D result (returned a 3D result before) > [node_calculator/issues/68](#)

5.7 Release 2.0.1

5.7.1 Bugs fixed

- Aliased attributes can now be accessed (`om_util.get_mplug_of_mobj` couldn't find them before)
- Operation values of zero are now set correctly (they were ignored)

5.8 Release 2.0.0

5.8.1 Dependencies

5.8.2 Incompatible changes

- “output” is now “outputs” in `lookup_table.py`!
- `OPERATOR_LOOKUP_TABLE` is now `OPERATORS`
- `multi_input` & `multi_output` doesn't have to be declared anymore! The tag “{array}” will cause an input/output to be interpreted as multi.

5.8.3 Deprecated

- Container support. It wasn't properly implemented and Maya containers are not useful (imo).

5.8.4 Features added

- Easy to add custom/proprietary nodes via extension
- Convenience functions for transforms, locators & `create_node`.
- `auto Consolidate` & `auto unravel` can be turned off (globally & individually)
- Indexed attributes now possible (still a bit awkward, but hey..)
- Many additional operators.
- Documentation; NoCa v2 cheat sheet!
- `om_util` with various OpenMaya functions
- Many other small improvements.

- Any attr type can now be created.
- Attribute separator convenience function added. Default values can be specified in config.py.
- config.py to make it easy and clear where to change basic settings.
- Default extension for [Serguei Kalentchouk's maya_math_nodes](#)
- Tests added, using [Chad Vernon's test suite](#)

5.8.5 Bugs fixed

- Uses MObjects and MPlugs to reference to Maya nodes and attributes; Renaming of objects, attributes with index, etc. are no longer an issue.
- Cleaner code; Clear separation of classes and their functionality (NcList, NcNode, NcAttrs, NcValue)
- Any child attribute will be consolidated (array, normal, ..)
- Tracer now stores values as variables (from get() or so)
- Conforms pretty well to PEP8 (apart from tests)

5.8.6 Testing

5.8.7 Features removed

5.9 Release 1.0.0

- First working version: Create, connect and set Maya nodes with Python commands.

CHAPTER 6

Overview

- `genindex`

b

`base_functions`, [76](#)

`base_operators`, [57](#)

c

`core`, [26](#)

l

`logger`, [92](#)

n

`nc_value`, [81](#)

o

`om_util`, [85](#)

t

`tracer`, [84](#)

Symbols

- `__add__()` (*core.NcBaseClass* method), 29
- `__copy__()` (*core.NcList* method), 39
- `__deepcopy__()` (*core.NcList* method), 39
- `__delitem__()` (*core.NcList* method), 39
- `__div__()` (*core.NcBaseClass* method), 29
- `__enter__()` (*core.Tracer* method), 47
- `__eq__()` (*core.NcBaseClass* method), 29
- `__exit__()` (*core.Tracer* method), 47
- `__ge__()` (*core.NcBaseClass* method), 30
- `__getattr__()` (*core.NcAttrs* method), 27
- `__getattr__()` (*core.NcList* method), 39
- `__getattr__()` (*core.NcNode* method), 44
- `__getitem__()` (*core.NcAttrs* method), 28
- `__getitem__()` (*core.NcList* method), 40
- `__getitem__()` (*core.NcNode* method), 44
- `__gt__()` (*core.NcBaseClass* method), 30
- `__init__()` (*core.NcAttrs* method), 28
- `__init__()` (*core.NcBaseClass* method), 30
- `__init__()` (*core.NcBaseNode* method), 34
- `__init__()` (*core.NcList* method), 40
- `__init__()` (*core.NcNode* method), 44
- `__init__()` (*core.Node* method), 46
- `__init__()` (*core.OperatorMetaClass* method), 46
- `__init__()` (*core.Tracer* method), 47
- `__init__()` (*tracer.TracerMObject* method), 85
- `__iter__()` (*core.NcBaseNode* method), 34
- `__iter__()` (*core.NcList* method), 40
- `__le__()` (*core.NcBaseClass* method), 30
- `__len__()` (*core.NcBaseNode* method), 34
- `__len__()` (*core.NcList* method), 40
- `__load_extension()` (*in module core*), 47
- `__load_extensions()` (*in module core*), 48
- `__lt__()` (*core.NcBaseClass* method), 30
- `__mul__()` (*core.NcBaseClass* method), 30
- `__ne__()` (*core.NcBaseClass* method), 31
- `__neg__()` (*core.NcBaseClass* method), 31
- `__pos__()` (*core.NcBaseClass* method), 31
- `__pow__()` (*core.NcBaseClass* method), 31
- `__radd__()` (*core.NcBaseClass* method), 31
- `__rdiv__()` (*core.NcBaseClass* method), 31
- `__repr__()` (*core.NcBaseNode* method), 34
- `__repr__()` (*core.NcList* method), 40
- `__reversed__()` (*core.NcList* method), 41
- `__rmul__()` (*core.NcBaseClass* method), 32
- `__rpow__()` (*core.NcBaseClass* method), 32
- `__rsub__()` (*core.NcBaseClass* method), 32
- `__setattr__()` (*core.NcBaseNode* method), 34
- `__setattr__()` (*core.NcList* method), 41
- `__setitem__()` (*core.NcBaseNode* method), 35
- `__setitem__()` (*core.NcList* method), 41
- `__str__()` (*core.NcBaseNode* method), 35
- `__str__()` (*core.NcList* method), 41
- `__sub__()` (*core.NcBaseClass* method), 32
- `__weakref__` (*core.NcBaseClass* attribute), 32
- `__weakref__` (*core.NcList* attribute), 42
- `__weakref__` (*core.Node* attribute), 46
- `__weakref__` (*core.OperatorMetaClass* attribute), 47
- `__weakref__` (*core.Tracer* attribute), 47
- `__weakref__` (*nc_value.NcValue* attribute), 83
- `__weakref__` (*tracer.TracerMObject* attribute), 85
- `_add_all_add_attr_methods()`
(*core.NcBaseNode* method), 35
- `_add_to_command_stack()` (*core.NcBaseClass*
class method), 32
- `_add_to_node_bin()` (*in module core*), 48
- `_add_to_traced_nodes()` (*core.NcBaseClass*
class method), 32
- `_add_to_traced_values()` (*core.NcBaseClass*
class method), 33
- `_add_traced_attr()` (*core.NcBaseNode* method),
36
- `_auto_consolidate` (*core.NcAttrs* attribute), 28
- `_auto_unravel` (*core.NcAttrs* attribute), 28
- `_check_for_parent_attribute()` (*in module*
core), 48
- `_compare()` (*core.NcBaseClass* method), 33
- `_concatenate_metadata()` (*in module nc_value*),
83

`_consolidate_plugs_to_min_dimension()`
(in module *core*), 48

`_convert_item_to_nc_instance()`
(*core.NcList* static method), 42

`_create_metadata_val_class()` (in module *nc_value*), 83

`_create_node_name()` (in module *core*), 48

`_create_operation_node()` (in module *core*), 49

`_create_traced_operation_node()` (in module *core*), 49

`_define_add_attr_method()` (*core.NcBaseNode* method), 36

`_extension_operators_init()` (in module *base_operators*), 57

`_flush_command_stack()` (*core.NcBaseClass* class method), 33

`_flush_traced_nodes()` (*core.NcBaseClass* class method), 33

`_flush_traced_values()` (*core.NcBaseClass* class method), 33

`_format_docstring()` (in module *core*), 49

`_get_next_value_name()` (*core.NcBaseClass* class method), 33

`_get_next_variable_name()` (*core.NcBaseClass* class method), 33

`_get_node_inputs()` (in module *core*), 49

`_get_node_outputs()` (in module *core*), 51

`_get_tracer_variable_for_node()`
(*core.NcBaseClass* class method), 33

`_held_attrs` (*core.NcNode* attribute), 45

`_held_attrs_list` (*core.NcAttrs* attribute), 28

`_holder_node` (*core.NcAttrs* attribute), 28

`_initialize_trace_variables()`
(*core.NcBaseClass* class method), 34

`_is_consolidation_allowed()` (in module *core*), 51

`_is_valid_maya_attr()` (in module *core*), 51

`_join_cmds_kwargs()` (in module *core*), 52

`_node_mobj` (*core.NcAttrs* attribute), 28

`_node_mobj` (*core.NcNode* attribute), 45

`_set_or_connect_a_to_b()` (in module *core*), 52

`_split_plug_into_node_and_attr()` (in module *core*), 52

`_traced_add_attr()` (in module *core*), 52

`_traced_connect_attr()` (in module *core*), 52

`_traced_create_node()` (in module *core*), 53

`_traced_get_attr()` (in module *core*), 53

`_traced_set_attr()` (in module *core*), 53

`_unravel_and_set_or_connect_a_to_b()` (in module *core*), 53

`_unravel_base_node_instance()` (in module *core*), 54

`_unravel_item()` (in module *core*), 54

`_unravel_item_as_list()` (in module *core*), 54

`_unravel_list()` (in module *core*), 55

`_unravel_nc_list()` (in module *core*), 55

`_unravel_plug()` (in module *core*), 55

`_unravel_str()` (in module *core*), 55

A

`acos()` (in module *base_functions*), 76

`add_enum()` (*core.NcBaseNode* method), 36

`add_int()` (*core.NcBaseNode* method), 36

`add_separator()` (*core.NcBaseNode* method), 37

`angle_between()` (in module *base_operators*), 58

`append()` (*core.NcList* method), 42

`asin()` (in module *base_functions*), 76

`atan()` (in module *base_functions*), 77

`atan2()` (in module *base_functions*), 77

`attr()` (*core.NcBaseNode* method), 37

`attr()` (*core.NcList* method), 42

`attrs` (*core.NcAttrs* attribute), 29

`attrs` (*core.NcNode* attribute), 45

`attrs_list` (*core.NcAttrs* attribute), 29

`attrs_list` (*core.NcNode* attribute), 45

`auto_state()` (*core.NcBaseNode* method), 37

`available()` (*core.OperatorMetaClass* method), 47

`average()` (in module *base_operators*), 58

B

`base_functions` (module), 76

`base_operators` (module), 57

`blend()` (in module *base_operators*), 58

C

`choice()` (in module *base_operators*), 59

`clamp()` (in module *base_operators*), 59

`cleanup()` (in module *core*), 55

`clear_handlers()` (in module *logger*), 92

`closest_point_on_mesh()` (in module *base_operators*), 60

`closest_point_on_surface()` (in module *base_operators*), 60

`compose_matrix()` (in module *base_operators*), 60

`condition()` (in module *base_operators*), 61

`core` (module), 26

`cos()` (in module *base_functions*), 77

`create_node()` (in module *core*), 56

`cross()` (in module *base_operators*), 62

`curve_info()` (in module *base_operators*), 62

D

`decompose_matrix()` (in module *base_operators*), 62

`dot()` (in module *base_operators*), 63

E

`emit()` (*logger.NullHandler* method), 92

euler_to_quat() (in module *base_operators*), 63
 exp() (in module *base_operators*), 63
 extend() (*core.NcList* method), 42

F

four_by_four_matrix() (in module *base_operators*), 64

G

get() (*core.NcBaseNode* method), 38
 get() (*core.NcList* method), 42
 get_all_mobjs_of_type() (in module *om_util*), 86
 get_array_mplug_by_index() (in module *om_util*), 86
 get_array_mplug_elements() (in module *om_util*), 86
 get_attr_of_mobj() (in module *om_util*), 86
 get_child_mplug() (in module *om_util*), 87
 get_child_mplugs() (in module *om_util*), 87
 get_dag_path_of_mobj() (in module *om_util*), 87
 get_mdag_path() (in module *om_util*), 88
 get_mfn_dag_node() (in module *om_util*), 88
 get_mobj() (in module *om_util*), 88
 get_mplug_of_mobj() (in module *om_util*), 88
 get_mplug_of_node_and_attr() (in module *om_util*), 88
 get_mplug_of_plug() (in module *om_util*), 88
 get_name_of_mobj() (in module *om_util*), 89
 get_node_type() (in module *om_util*), 89
 get_parent() (in module *om_util*), 89
 get_parent_mplug() (in module *om_util*), 89
 get_parents() (in module *om_util*), 89
 get_selected_nodes_as_mobjs() (in module *om_util*), 90
 get_shape_mobjs() (in module *om_util*), 90
 get_shapes() (*core.NcBaseNode* method), 38
 get_unique_mplug_path() (in module *om_util*), 90

H

hold_matrix() (in module *base_operators*), 64

I

insert() (*core.NcList* method), 43
 inverse_matrix() (in module *base_operators*), 65
 is_instanced() (in module *om_util*), 90
 is_valid_mplug() (in module *om_util*), 90

L

length() (in module *base_operators*), 65
 locator() (in module *core*), 56
 logger (module), 92

M

matrix_distance() (in module *base_operators*), 65
 mult_matrix() (in module *base_operators*), 66

N

nc_value (module), 81
 NcAttrs (class in *core*), 27
 NcBaseClass (class in *core*), 29
 NcBaseNode (class in *core*), 34
 NcList (class in *core*), 39
 NcNode (class in *core*), 43
 NcValue (class in *nc_value*), 83
 nearest_point_on_curve() (in module *base_operators*), 66
 noca_op() (in module *core*), 56
 Node (class in *core*), 45
 node (*core.NcAttrs* attribute), 29
 node (*core.NcList* attribute), 43
 node (*core.NcNode* attribute), 45
 node (*tracer.TracerMObject* attribute), 85
 nodes (*core.NcBaseNode* attribute), 38
 nodes (*core.NcList* attribute), 43
 normalize_vector() (in module *base_operators*), 66
 NullHandler (class in *logger*), 92

O

om_util (module), 85
 OperatorMetaClass (class in *core*), 46

P

pair_blend() (in module *base_operators*), 67
 pass_matrix() (in module *base_operators*), 67
 plugs (*core.NcBaseNode* attribute), 38
 point_matrix_mult() (in module *base_operators*), 68
 point_on_curve_info() (in module *base_operators*), 68
 point_on_surface_info() (in module *base_operators*), 68
 pow() (in module *base_operators*), 69

Q

quat_add() (in module *base_operators*), 69
 quat_conjugate() (in module *base_operators*), 70
 quat_invert() (in module *base_operators*), 70
 quat_mul() (in module *base_operators*), 70
 quat_negate() (in module *base_operators*), 70
 quat_normalize() (in module *base_operators*), 71
 quat_sub() (in module *base_operators*), 71
 quat_to_euler() (in module *base_operators*), 71

R

remap_color() (in module *base_operators*), 72

`remap_hsv()` (in module `base_operators`), 72
`remap_value()` (in module `base_operators`), 73
`rename_mobj()` (in module `om_util`), 90
`reset_cleanup()` (in module `core`), 56
`reverse()` (in module `base_operators`), 74
`rgb_to_hsv()` (in module `base_operators`), 74

S

`select_mobjs()` (in module `om_util`), 91
`set()` (`core.NcBaseNode` method), 38
`set()` (`core.NcList` method), 43
`set_auto_consolidate()` (`core.NcBaseNode` method), 38
`set_auto_unravel()` (`core.NcBaseNode` method), 39
`set_global_auto_consolidate()` (in module `core`), 56
`set_global_auto_unravel()` (in module `core`), 57
`set_mobj_attribute()` (in module `om_util`), 91
`set_range()` (in module `base_operators`), 74
`setup_file_handler()` (in module `logger`), 92
`setup_stream_handler()` (in module `logger`), 93
`sin()` (in module `base_functions`), 78
`soft_approach()` (in module `base_functions`), 78
`split_attr_string()` (in module `om_util`), 91
`split_node_string()` (in module `om_util`), 92
`split_plug_string()` (in module `om_util`), 92
`sqrt()` (in module `base_operators`), 75
`sum()` (in module `base_operators`), 75

T

`tan()` (in module `base_functions`), 79
`to_py_node()` (`core.NcBaseNode` method), 39
`Tracer` (class in `core`), 47
`tracer` (module), 84
`tracer_variable` (`tracer.TracerMObject` attribute), 85
`TracerMObject` (class in `tracer`), 85
`transform()` (in module `core`), 57
`transpose_matrix()` (in module `base_operators`), 75

V

`value()` (in module `nc_value`), 84

W

`weighted_add_matrix()` (in module `base_operators`), 76