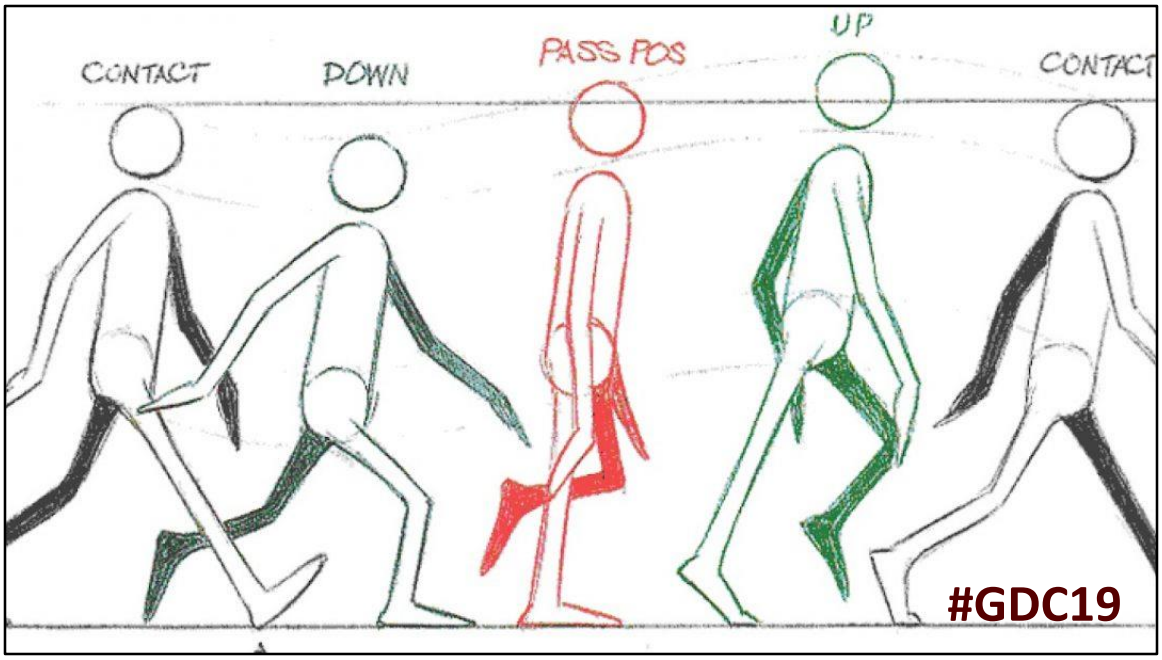
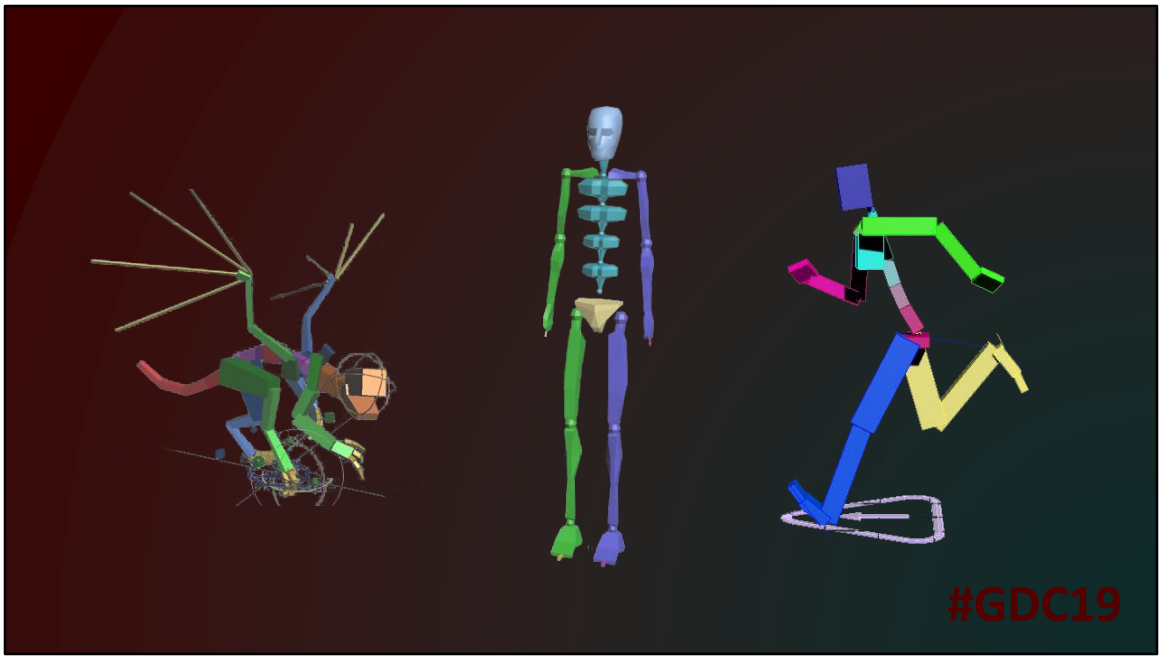
How we feeling GDC?

Metanodes!!!!
Not great at **elegant** or **concise** subtitles so welcome to,
Metanodes: The Dual Power of Metanodes in Maya!
Airhorn

I'm Andrew
Senior Technical Artist at ArenaNet
We make **Guild Wars**
I've been developing our **componentized rigging system**.



Started in **2007** at **GPG** as an **animator**
Semi technical **building rigs** with third **party tools**

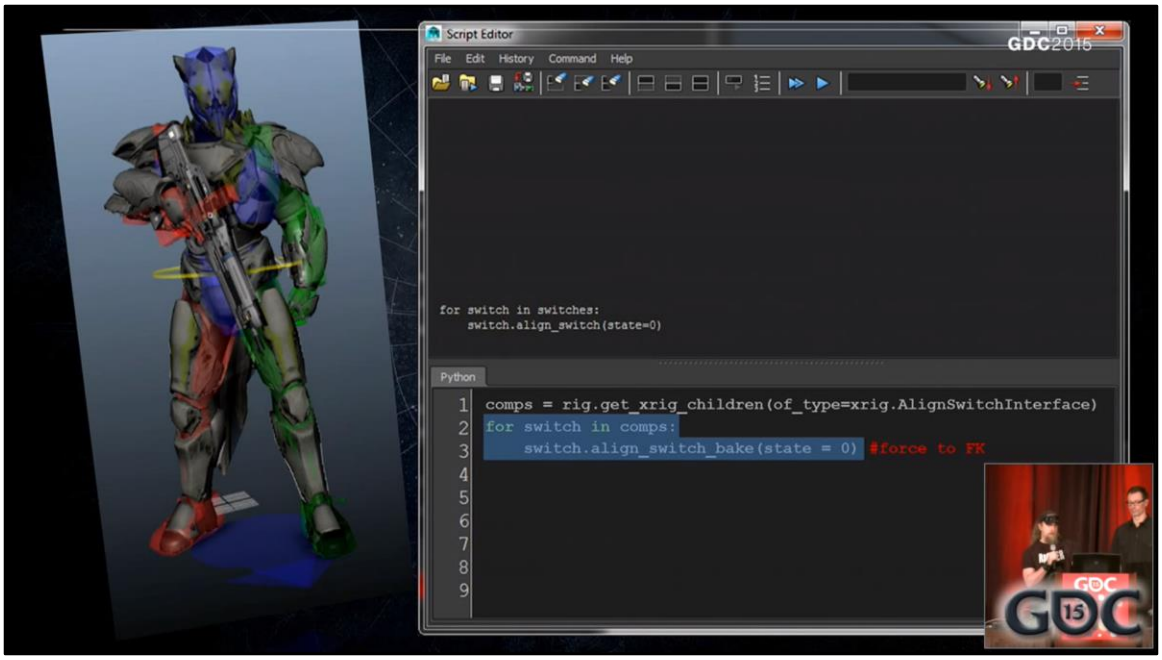


Puppetshop
Biped rawr Quaternions
CAT infinity
Lots of solid **features**
Blackbox rig solutions make it **hard to solve systemic** issues

I’m going to **switch to Maya**
Make **my own rigs** and they **will be amazing**
Turns out it **takes awhile**
Time to **learn Python** and make an **auto-rigger**

We **shipped** a VR game
It worked, but I **tracked no data** in the scene.
Time for **version 2.0** and this time lets **manage the data**

SEG; One of my inspirations



2015 GDC presentation **Tools Based Rigging in Bungie's Destiny**
David **Hunt** and Forrest **Söderlind**
Half way through a 10 minute segment
Coding Rig Components in Python
Starting to build it myself, but I couldn't get over some initial hurdles

SEG; How did I figure it out? I went to **work with people smarter** than me.

Be the dumbest person on your team.

#GDC19

Can’t **recommend** being with smart people enough.
I **got a job at ArenaNet** but you could **also hire people**.

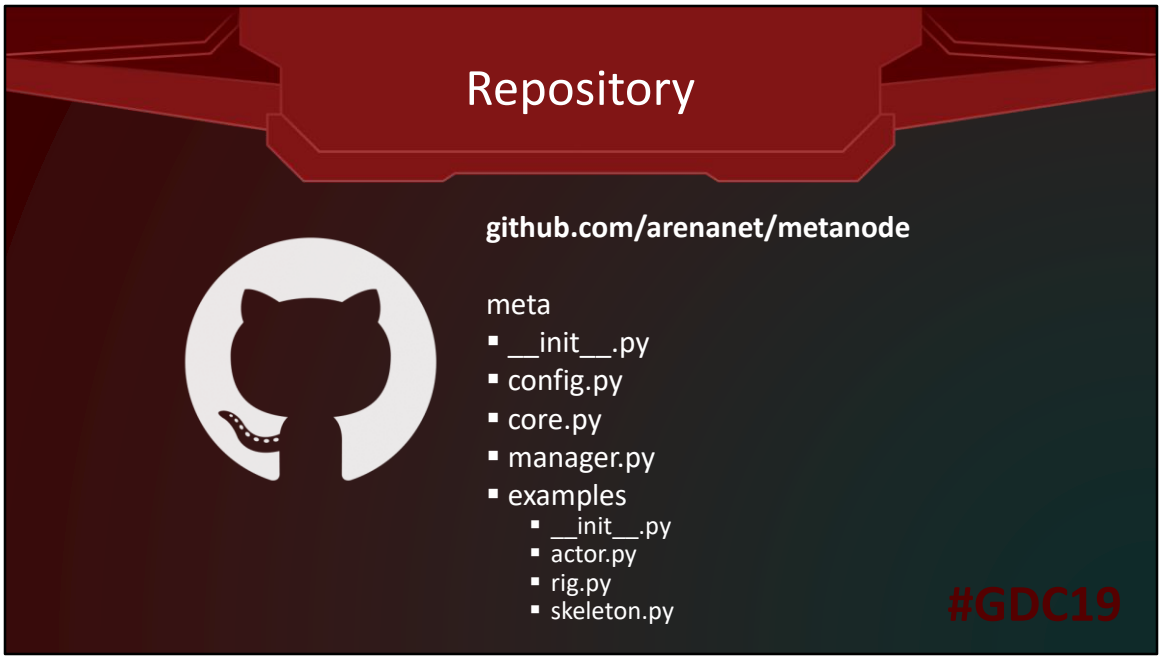
SEG; Working with smart people is the **best way to better myself**.

It’s worked in my **career**.

And, in my **marriage**.

Getting to ArenaNet gave me access to very **talented technical artists**
Examples from the team; Tim's RBF solvers. Dan's MoCap pipeline. Austin's Maya API libraries
And **whole codebase** of **great ideas**, one being **Metanodes**
Credit Kyle Mistlin-Rude

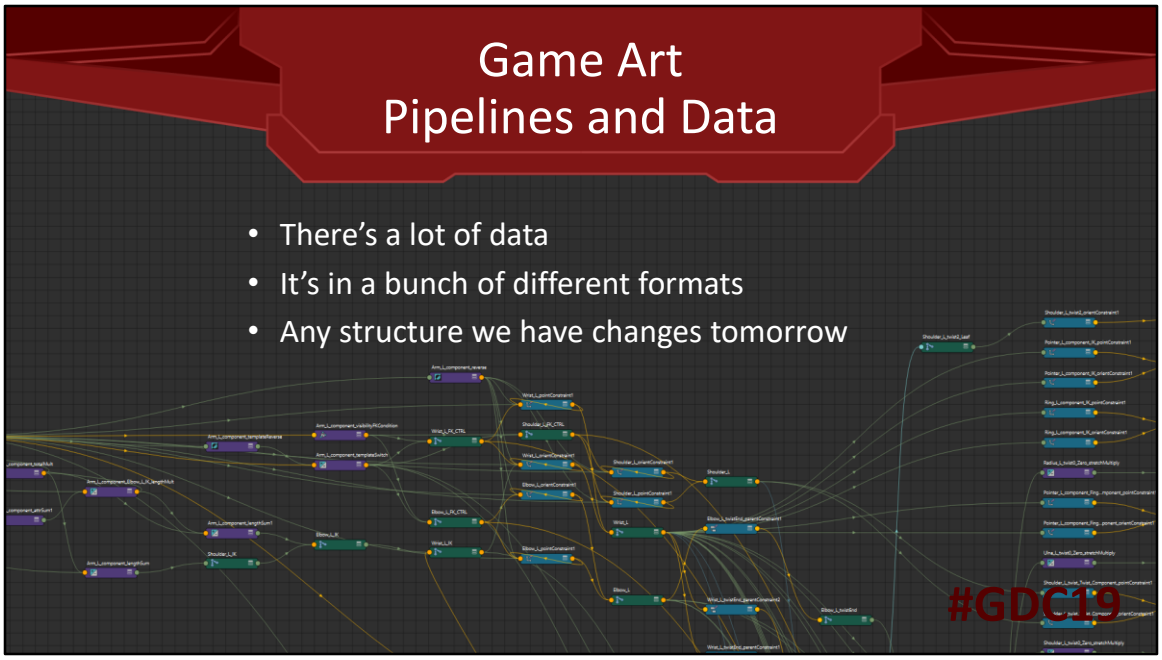
SEG; **Not just talk**, I want to **leave you with some code**.



Everything you need up on **ArenaNet Github**
I've built a **generic version** of our Metanode classes
Conformed it to PEP8
Set of **example** modules
Encourage **forking the depot** and exploring other ideas

SEG; Lets get started

My experience is **half of technical art is sheepherding data** from one place to another



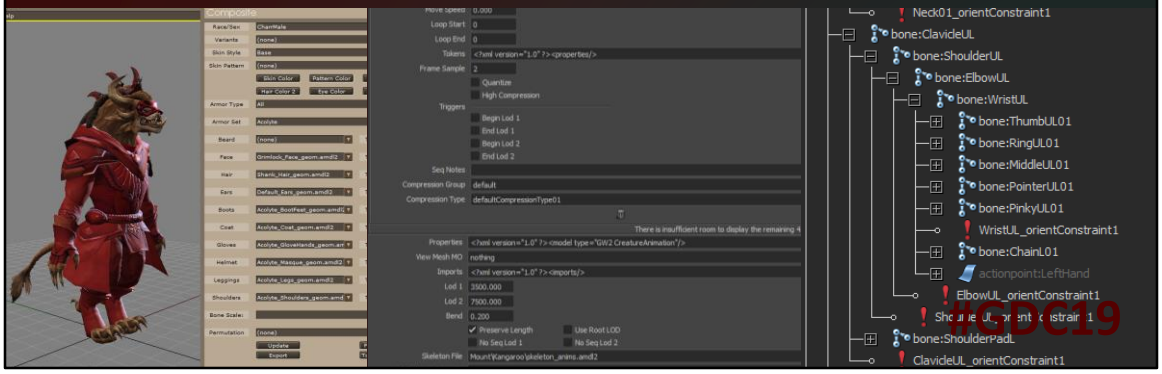
So much data we store in our Maya pipeline.
Data **specific to the pipeline** we've developed

It could be dumps of JSON, transforms, rig definitions, joint mappings, scene state flags

As the **pipeline develops** or other teams make **requests**, that **data needs to evolve**

Less-Optimal Structures

Magic Naming Schemes | Custom Attributes | Namespace Madness

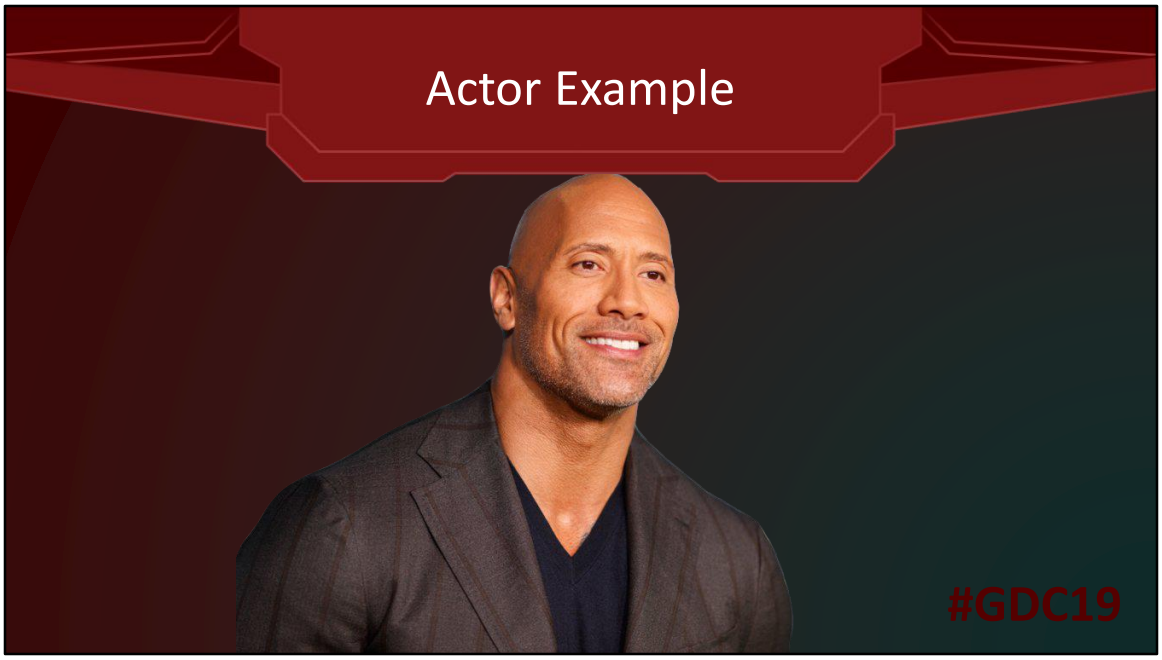


There are many **less optimal** ways to store data
I've relied on some of these structures in the past but **found drawbacks later**
The **decade old GW2** pipeline had some of these data schemes

Naming schemes get **rigid structures** in place that can be hard to adapt later
Examples; _LOD0

Custom Attributes great at storing data but offer **no functionality**

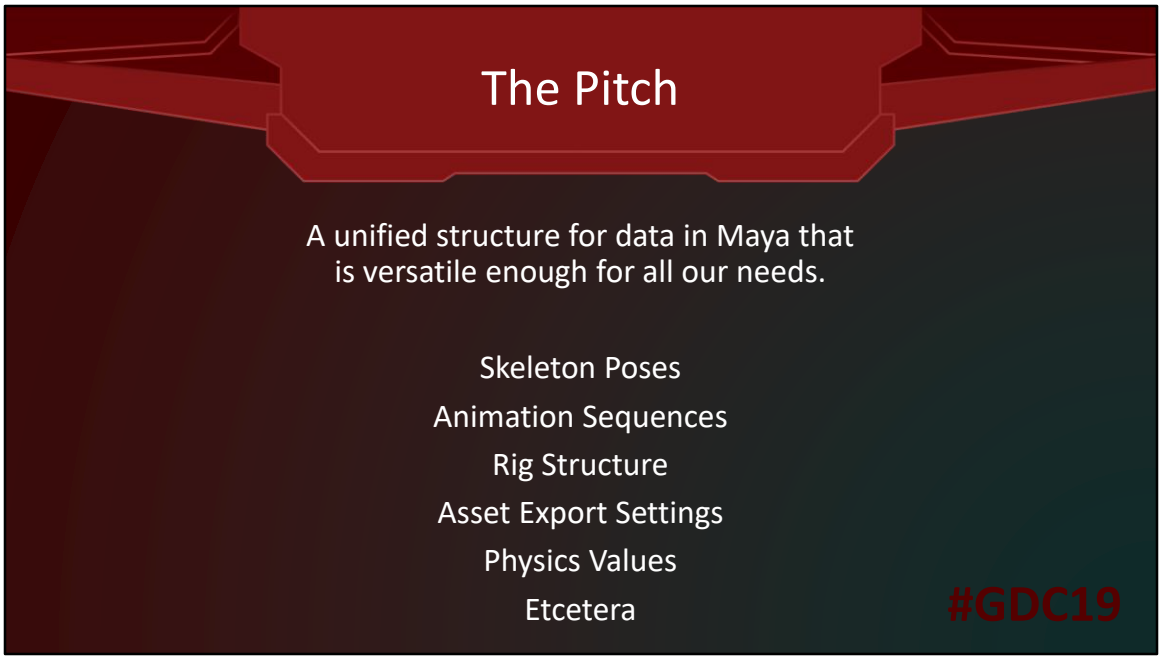
Namespaces offer convenient buckets but **overlap can be tricky**



First image that comes up when **searching Actor in Google**. Appropriate
Actor is our term for **asset management**
It needs to **track export assets** and **any supplemental data**
Open-ended enough to **append new asset types**.
Feature rich to support complex relationships like **LOD**s

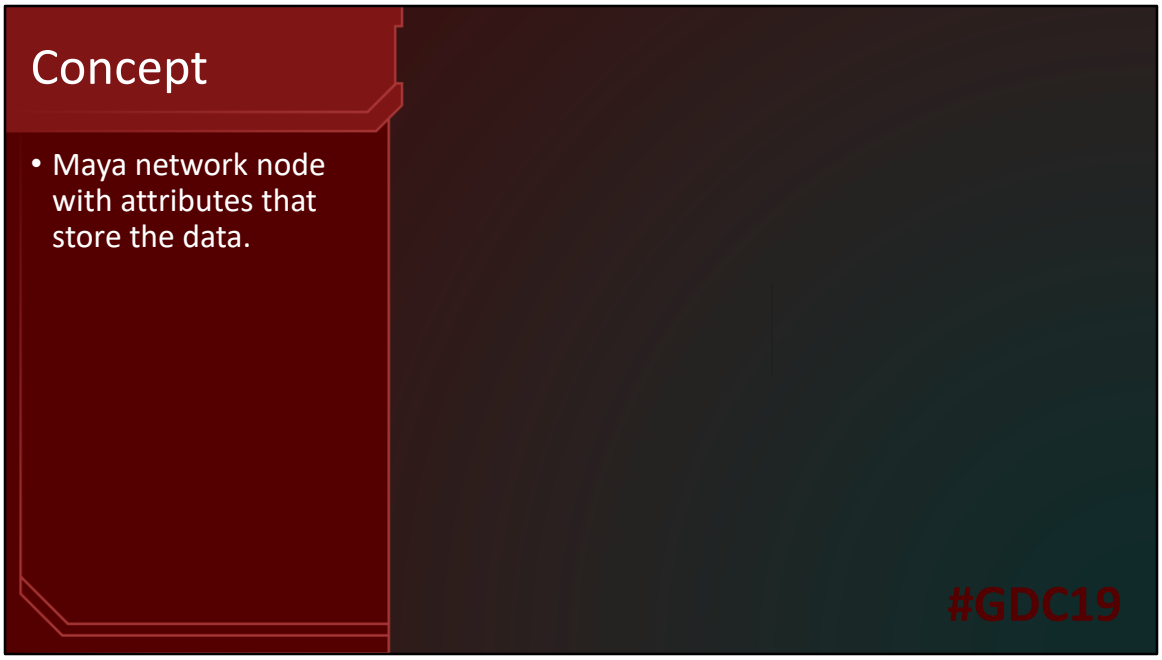
SEG; But, there is a better way.

How Metanode solves our problems.



Centralized system for data
Quick to setup for new uses
General enough for a **variety of applications**.

Examples

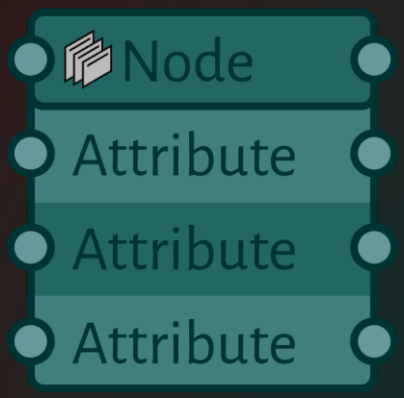


What do I mean when I say Metanode.
Foundational is a **Maya Network node** that will **remain persistent in the scene**.
Node is appended attributes to store data using the variety of Maya's **attribute types.**

SEG; Now for the Python.

Concept

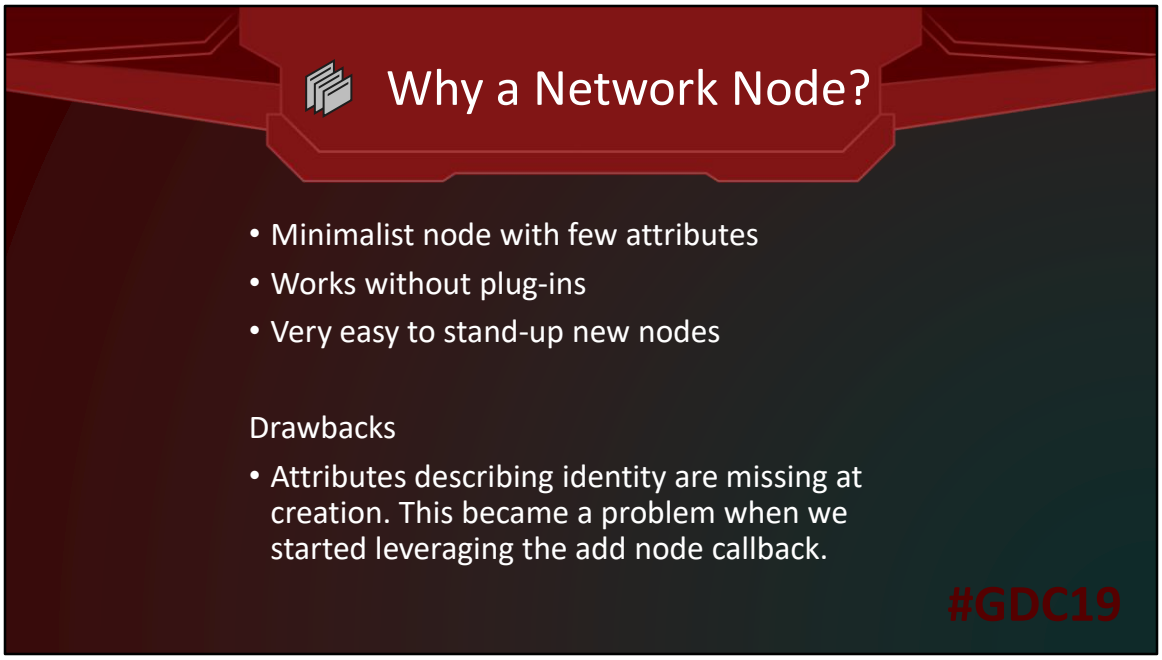
- Maya network node with attributes that store the data.
- Wrapped in Python class that inherits from the base Metanode class.
- All complex behaviors built into the class.



#GDC19

To enhance this node we **wrap it in a Python class**.
Inherits from a **base definition** that sets up **conventions** for access.
Creating a natural **interface for our data**.

The class also houses any **complex behaviors** we need for parsing data.
Give example of **saving and setting skeleton** pose data.
Those functions can be **centralized to the object** they operate on and not in a random library.

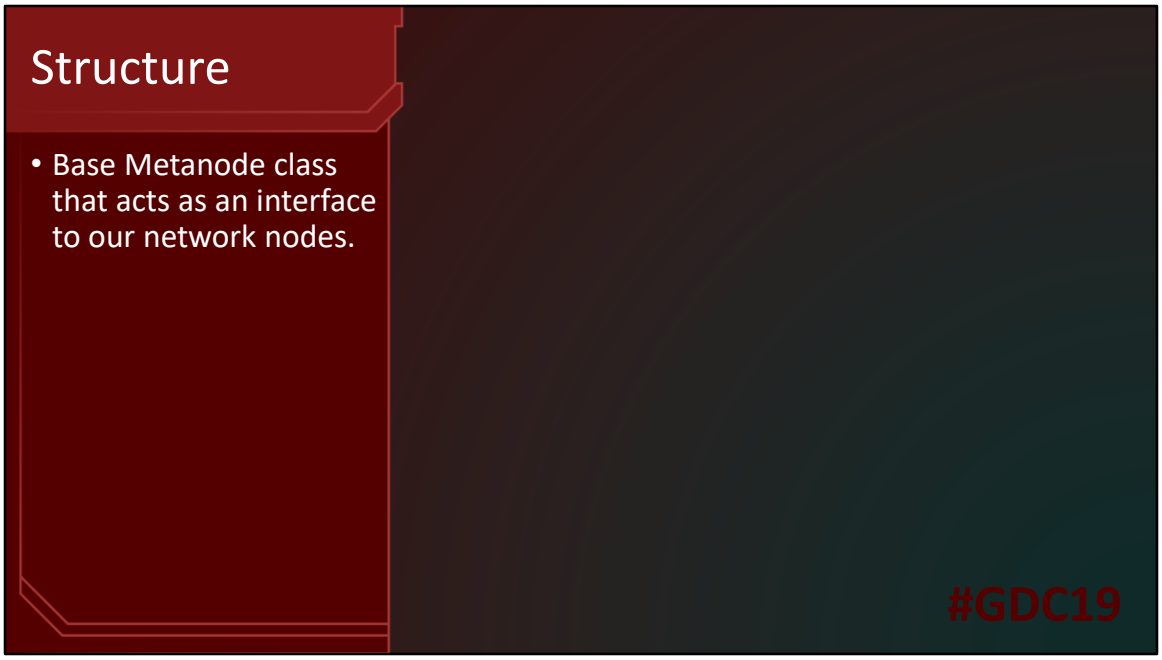


Why a network node?
Simple node without many attributes.
Network node is a **non-DAG** node so it stays **hidden** in the scene.

The scene can be **opened in any environment** without importing your plugins.

It is very **fast to create new classes** of Metanodes with no need for a new node.

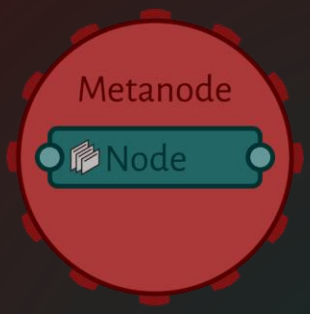
Drawbacks can be without or defining attributes at instantiation callbacks can get confused when we implement our signal system.



Next **three sections** we will cover the most important **Python objects** for the system. First up is the base **Metanode** class that **holds and edits our scene nodes**.

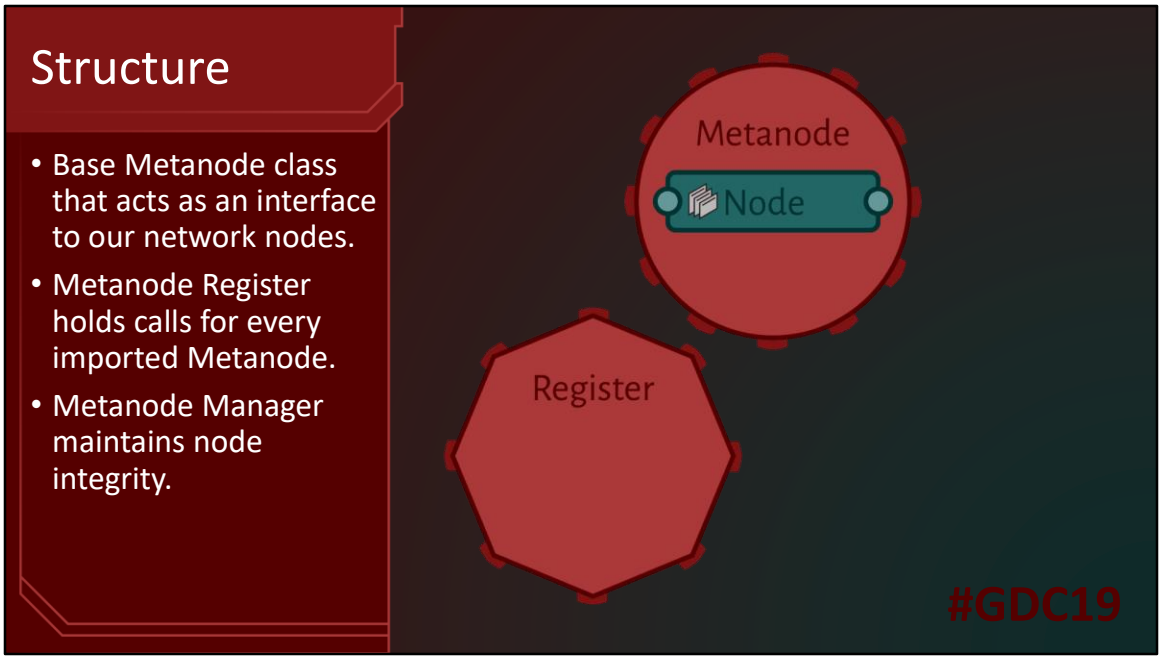
Structure

- Base Metanode class that acts as an interface to our network nodes.
- Metanode Register holds calls for every imported Metanode.



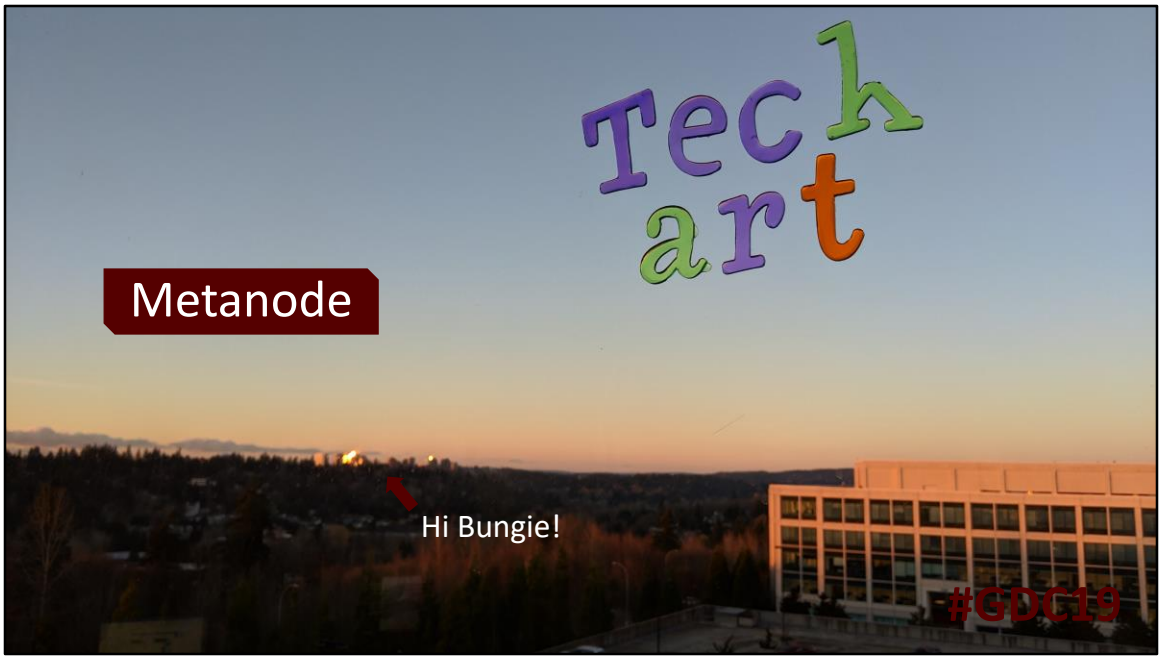
#GDC19

Next we'll talk about the **Register** and how we **track all the Metanode classes** the team creates.



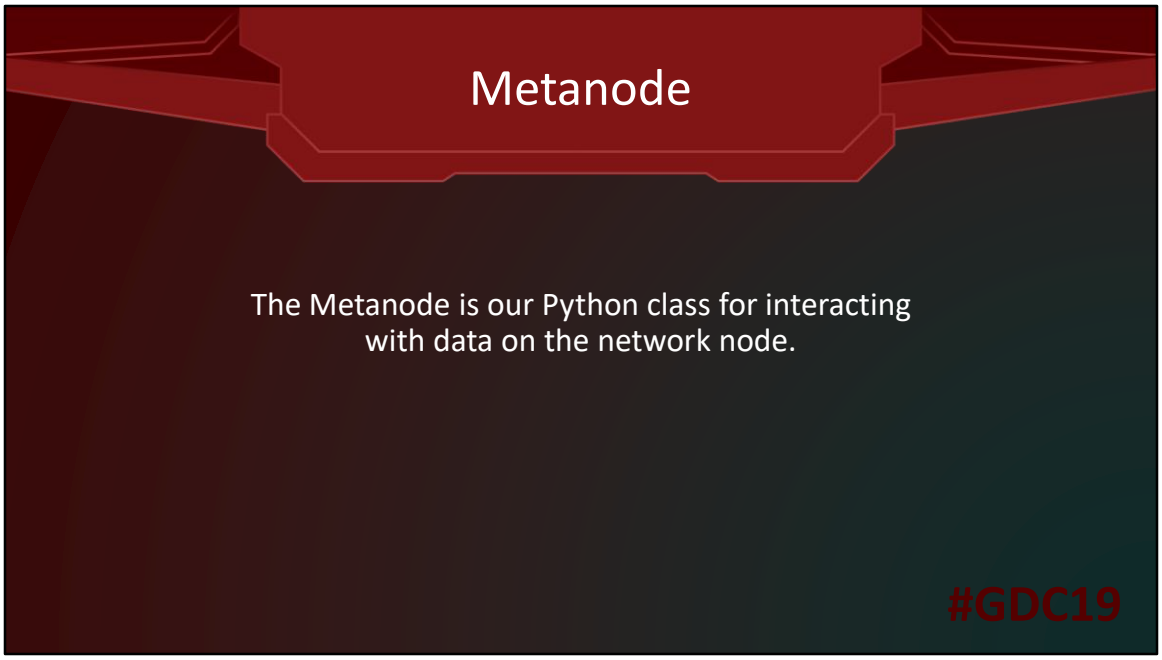
Finally we'll review the **Manager** and how we **keep nodes in our scenes valid**.

SEG; I'm excited, let's get started.



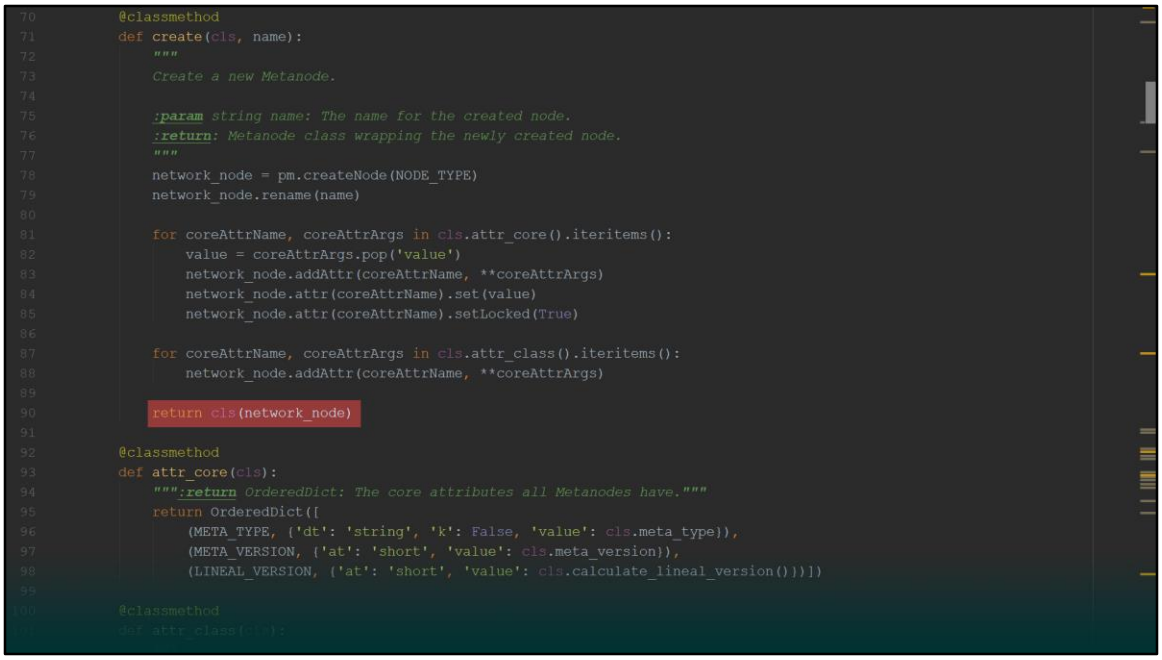
The dawn of the Metanode
This is my view at work.
I go through many **complex and arcane** steps brewing my perfect cup of coffee in the morning and then wave at Bungie.

SEG; Settle in for an exciting day of data management.



As mentioned, Metanode is our **Python class for wrapping the Maya network node**. They work in **collaboration**, the network node **storing data** and the Metanode **defines interface to data**.
Example; Actor node

SEG; And we make one with the **create()** call



```python
    @classmethod
    def create(cls, name):
        """
        Create a new Metanode.

        :param string name: The name for the created node.
        :return: Metanode class wrapping the newly created node.
        """
        network_node = pm.createNode(NODE_TYPE)
        network_node.rename(name)

        for coreAttrName, coreAttrArgs in cls.attr_core().iteritems():
            value = coreAttrArgs.pop('value')
            network_node.addAttr(coreAttrName, **coreAttrArgs)
            network_node.attr(coreAttrName).set(value)
            network_node.attr(coreAttrName).setLocked(True)

        for coreAttrName, coreAttrArgs in cls.attr_class().iteritems():
            network_node.addAttr(coreAttrName, **coreAttrArgs)

        return cls(network_node)

    @classmethod
    def attr_core(cls):
        """:return OrderedDict: The core attributes all Metanodes have."""
        return OrderedDict([
            (META_TYPE, {'dt': 'string', 'k': False, 'value': cls.meta_type}),
            (META_VERSION, {'at': 'short', 'value': cls.meta_version}),
            (LINEAL_VERSION, {'at': 'short', 'value': cls.calculate_lineal_version()})])

    @classmethod
    def attr_class(cls):
```

It’s a **classmethod** that **generates the metanode,** pass in name

Here we **create** the **network node**

adds the appropriate **attributes** from class dictionaries

returns it wrapped in the **class**.

SEG; Node attributes defined in the class, come in **3 major types**.

Core Attributes

META_TYPE

- Fully qualified path to Metanode class that created this node

META_VERSION

- Version of the Metanode when node was created or updated

LINEAL_VERSION

- Class lineage versions as sum

```
@classmethod
def attr_core(cls):
    """:return OrderedDict: The core attributes all Metanodes have."""
    return OrderedDict([
        (META_TYPE, {'dt': 'string', 'k': False, 'value': cls.meta_type}),
        (META_VERSION, {'at': 'short', 'value': cls.meta_version}),
        (LINEAL_VERSION, {'at': 'short', 'value': cls.calculate_lineal_version()})])
```

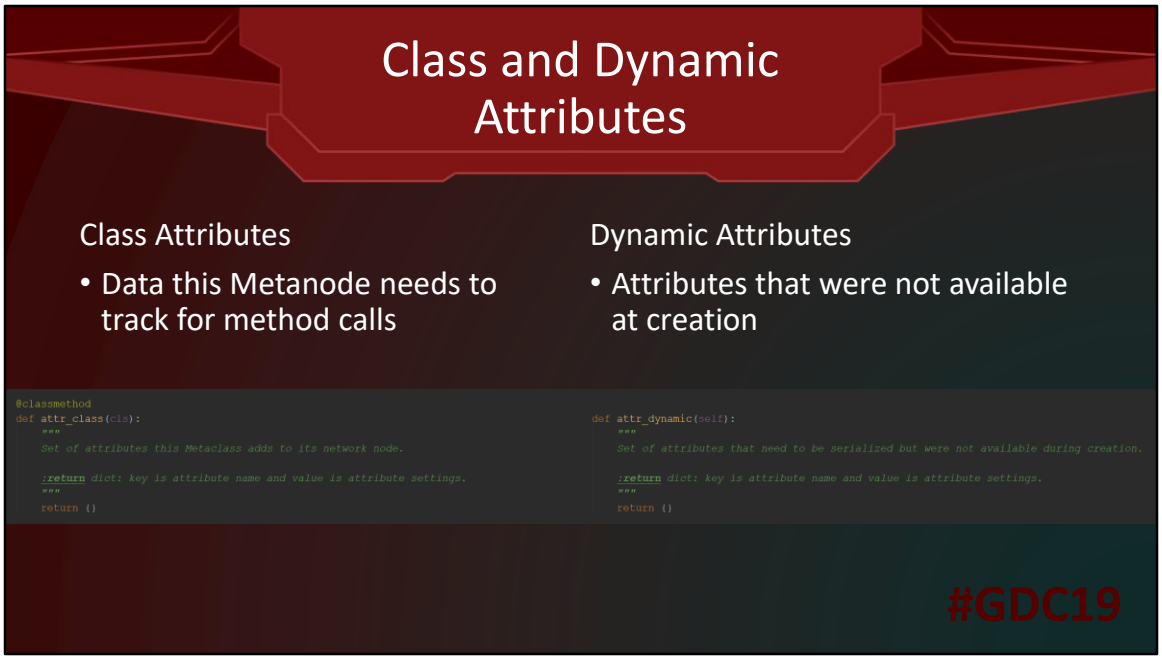
#GDC19

Core attributes are the only attrs **defined on the base class**.
They are the **identifying information** for the Metanode class that created them.
Call **returns a dictionary**, **keys** are names and **values** are attribute **arguments** for the attribute type.
Types of attributes as examples.
They are created and **values are set and locked**.

The **meta_type** is the **path to the python class** used to wrap this node.

Meta version is the **last iteration of the class** this node has been updated to.
Some changes **don't require a version change** but updating one of the attribute lists usually does.

Lineal version is the **sum of the Method Resolution Order**.
If there is a change at any point in the **class inheritance** this integer advances.



Now we **know what the node is** but we need **something useful** on it.
There are **two ways we store data** on Metanodes.
Class level and the other is **object based**.

Class attributes are every attribute our **methods will expect** on the node.

Dynamic attributes not created with the node.
The **class controls how we check the dynamic attributes** but they can **change from one instance to another**.
Avoid declaring similar Metanode classes with only **minimal attribute differences**.
Example; **Template node** that takes a passed list of attributes for that instance to track.

SEG; We have a Metanode with attributes we need a way to **query** and **edit** them.

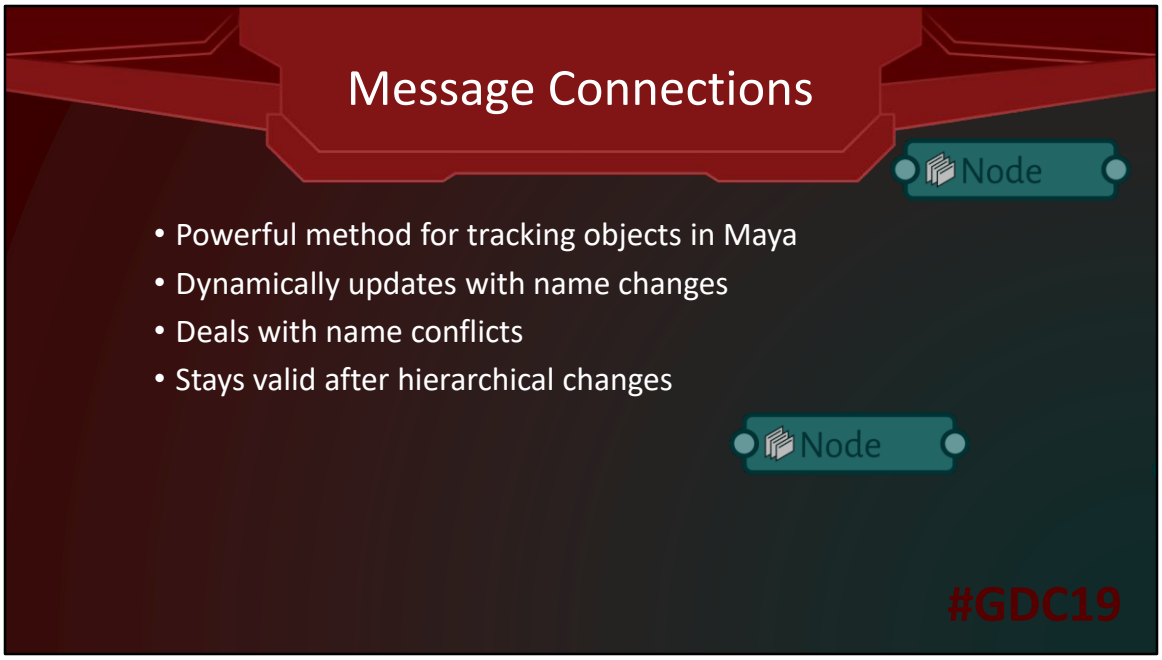
Get and Set

- We need a way to get and set attribute values
- Simple interfaces that only requires an attribute name
- Complex attribute interactions may need custom calls

Attributes can be a **variety** of **types** but as **Python people** we don't want to worry about that.

Attribute name gives us type information from **attribute dictionaries**.
We know the type, whether it's multi.
One **centralized** place to **validate data** passed in and **standardize the output**.

More **complex getters and setters might be required** for some Metanodes but these base calls will work in most cases.
Example; Two way message connection.

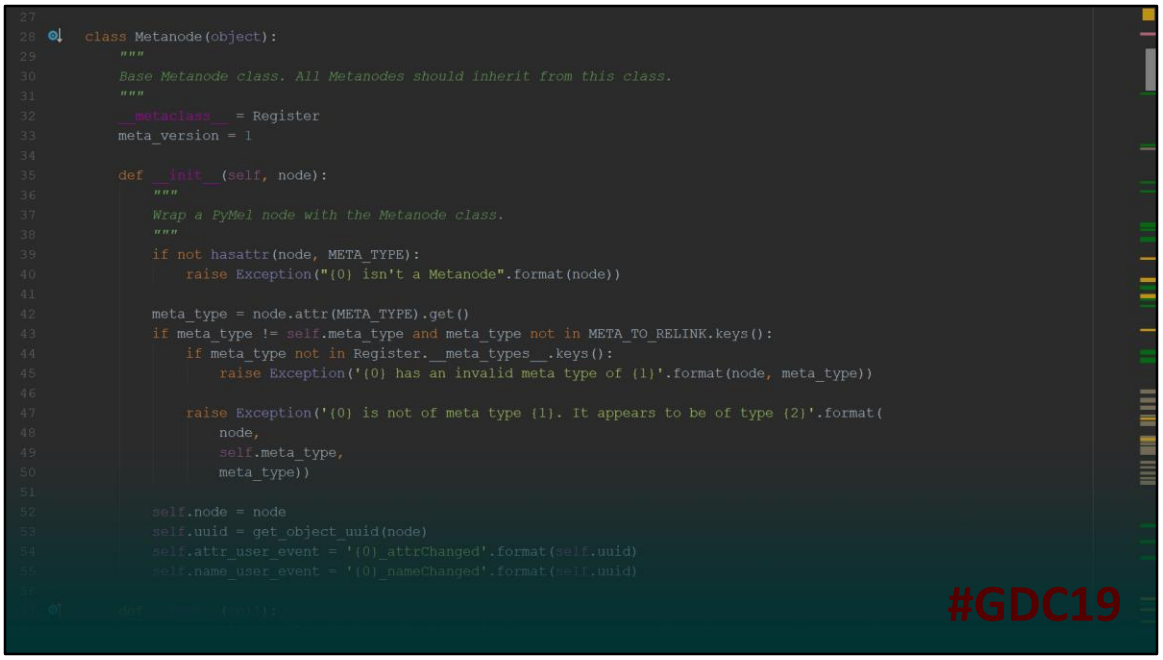


Powerful data tracking with **Message connections**

Updates with **name changes**

Works with **name conflicts**

Valid even after **DAG hierarchical** changes



Create() returned a class but in the future we will wrap the class with the **initialization**.
Here in the init we verify the **node and class match**
Register the node and it's UUID as **properties** of the class

Fast Setup

- New Node creation is simple
- Add attributes to class attributes and the Metanode is ready
- Any complex methods can be appended to the class

```
import meta.core


class Skeleton(meta.core.Metanode):
    '''
    A Metanode for saving skeletons.
    '''
    metaversion = 1

    @classmethod
    def attr_class(cls):
        return {'skeletonPath' : {'dt':'string'},
                'bindPose' : {'dt':'string'},
                'zeroPose' : {'dt':'string'},
                'root': {'at':'message'},
                'noBindJoints': {'at':'message', 'multi':True},
                'noExportJoints': {'at':'message', 'multi':True}}

```

#GDC19

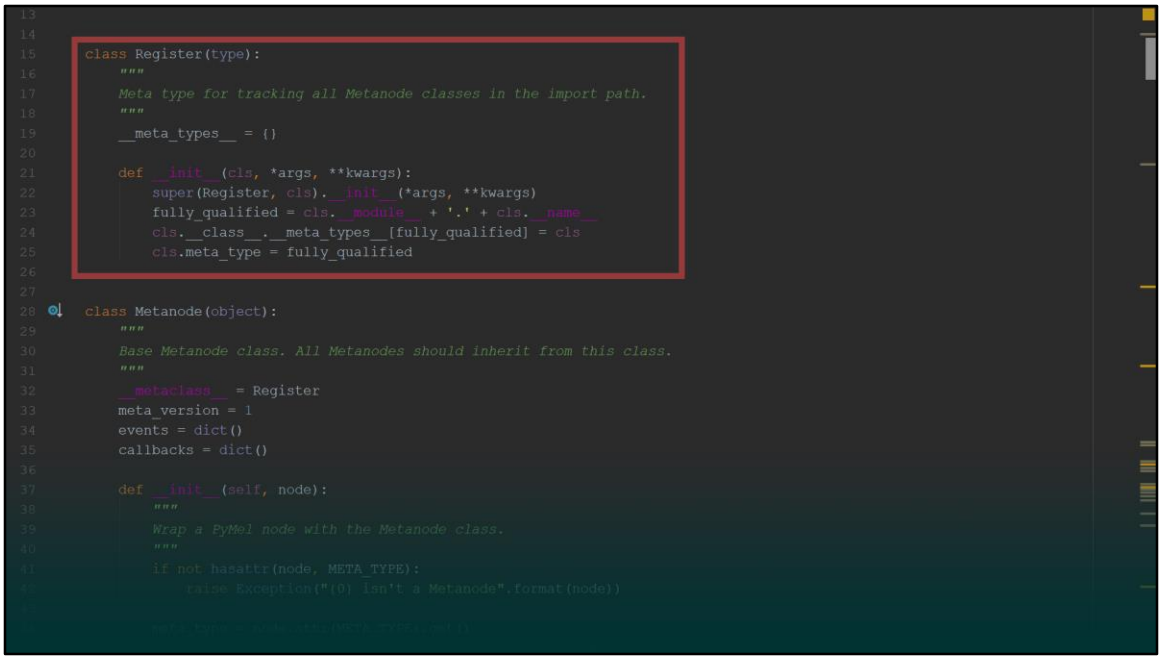
Easy to add a new node

Update class attribute dictionary and the node is ready

Later we can add methods for saving bind pose.
Example; Tim's face poses.

SEG; We're ready to **define** a bunch of **subclasses** but how are we going to **keep track** of all of them?

Custom meta class for **tracking** all Metanode classes in the **imported code base**.
There are **mixed opinions** online about creating your own meta classes, but we made one and it has been **very useful**.



```python
class Register(type):
    """
    Meta type for tracking all Metanode classes in the import path.
    """
    __meta_types__ = {}

    def __init__(cls, *args, **kwargs):
        super(Register, cls).__init__(*args, **kwargs)
        fully_qualified = cls.__module__ + '.' + cls.__name__
        cls.__class__.__meta_types__[fully_qualified] = cls
        cls.meta_type = fully_qualified


class Metanode(object):
    """
    Base Metanode class. All Metanodes should inherit from this class.
    """
    __metaclass__ = Register
    meta_version = 1
    events = dict()
    callbacks = dict()

    def __init__(self, node):
        """
        Wrap a PyMel node with the Metanode class.
        """
        if not hasattr(node, META_TYPE):
            raise Exception("{0} isn't a Metanode".format(node))
```

__metaclass__ is set as the Register.

All **subclasses** will **inherit** this **metaclass**.
No need to manually add new subclasses.

Custom Metaclass

```
class Register(type):
    """
    Meta type for tracking all Metanode classes in the import path.
    """
    __meta_types__ = {}

    def __init__(cls, *args, **kwargs):
        super(Register, cls).__init__(*args, **kwargs)
        fully_qualified = cls.__module__ + '.' + cls.__name__
        cls.__class__.__meta_types__[fully_qualified] = cls
        cls.meta_type = fully_qualified
```

- Register builds a the meta_type and adds to the Register's __meta_types__.
- Any Metanode subclass will be tracked and we can check this meta_type to verify a node is valid.

#GDC19

Register builds a **fully qualified path** to our class and adds it as a **key in meta_types**
The **value** is set as the **class object**
Then the new class's **meta_type property** is set to the path.

Any **subclass** of Metanode **imported** is added to the Register's **__meta_types__ automatically**.
Once collected it's easier to **validate and find** the right Python class given a **meta_type**.

Get Meta

With a Register we can easily test the Metatype of our node and get the correct class.

```
def get_metanode(node, *args, **kwargs):
    """
    By passing a network node with a meta type attribute, a Metanode instance will
    be returned of the appropriate meta type.

    :param pm.PyNode() node: a Maya network node with a meta type attribute
    :return: A subclass of Metanode of the type set on the network node.
    """
    if isinstance(node, basestring):
        node = pm.PyNode(node)
    if not pm.hasAttr(node, META_TYPE):
        raise Exception("{0} isn't a Metanode".format(node))
    meta_type = node.attr(META_TYPE).get()
    if meta_type not in Register.__meta_types__.keys():
        raise Exception("{0} has an invalid meta type of {1}".format(node, meta_type))

    return Register.__meta_types__[meta_type](node, *args, **kwargs)
```

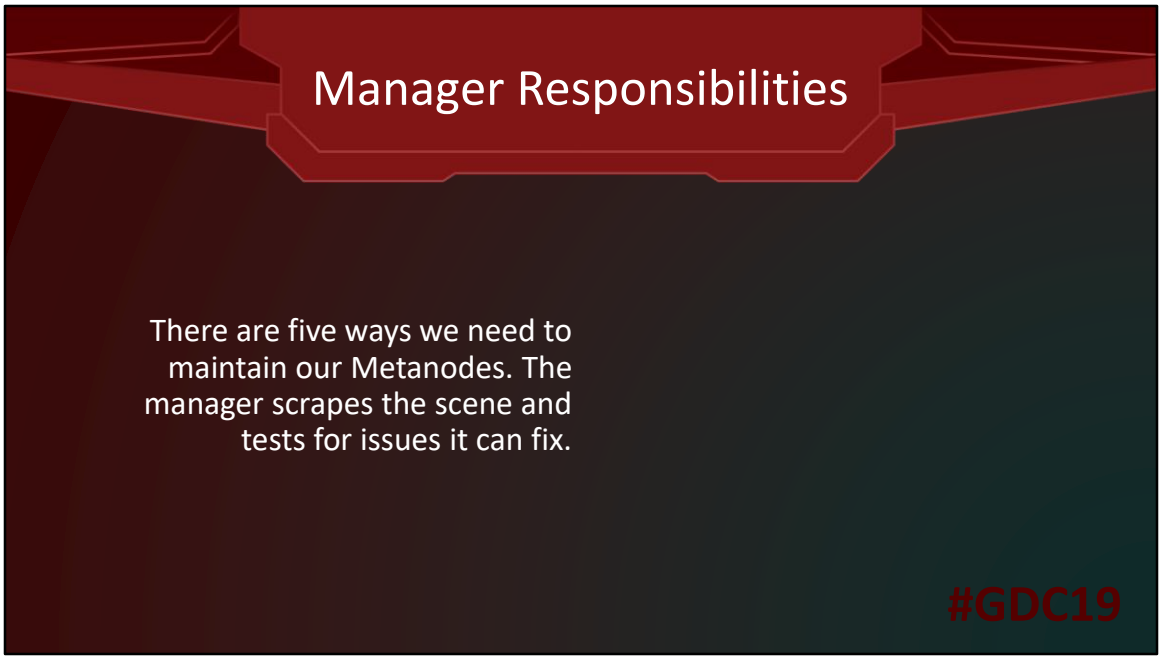
#GDC19

With the Register we are able to build things like **get_metanode**.
This function **automatically finds class** to wrap our nodes in.

SEG; Register only responsible for tracking current Metanodes but we have **another class for maintaining** them.

The Manager
You give me **too many slides** to fill and I'll start adding **pictures of my kids.**
He looks very serious about you updating your **JIRA**

SEG; So what does the the manager do?



Responsible for **maintaining** our network nodes.
Keeps **events** for **creating and destroying** network nodes.
Class attribute **track any network nodes** it find in the scene.
On scene open checks nodes for **five possible issues**.

Update

- Most updates are only required when the node attributes change.
- Manager only enforces updates on a set of Metanodes referenced in the config.py.
- There is a generic update that tries to migrate data to new nodes.

Update, **most important**,
checks **version and lineal**.
Update function **copies attributes**, **renames** node Metanode, **instantiates new** Metanode and tries to **apply old data** to new attributes.
Not all are **automatically updated**.
Example; Rig nodes.
If **changes are too severe** we will write a specific update call for that version.
Example Tag Nodes.

Relink

- Re-organizing Metanode modules requires fixing the meta type attribute.
- Relink uses a dictionary of old paths with their new location as the value.

Periodically we need to **reorganize our code base**.
If an old network node's **meta_type doesn't match** an imported python class it **wont be valid**.
Fully qualified paths mean we have to relink a node if the **package, module or class change**.
Manager updates meta_type from **config dictionary**.

Orphaned

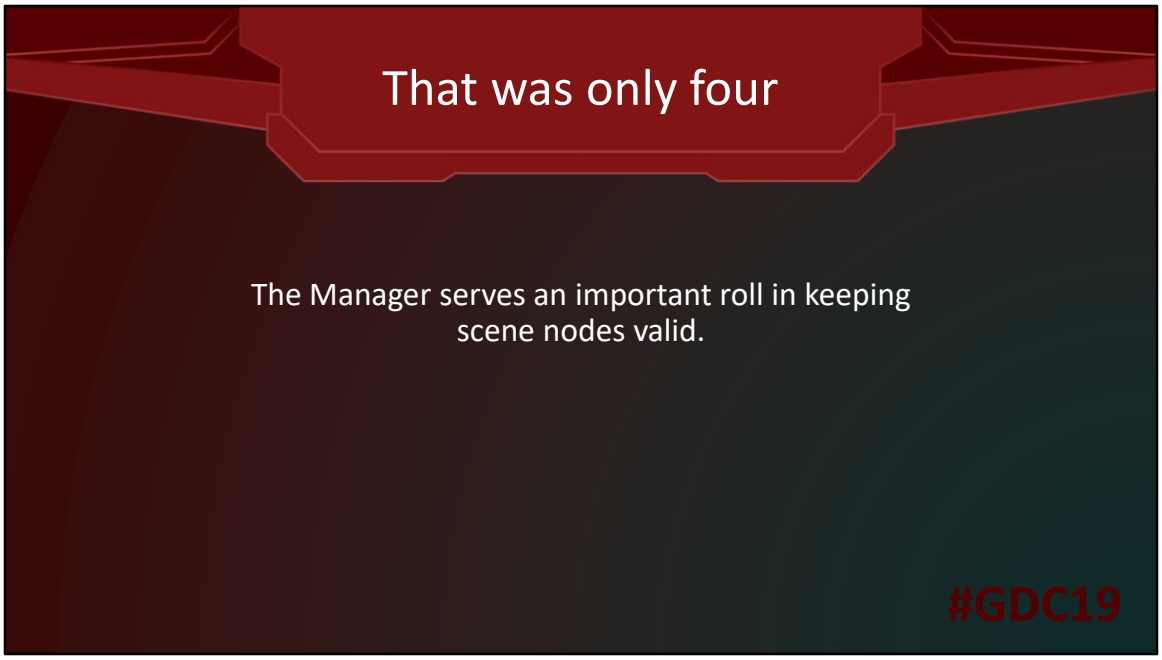
- Sometimes nodes have their assets deleted from under them.
- Subclasses overwrite is_orphaned call.

Base method on Metanode class is_orphaned() **returns False**.
Subclasses can overwrite this call to check the node is still being used.
Example; skeleton meta with no root
Manager queries all Metanodes for orphaned state and **delete those returned True**.

Deprecated

- When it's time to retire a Metanode we can put it's jersey up in the rafters of our config file.
- Nodes with deprecated Metatype will be deleted.

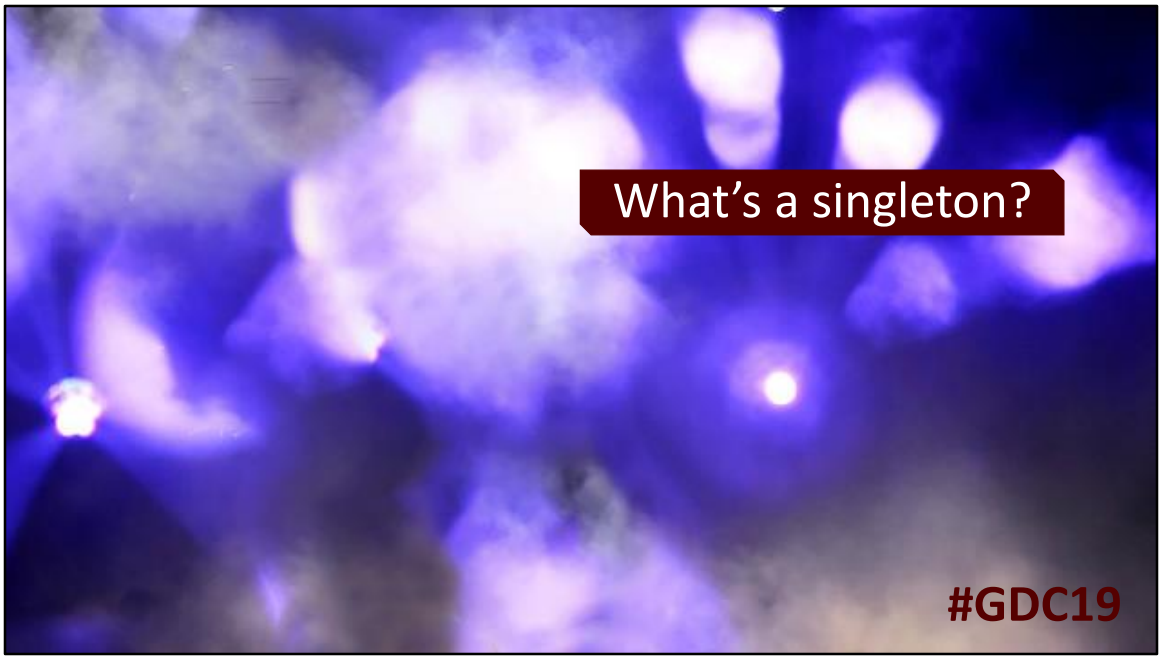
Sometimes a Metanode is **no longer useful**.
If the Metanode **no longer brings you joy**, say **thank you** and deprecate it.
Delete the class and add it to **deprecated list**.
All nodes on deprecated list will be **deleted**.



I'm getting to **the fifth one**, it gets its own chapter
Important take away there are a **lot of ways to break your data**.
Important that we **systematize maintenance**

SEG; Now for that **fifth one** I promised

That's the **other kid**.
He's quite the singleton.

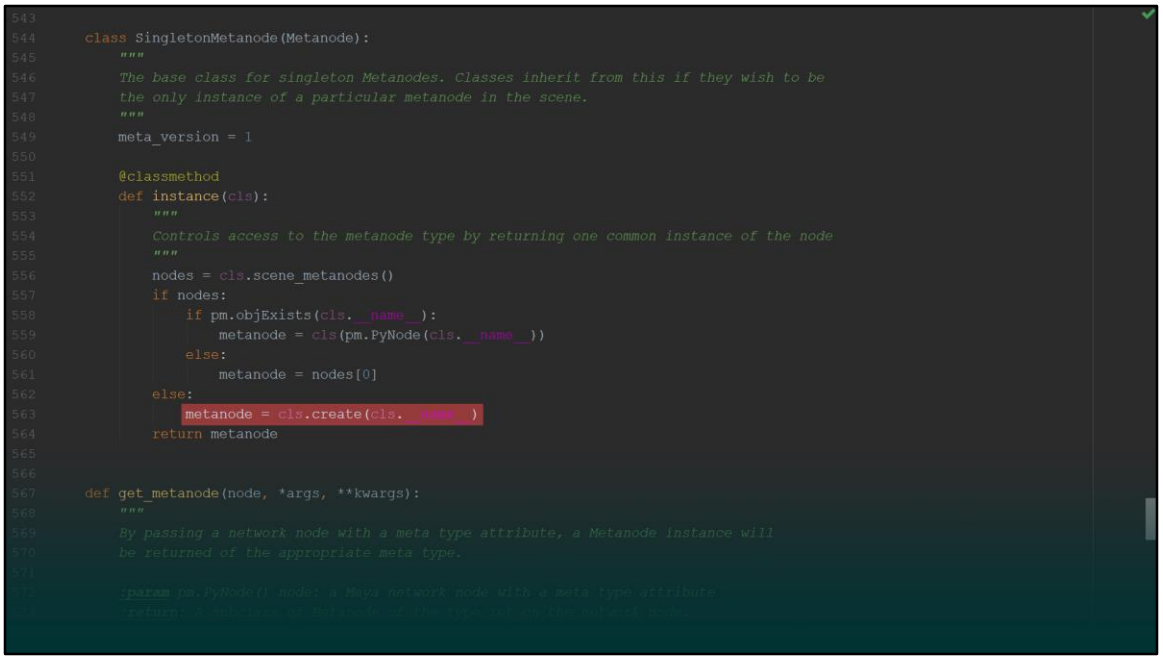


What is this structural concept of a Singleton?
It's a **construct without peers**.
An **entity** who's **completely unique without duplication**.

SEG; In other words,

It’s Beyoncé.
You can’t **instantiate another Queen Bey**, she’s not a mall Santa.
Sasha Fierce might not fit this metaphor

SEG; This structure gives us a **singular authority** for the scene



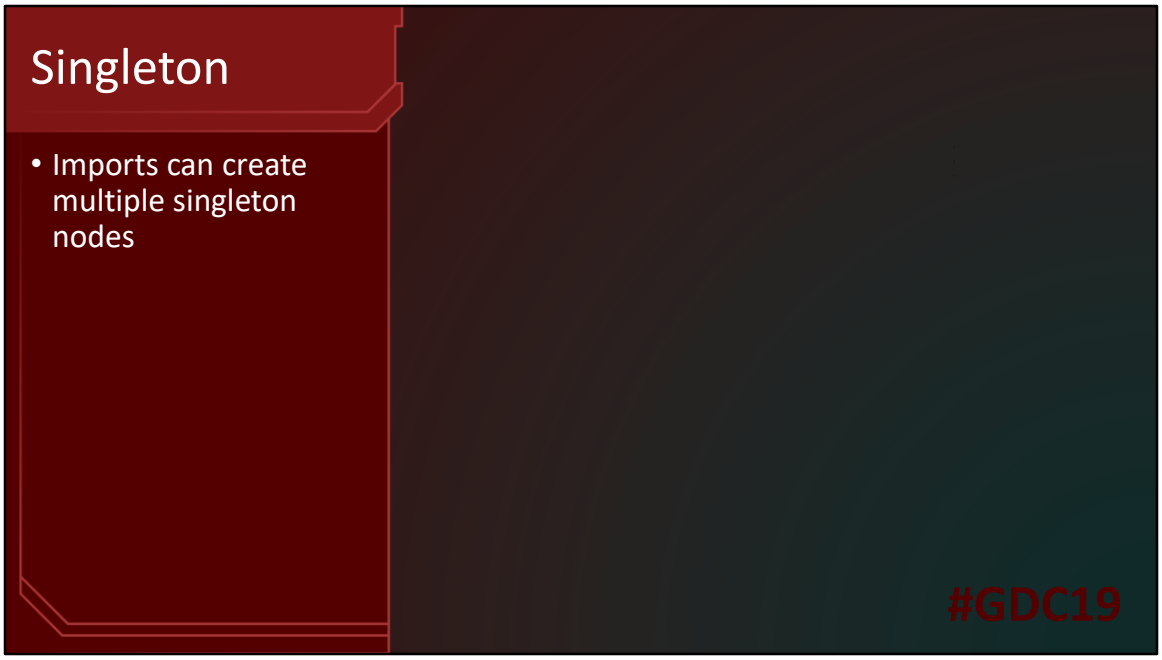
```python
class SingletonMetanode(Metanode):
    """
    The base class for singleton Metanodes. Classes inherit from this if they wish to be
    the only instance of a particular metanode in the scene.
    """
    meta_version = 1

    @classmethod
    def instance(cls):
        """
        Controls access to the metanode type by returning one common instance of the node
        """
        nodes = cls.scene_metanodes()
        if nodes:
            if pm.objExists(cls.__name__):
                metanode = cls(pm.PyNode(cls.__name__))
            else:
                metanode = nodes[0]
        else:
            metanode = cls.create(cls.__name__)
        return metanode


def get_metanode(node, *args, **kwargs):
    """
    By passing a network node with a meta type attribute, a Metanode instance will
    be returned of the appropriate meta type.

    :param pm.PyNode() node: a Maya network node with a meta type attribute
    :return: a subclass of Metanode of the type set on the network node
```

Here is our **Singleton class** which is a **subclass of Metanode**

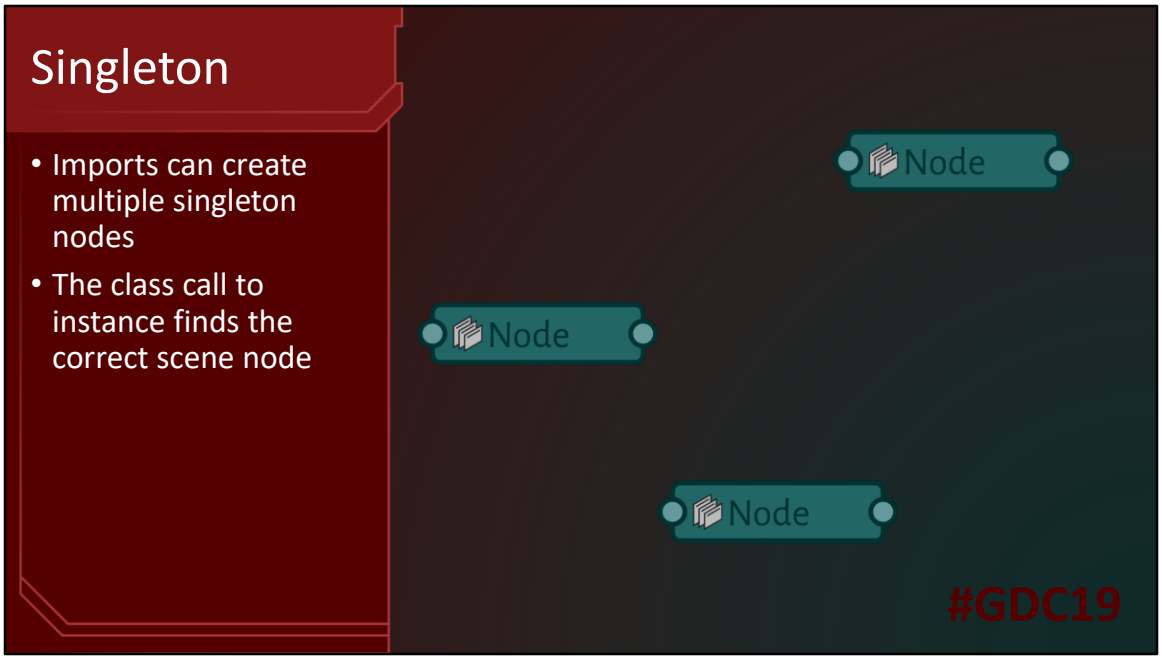
The important call is **instance()** which defines how we get the correct network node.

We check the scene for a node with the **name of the class**.

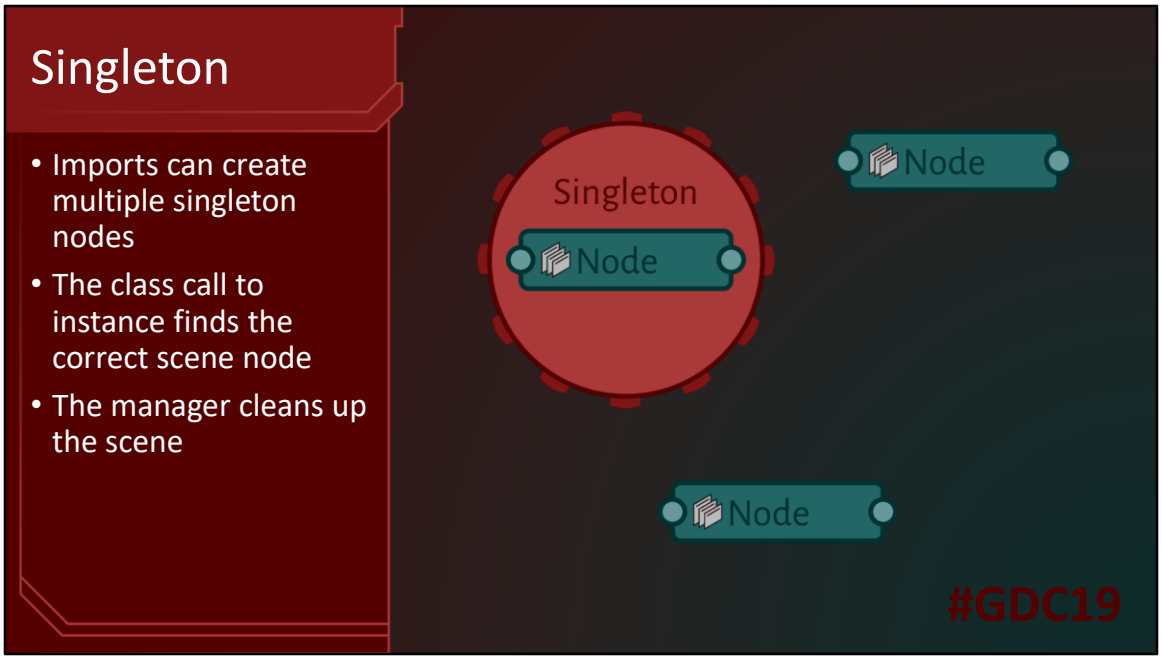
If **none exist** a new singleton node is **created**
Class **isn't a singleton**, the node is a singleton.



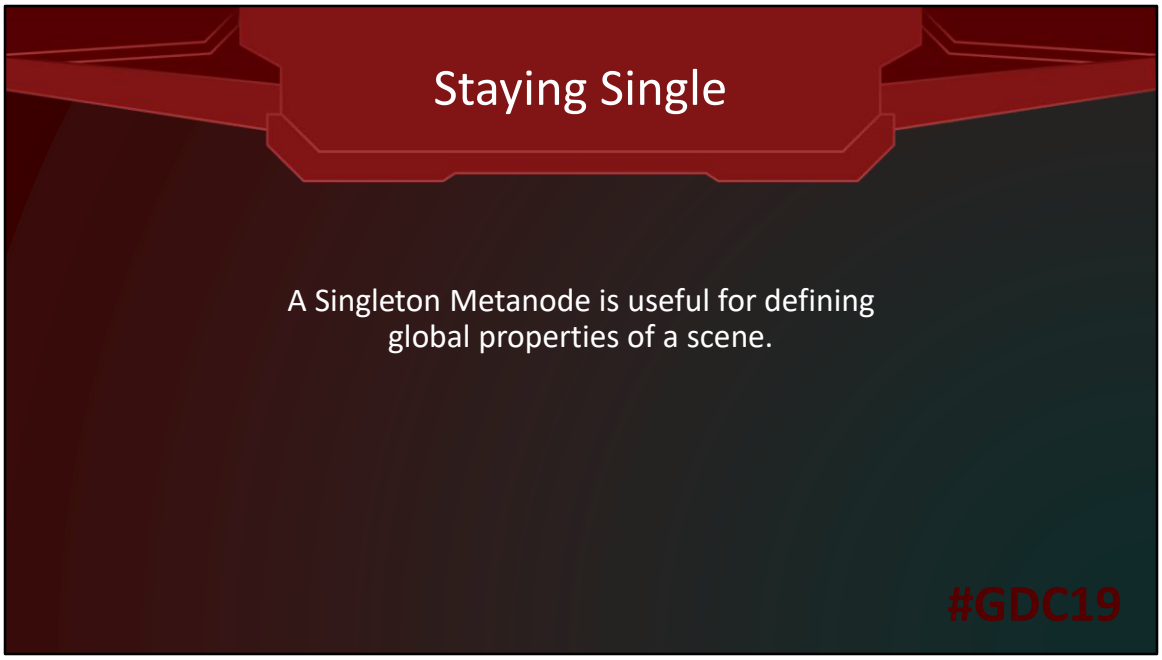
There could be **multiple Singleton** nodes in the scene
Example; Artist scale check



Here is where our **instance call** finds our correct node.
Wraps it in our class.
This is why it's important to have a **centralized** way of finding our Singleton.



Here is the fifth way our **Manager fixes** up the scene.



Singleton works as **global scene data**
Example; Active Actor, no disagreement

SEG; Those are our objects for data but now we need to **tap into that well**.

We have a lot of data in our Metanode system, but how do we **expose that to our GUI's**

SEG; My first book when learning Python was **Maya Python for Games and Film**, but my second book

Was **Practical Maya Programming with Python**
By **Robert Galanaksi**, founder of Tech Artists dot Org
Chapter 5: **Building GUIs for Maya**
Separate your Maya scene code from the **interface code**.
I ignore this at **my own peril**.

SEG; So to support independent data models in our UI we **build it into the Metanode system**

User Events

- Two API callbacks and events created during create_event.
- Other classes can subscribe directly to a Metanode for name and attribute changes.
- Callbacks can be removed with the unsubscribe call from any Metanode.

```
@classmethod
def unsubscribe(cls, callback_data):
    '''
    Remove callback for subscribed event.

    :param tuple callback_data: UUID and Index identifier for callback to remove.
    '''
    uuid, index = callback_data
    if uuid in cls.callbacks:
        cls.callbacks[uuid].remove(index)
        om2.MMessage.removeCallback(index)
```

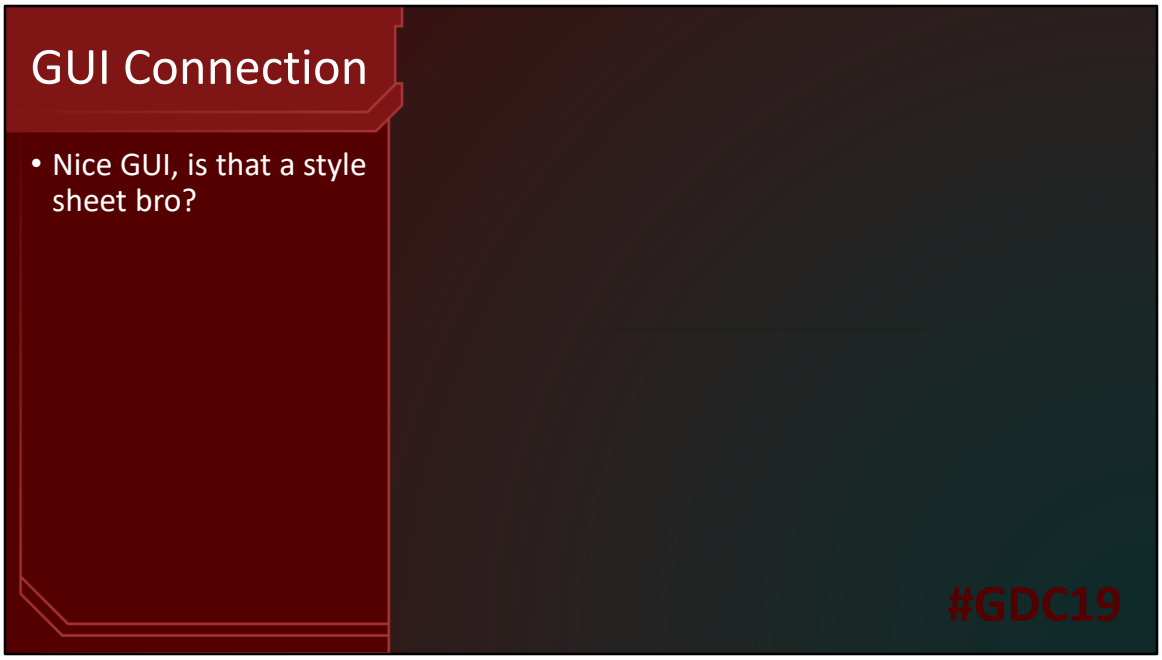
#GDC19

When **Manager** gets an **API event** for a newly created network node it **calls the create_event**
Create_event **generates user events** for node names and attributes

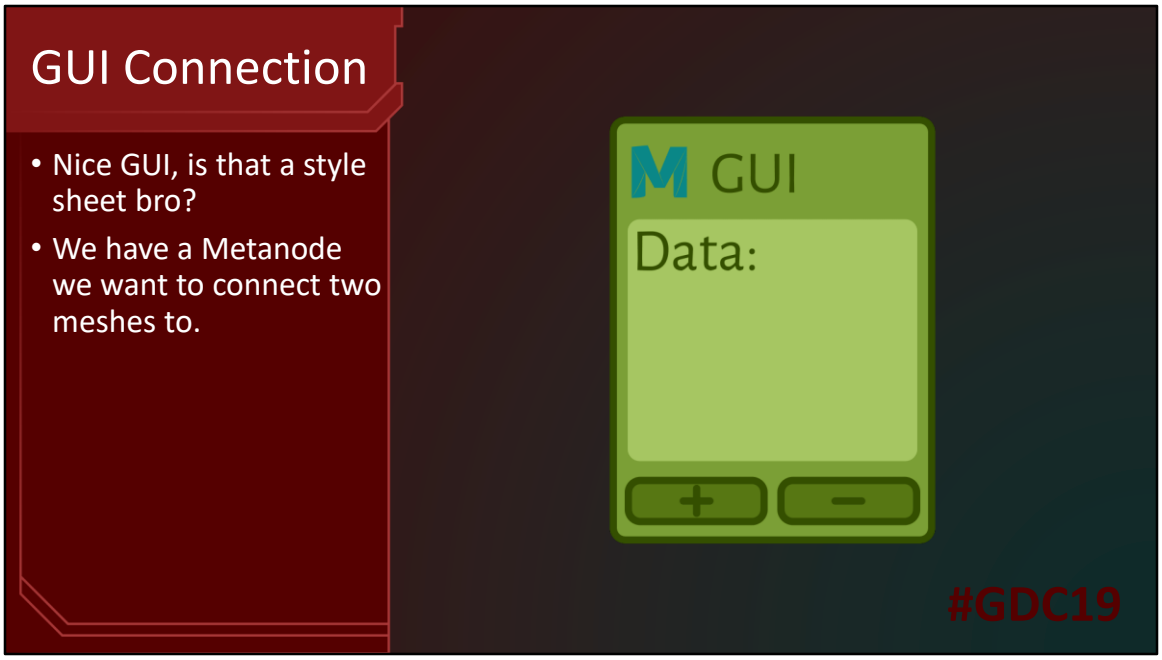
Here are our currently **supported subscriptions** on the base Metanode class.
Any node **attribute or name change** will push out to these events.

We have **one unsubscribe** that uses the **UUID and index tuple**.
Node **UUID** if tracking multiple Metanodes.
Helpful that Metanode **tracks attached callbacks** and automatically **cleans them up on destruction**.

SEG; Now let's apply this to an interface



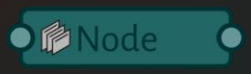
We have our **GUI**.
It **looks amazing**, even docks correctly.



We have an **Actor network node** we want to **connect two export mesh assets** to.

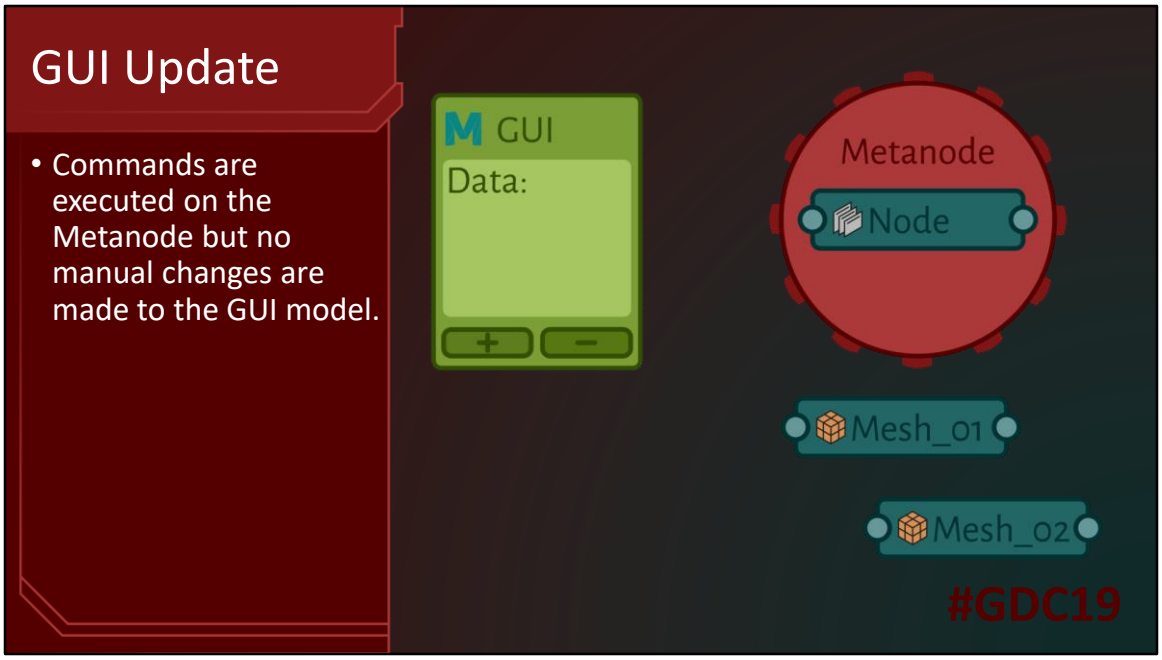
GUI Connection

- Nice GUI, is that a style sheet bro?
- We have a Metanode we want to connect two meshes to.
- The GUI finds the appropriate Metanode and instantiates its class.

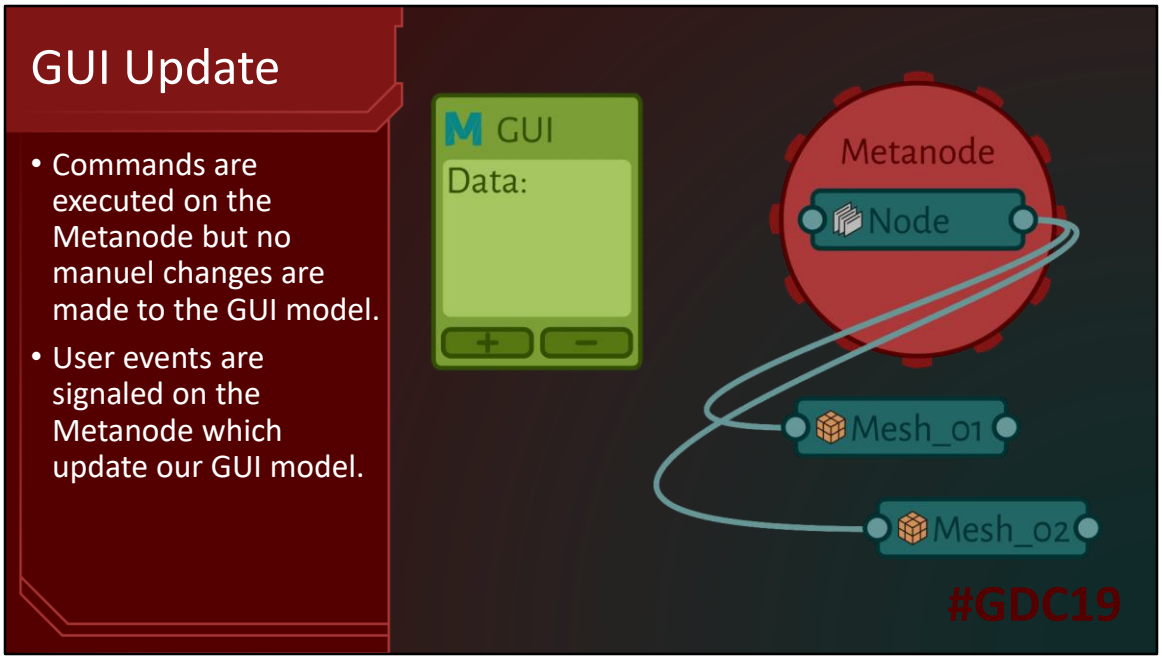


#GDC19

The GUI finds node and **wraps it in the Metanode**
GUI **subscribes to attribute change events** from the Metanode class.

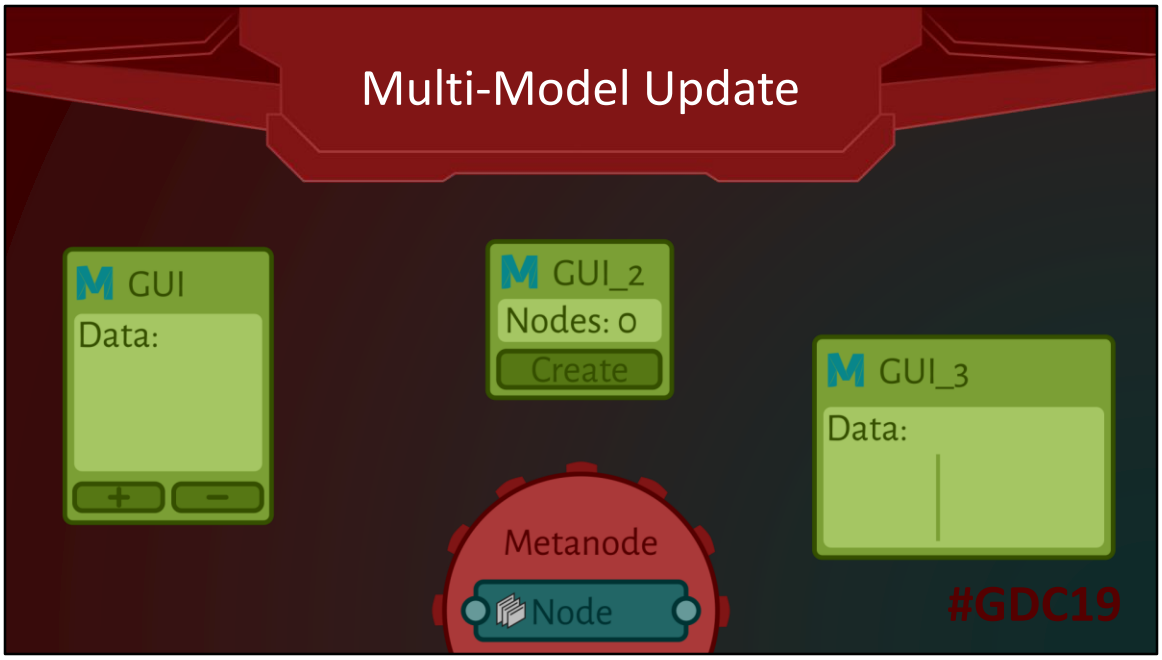


We **pass commands**
Metanode **connects to meshes**.
We should **not assume the data is set**.

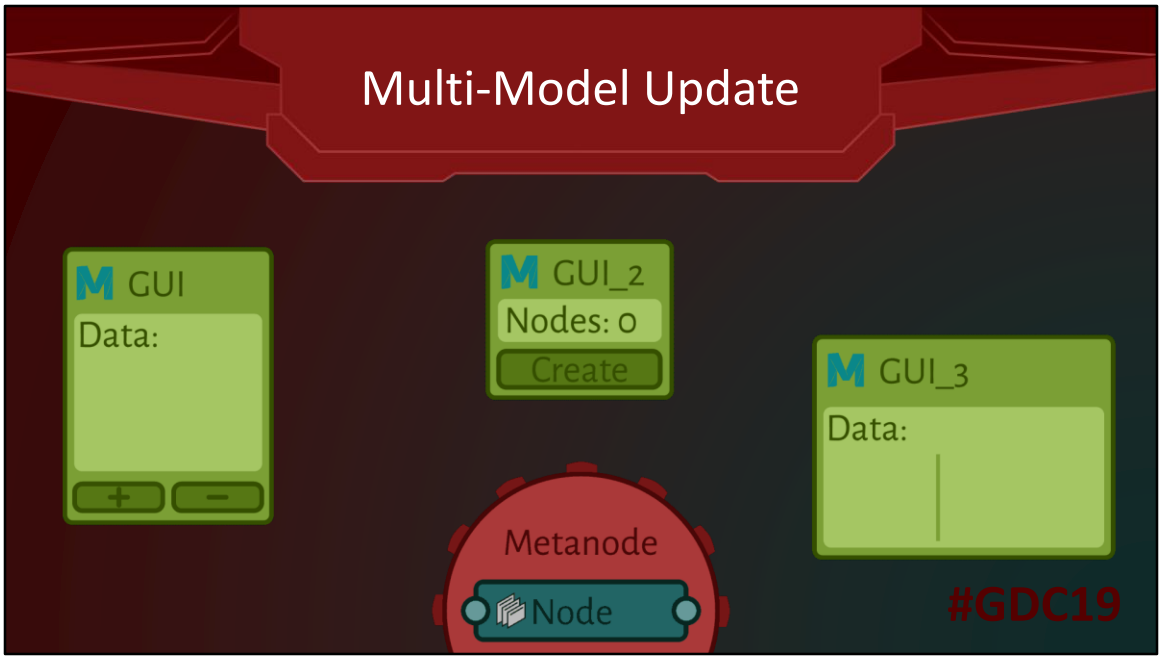


Signals emitted
Subscribed **model is updated**.

SEG; This might seem **overly elaborate**, but it has **modularity benefits**.



Structure works well when supporting **many GUIs** watching the **same data**.
If our GUIs **updated their own data** models they would also have to update any **future UIs, gets messy**.

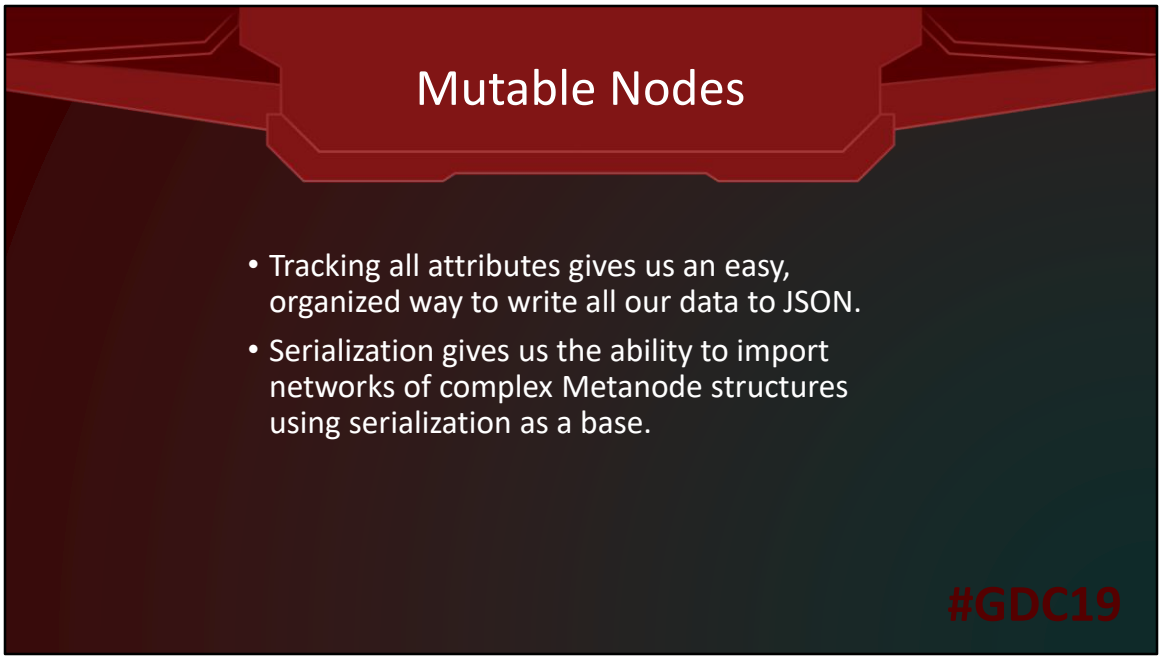


Now with all **GUIs subscribed** to our Metanode they are all **updated no matter how the data was edited**.

We want to make **good GUI architecture easy to implement**.
By **standardizing Callback management** on the Metanode class we encourage **modular UI**

SEG; Now for the **final chapter**. You can taste that post-GDC beer already

Cereal silos.
New pipeline **avoids** Maya **referencing**.
All Maya data can be **dumped** and **instantiated** in new scenes.



Class **defined attributes** makes it **easy to get data**.

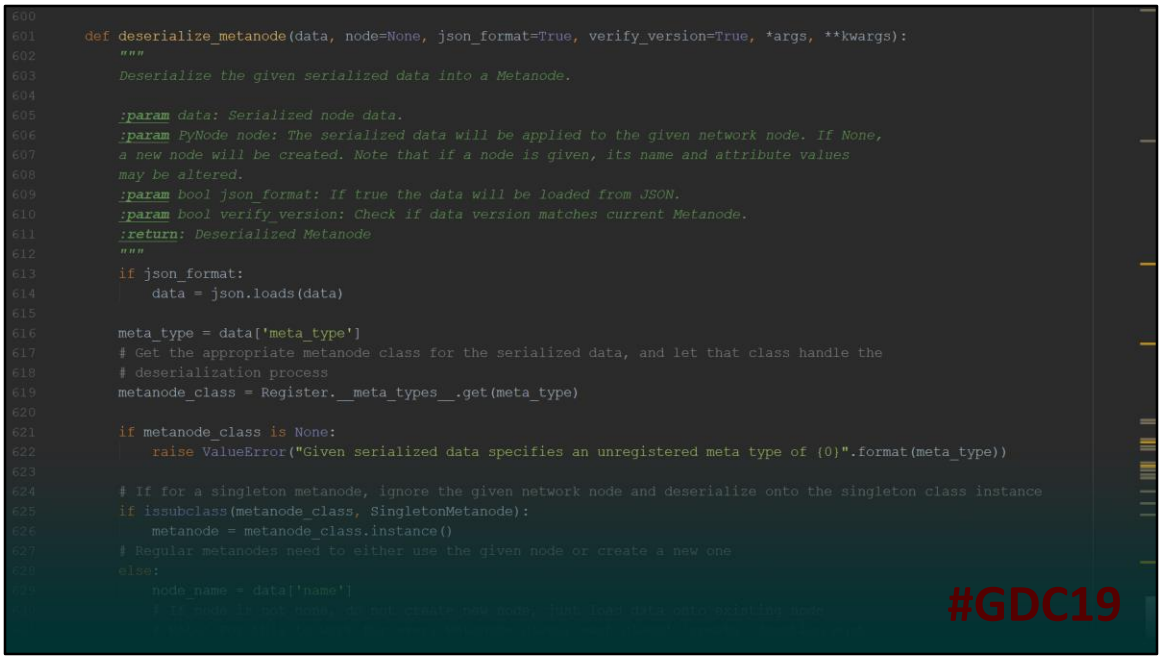
We can **expand** on the serialization system to **save off entire networks** of Metanodes like rigs.
Instead of referencing we can deploy our rigs through script.

```python
    def serialize(self, json_format=True):
        """
        Create a serialized representation of this node.

        :param bool json_format: formats serialized data as json
        :return: Serialized representation of this node
        """
        result = {'name': self.name, 'meta_type': self.meta_type, 'version': (self.node_version, self.node_lineal)}

        attributes = []
        for attrName in self.attr_class():
            serialized = self.serialize_attr(attrName)
            if serialized:
                attributes.append(serialized)
        result['attr'] = attributes

        dynamic_attr = []
        for attrName in self.attr_dynamic():
            serialized = self.serialize_attr(attrName)
            if serialized:
                dynamic_attr.append(serialized)
        result['dynamic_attr'] = dynamic_attr

        return json.dumps(result) if json_format else result

    def created_event(self):
        '''
        Create user events for attribute changes and node renames.
        '''
        sel_list = om2.MSelectionList()
```

Base serialize call **saves Core attributes**.

Class and Dynamic are serialized based on their attribute types with **serialize_attr**

Return as **JSON** for saving to file or raw **dictionary** for scene editing.



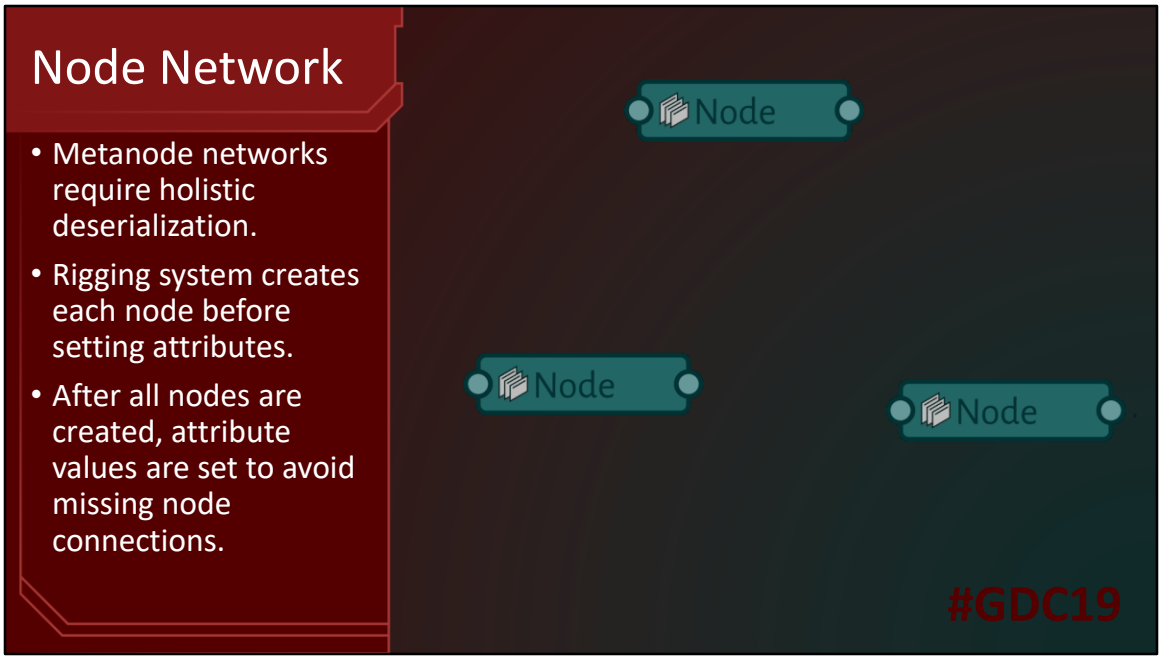
Deserialization is a **function outside of Metanode**.
Serialization dictionary is passed and the **meta_type is called from the Registry**.
Saved attribute **values are applied**

Node Network

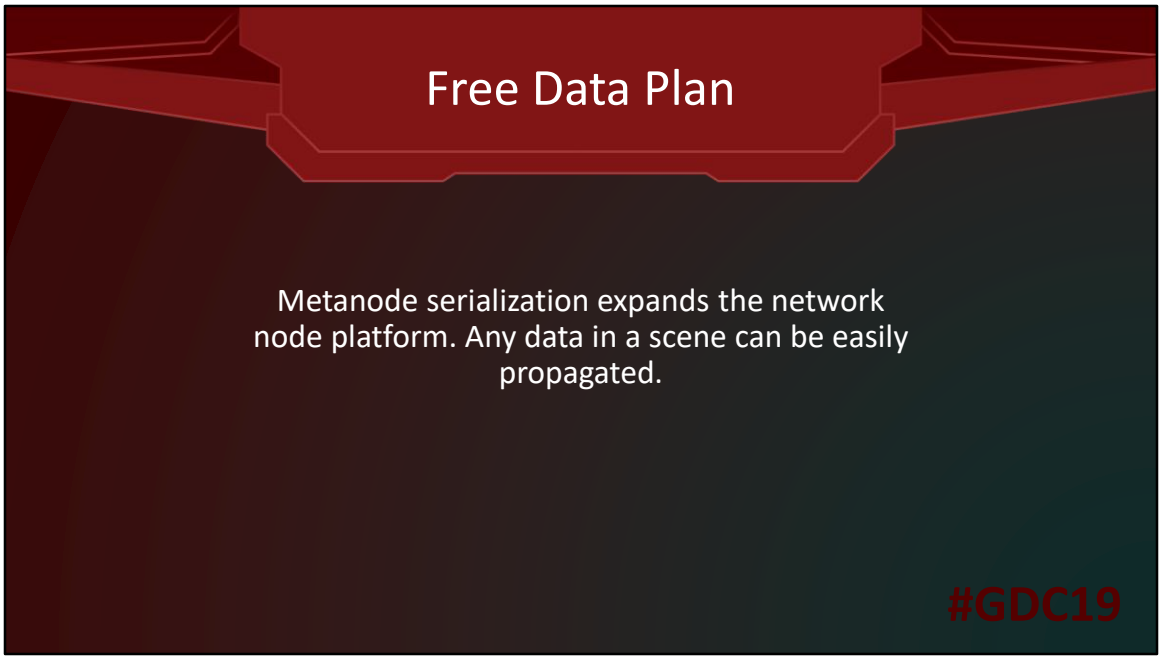
- Metanode networks require holistic deserialization.
- Rigging system creates each node before setting attributes.

Extend base serialization to networks
Rigs have a central rig Metanode with many connected components

We start by **creating every node** in the network.
Attributes are created but **not set**.

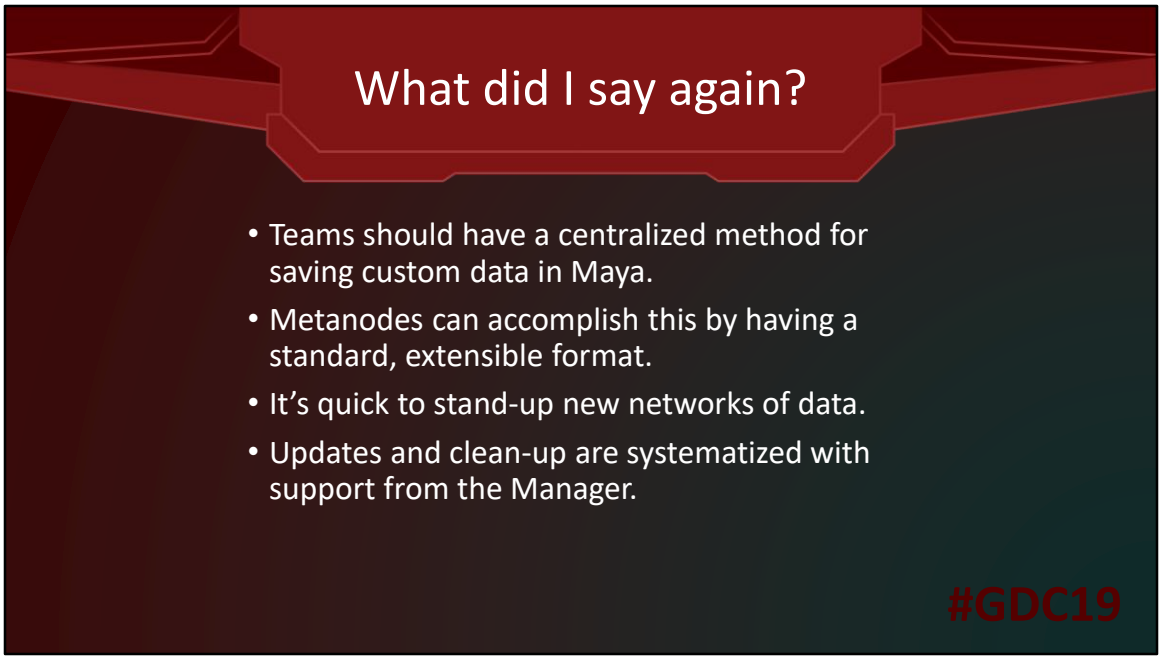


Once **all nodes exist** we can **check for name conflicts**.
Update data with any name changes
Set data



Metanode serialization **expands the possibilities** of our data.
Serialized data could be pushed to **other uses** outside of Maya.
Serialization, **example** of how a Metanode **standard** gives us a **foundation** to build on.

What did we learn today gang?

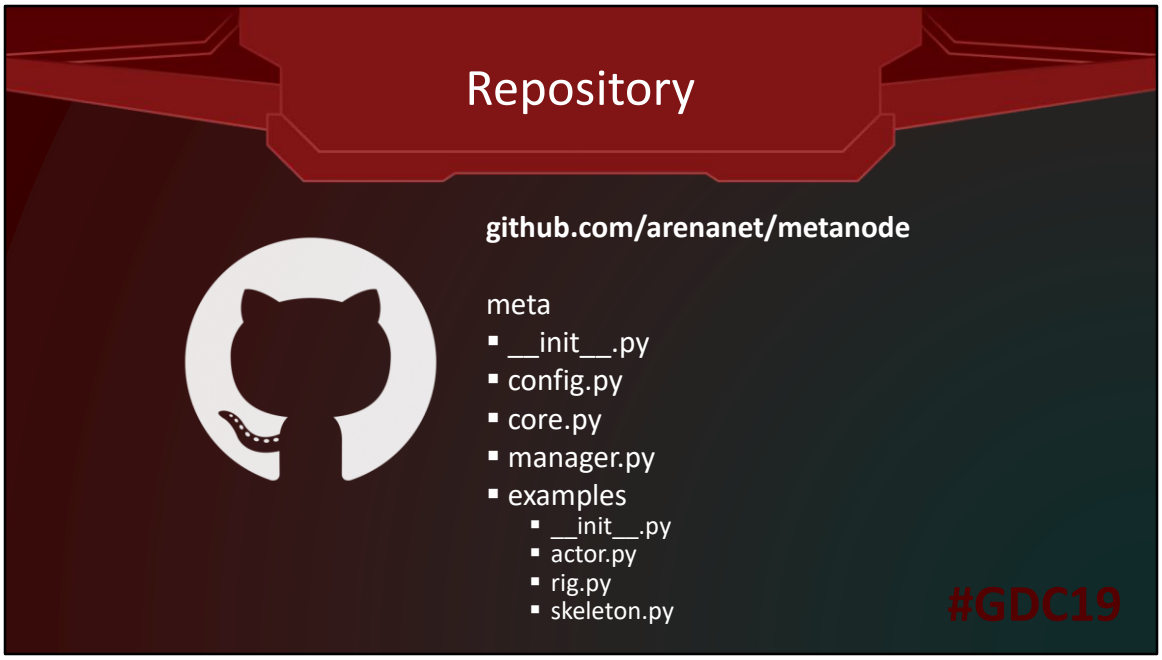


Having a **centralized** approach to storing data in Maya **saves time and sanity**. Team members have a **common standard to write** to.

The Metanode system **extensible to many uses**.

It's very **fast to create new Metanodes** and start **protyping**.

Metanodes are **maintainable** and have a clear path for **evolving with the project**.



Remember to go check out the repository!

If you need Technical Artists in the Seattle area come talk to me about the many talented TAs we recently had to let go from our team.

If you want to ask question about Metanodes
Get at me these places


ANDREW CHRISTOPHERSEN
andrewchristophersen@gmail.com
@def_tech_andrew
Andrew on tech-artists.slack.com
Questions?
#GDC19

Congratulations you made it!

#GDC19