A Programmer's Intro to Maya Scripting

Disclaimer: As of writing this, I am not a professional Maya programmer. This is just my attempt to make Maya Programming easier to learn for the next person, as I struggled to find effective resources online. **I'd be surprised if what I've written here didn't contain errors.**

If you have any corrections, additions, or general feedback, please don't hesitate to send them my way. I'd really appreciate your help!

When to use this guide:

1) You're committed to using Maya:
This is actually an important thing to think about. In fact, the most common advice I ever hear about Maya is "have you tried using Blender?" If you can't name a tangible reason for picking Maya, I encourage you to look at your options before starting anything too involved. Maya *is* the industry-standard for CGI, but it is far from the easiest software to use.

2) You're a confident programmer:
I'd say this guide is at about the level an upper-division Computer Science course. Solid Python skills and familiarity with your Machine's operating system are strongly recommended.

3) You know your Maya fundamentals:
I assume people reading this guide are comfortable performing basic tasks with the software. Maya's has some pretty robust learning resources to get you familiar with its user-interface. I will try to explain all the programming paradigms from square-one.

4) You're ready to get "in the weeds":
You can accomplish quite a lot in Maya without a single line of code. Even if a script could speed up your workflow, sometimes writing said script will take much longer than just doing things "by hand." If you're not programming for its own sake, make sure you've done some cost-benefit analysis.

Still here? Great! Let's get started.

Table of Contents:

Section 1: Scripting Overview

The most basic type of program you can run in Maya is a script.

Scripts are written in one of Maya's two main programming languages: MEL (the Maya Embedded Language) and Python3, which I will refer to as "Python" from here on out. Maya has separate, internal environments for scripts written in both of these languages. The differences between these languages will be easier to go over when they become relevant to our discussion of Maya programming as a whole.

For the most part, scripts run just like any program on your machine. They carry out a series of sequential operations whenever they're triggered, interacting with the user through a command line when necessary.

You have access to a traditional console in Maya's script editor window. Looking for a print statement? That's where it went. You can also import, save, and run scripts from this window. One small caveat is that Maya's user interface has certain states, like playing an animation, that it will await to exit before running a script that's been triggered.

An important thing to bear in mind about Maya scripts is that they can influence several different contexts. The names I've given them here are my own:

- **Language context**
    - Entities handled exclusively by your language's interpreter
    - Remembered automatically by the session
    - Deleted when the session ends

- **Session context**
    - Entities that configure your session

  - Remembered automatically by the session, but outside of a designated interpreter
  - Deleted when the session ends

- **Maya Context**
  - Entities that configure your software
  - Remembered automatically by your Machine's OS
  - Must be explicitly changed or deleted

- **Scene context**
  - Entities within your specific scene (.ma or .mb) file
  - Remembered by your scene if its saved before closing

Section 2: Environment

Let's break down the script environment a bit more before we explore these contexts in depth:

Maya executes all its scripts in a single interpreter per-language, per Maya session.
Note: It would be inaccurate to say that Maya's environment is single-threaded, because it does utilize certain multithreaded functionality alongside procedures to guarantee that execution in the interpreter is **strictly sequential.** Autodesk is very clear that the software is **NOT thread safe.**

Maya sessions are created when you boot the software up and their data is cleared from memory when you quit the software.

Every action taken in Maya's user interface, when it effects the state of the application, runs a pre-defined "payload" of scripted commands in the interpreter. You can see these commands echoed in the script editor console when it has certain setting selected.

Scripting at its core, is a process of defining and delivering custom, user-defined versions of these "payloads." The default way to deliver a custom payload is simply "running the script" via the script editor, but there are plenty of alternative triggers. You can even define your own custom triggers by sending additional commands to the interpreter.

Tl;dr?
You can think of every state change in Maya as an addition to one long, ongoing script that begins when you open Maya and clears when you close it.

Important Implications:

> The strictly sequential nature of Maya's script interpreter suggests that everything we execute is predetermined. However, because code is broken up into payloads, each with individual trigger conditions, we cannot know the number of times or the order in which these payloads will be interpreted.
>
> **Therefore, it is error-prone to assume what or how many state changes have occurred between payloads.** This is *especially* true when your Maya environment contains scripts from multiple sources and authors.

We'll go over how to write safe, effective code in this environment after we've discussed its different contexts in more detail.

Section 3: Language Context

Relative to the rest of a script, language context deals with **"backend" entities, or entities that only exist in their respective languages' interpreters:**

- Primitives
- MEL and Python's built-in data structures
- Identifiers (including those assigned to entities from other contexts).

Interpreters are tied to your current Maya session, which is created when you open the software and deleted when you close it. **Therefore, even if you save your file(s), all entities in language context will be deallocated when you close the software.**

The flip side is, if you define an entity at the global scope in one file, you can access it from any Maya file on your machine as long as the application stays open. This is true regardless of what you do with the file you had open when the entity was defined.

<u>**WARNING:**</u> Global scope in Maya is not just something you evoke with a keyword. **All language context entities at the top scope of a payload will be global!**

> **There is no implicit encapsulation of individual script payloads.** If an entity is at top scope in the payload, it will be at top scope in the session's interpreter which treats all payloads as part of the same script.

If you're worried about namespace collision, that's a very good instinct.

Namespace collisions are the biggest risk for entities in the script context. There will be more on how to combat them in sections 9 and 10.

Section 4: Maya Commands

Before we discuss the next three contexts, we're going to take a brief interlude about **Maya commands.**

Maya context, Session context, and Scene Context all use Maya commands to influence the state of the application, the current session, and the current scene respectively.

The terminology and documentation for Maya commands can be really confusing. I hope this helps clear some things up:

> **Maya commands are different from runtime commands.** Runtime commands are just wrappers for blocks of code, mostly for hooking up to hotkeys and certain UI elements.
>
> **MEL uses what are called "MEL commands"** to execute Maya commands.
>
> **Python uses the maya.cmds library, which needs to be imported to work.** Functions in this library were translated one-to-one from MEL commands, so you may feel they're not exactly suited to Python's syntax.
>
>> **Note:** Autodesk does promote a very popular third-party plugin called PyMEL, which is a much more "pythonic" wrapper for Maya's built-in Python library.
>
> **Maya commands influence the same data regardless of the language they're invoked in,** meaning the app and scene contexts can facilitate indirect interaction between MEL and Python scripts.
>
> **The large majority of Maya commands exist in both MEL and Python**, but if you compare the languages' online command libraries with each other, you'll notice that the Python library is missing something called MEL procedures.
>
>> **The explanation is quite simple, though:**
>> MEL procedures are evoked with a function call because "procedure" is just MEL's term for "function." Autodesk has just chosen to intersperse MEL's built-in procedures with MEL's Maya commands in the documentation, likely because these procedures often evoke a series of Maya commands

Determining which context a Maya command is influencing unfortunately boils down to a little strategic inference, so let's talk about what makes the contexts different.

> **Remember that I made these divisions up** to help people conceptualize Maya's environment. Don't fret about the minutia if it doesn't serve you.

Section 5: Session vs Script Context

Entities in the in the session context behave very similarly to entities in the script context. They don't need to be saved and effect the entire session but only last until you close the software.

The major difference between script context and session context is session context actually interacts with the application.

> Code for the session context will use Maya's command libraries or a custom Maya plugin, while code for script context will use basic operations and external libraries built "on top of" these basic operations.
>
> **The only time the script context will have anything to do with a Maya command is when you assign an identifier to the result of a command.**

Section 6: Maya vs Scene Context

Scene context and Maya context can both persist in memory after a Maya session ends. The difference between them is how they're saved, which corresponds to the type of data they handle.

To make scene context updates permanent, they must be saved with a Maya command or the user interface. They only exist on a per-scene basis.

> Scene context commands don't have any special markers, but, in practice, they're some of the easiest to identify. **If the changes you're making are specific to your scene, they're probably in scene context.**
>
> The DG or dependency graph contains all of the information pertaining to your scene (more on the DG in section 14). If you're wondering if a piece of data is scene specific, look for it in the DG with the Node Editor.
>
> **When all else fails, see if the changes a command makes persist after you close your scene file without saving.** If not, the changes are part of the scene context.

Changes to the Maya context are saved automatically for your entire application.

> This is because Maya preferences are stored on your machine and changes to these files are made in real-time.

Most of the settings relevant to Maya context are in a special file system associated with your Maya configuration. Autodesk refers to the parent directory of this file system as your "User App Directory." Changes to the information there can be a bit tricky to spot because Maya handles it in the background.

> **NOTE:** The User App Directory is different than the directory of Maya.app: the directory your OS uses to *launch* Maya itself. Autodesk refers to the directory of this launch-point the "Maya Install Directory".

One strategy is to set your script editor console to "echo all commands," which will usually echo the pathways to files its *creating* on your machine.

I haven't found a reliable way to check for *changes* to existing files in the script editor, but you can always investigate files in the User App Directory with your OS. These files are made to be customized, so they're relatively well-documented and easy to navigate.

Section 7: Scripting Overview

Now we return to our concern about Maya's script interpreter:

The Maya scripts we write could share a thread with any number of other scripts, that could execute in any order, any number of times. At least, these are the assumptions we need to make if we want to write the safest code.

How can we even begin to approach this environment?

Let's establish some things about the environment to help narrow it down:

> We are going to assume that no script would attempt to change an entity that it doesn't expect to exist.
>
> We are also going to assume that if any two given lines of code are intended to function in immediate succession, they will be delivered to the interpreter in immediate succession.
>
> **Therefore, the only situation where we should see a payload access or change an entity that has not been declared within the same payload, is if it expects a possible state change since the entity's declaration.**
>
> If you haven't caught on, this practice is very error-prone, because we have no way to guarantee that the state will (or won't) change in the way we want it to.

There are two categories of unexpected state changes that could cause problems:

A change to Maya's pre-defined global variables

Entities in Maya context are included under this umbrella

Handle pre-defined global variables in Maya with the same care that you would in any other environment. Don't assume their value or modify them without considering the consequences.

A change to an entity a programmer on the user-end has defined

Note: Unless a script is looping through all entities of a specific type, it must use the name of an entity to access it.

Hacking isn't really motivated in this type of programming, so we are going to assume that namespaces only overlap by accident. **This scenario will be referred to as a namespace collision.**

If the entity is a variable (script context), we can be sure that the variable is at the top scope. Payloads will not compile as incomplete statements, so any nested blocks that could contain a local variable must have already been closed.

Section 8: Script Triggers

Knowing when a script will run in our environment without us invoking it is essential for anticipating unexpected state changes.

There are only a handful ways a script can execute without you triggering it explicitly.

These are the ones I know of:

A: It was triggered by custom UI

Maya provides commands for creating custom session context instances of its internal UI assets.

Many of these assets can be configured with a function that will execute when certain interactions take place. While this function is not a payload in and of itself, the UI functions by delivering a payload consisting of a call to the specified function.

B: It was trigged by a **script job** **or a** **script node**

Script jobs and script nodes deliver a script payload in MEL to the interpreter when a specified condition is met.

Because script jobs are in session context, they can't trigger without being created by another script in the current session. This means script jobs can only participate in entirely implicit script execution in conjunction with another the source from this list.

Script jobs can be eliminated as a possible culprit in batch mode, which we will discuss more in section 12.

Script nodes, on the other hand, are scene context entities, so be wary of them. You can check on script nodes in your scene with the expression editor and, if you need to, turn off the Maya setting that executes script nodes automatically.

E: It was triggered by a **expression node**

These nodes are also viewable in the expression editor but they are distinct from script nodes. I'll provide a little more information them in section 15. For now, all you need to know is that they're a scene context entity that can run an encapsulated script when certain conditions are met.

The encapsulation prevents these nodes from adding new entities to the script context.

However: they can influence the other contexts regularly, as well as global entities that already exist in the script context.

F: It was triggered by a **setup script**

While they do not exist by default, files named userSetup.py and userSetup.mel in certain "scripts" directories of your Maya context folder have special predefined behavior.

If you have setup scripts in multiple languages and valid locations, they will execute as soon as your Maya session begins in the following order:

1. Maya/<version>/prefs/scripts/userSetup.py
2. Maya/<version>/scripts/userSetup.py
3. Maya/scripts/userSetup.py
4. Maya/<version>/prefs/scripts/userSetup.mel
5. Maya/<version>/scripts/userSetup.mel
6. Maya/scripts/userSetup.mel

Note: The default directory of these scripts is /…Maya/<version>/scripts (a subfolder of the User App Directory) NOT the scripts folder in your current project or anything in the Maya Install Directory.

E: It was triggered by an **API callbacks**

In addition to its maya.cmds library, the Python script interpreter can access another Maya-specific library for interacting with the software's lower level API. This API includes features for declaring callback functions to execute when certain conditions are met.

Its generally understood that the API shouldn't be utilized unless all of your options with built-in tools and the scripting library have been exhausted. **I am mentioning it here in an attempt to be comprehensive about the possible sources of implicit script execution**.

For a brief introduction to the API, see section 17.

F: It was triggered by an **evaluator function**
Both MEL and Python have access to evaluator functions that will send code as a string to the MEL or Python interpreter.

In Python these are typically exec() for evaluating Python and mel.eval for evaluating MEL.

In MEL these are typically python() for evaluating python and eval() or evalDeferred() for evaluating MEL.

Note: evalDeferred() will wait for Maya's next "idle time" to evaluate the String, typically after the payload that called it has finished executing, while eval() do so right away.

Even when these functions are called within a manually-triggered payload, they can still cause code to execute in an unexpected sequence if they are not handled carefully.

Calling these functions sends the code you provide as a string to the same session-wide interpreter, so all the same behavior quirks will apply.

Section 9: Namespace Collisions in Scene and Session Context

Session Context entities are handled in memory by the application's internal processes.

The only time you need to worry about namespace collisions is if you've connected an identifier to them in script context. This scenario is relatively common, as many commands you can evoke on existing session context entities require an identifier.

For scene context entities, Maya provides a convenient "Namespace" datatype that you can create with a scene context Maya Command.

> Utilizing Namespaces are particularly important when your scene may be importing or referencing other scenes that contain scene context entities.
>
> Unfortunately, because namespaces are in the scene context themselves, you can not use them to encapsulate any entities from the other three contexts.

Section 10: Namespace Collisions in Script Context

Entities with identifiers in the script context are the most "exposed" to namespace collisions in that they can be modified without Maya Commands.

> **NOTE:** To view all globally used namespaces, you can use print(str(globals())) in Python and env in MEL.

Let's go over how to combat this:

Option 1: Declare variables where you use them

> **Both MEL and python have strong, dynamic typing.** This means the data attached to a script context identifier will never change unexpectedly but it can be overridden with a new assignment of a different type.
>
> **If all of the variables you manipulate in a given payload are defined within said payload, this can work to your advantage:** You won't have to worry about namespace collision because your use of a given namespace will override any previous uses.

Option 2: Careful naming

> **Please be aware that this approach comes with no guarantees attached. It can make the possibility of namespace collision negligibly small, but, by definition, cannot eliminate it.**
>
> **Before you try this approach, remember:**
>
> > "The only situation where we should see a payload access or change an entity that has not been declared within the same payload, is if it expects a possible state change since the entity's declaration."

If you can't name what state change you're expecting, make sure you have it identified before moving forward. Ideally, you can avoid referencing entities defined in another payload.

Careful naming isn't just about picking the most obscure namespaces possible. There are a few more precise methods:

> **You can wrap everything in a function or a class**, bearing in mind the wrapper itself will still be global. This is a nice option if you just want something quick and easy.
>
> **If you want to accomplish the script context encapsulation above in a more "proper" way**, you can import your script as a module. Bear in mind that your module will be passed to the interpreter when it's imported. Wrap any lines that you don't want running right away in a function.
>
> **In Python, name mangling can help you make namespaces even more protected when using the methods above.**

Section 11: Paths, the Ecosystem, and the Environment

We should talk a little bit more about how the scripting environment interacts with the rest of the application and, to an extent, the rest of your machine.

Our discussions of the Maya Context and setup files has already touched on this concept a bit.

To summarize:
Maya context entities exist as files saved to your machine within Maya's ecosystem. Some special directories within this ecosystem are added to the script context's environment.

Recall: In a special case for several subdirectories of The User Application Directory called "scripts", files with the name "userSetup.py" or "userSetup.mel" will run automatically at the beginning of every Maya session.

If you tested out (str(globals())) in Python or env in MEL from section 10, you probably noticed that quite a few global variables are declared by default. Global variables you create are appended onto the end of the list.

Global variables are not the only built-in entities relevant to your Maya scripting environment.

You can read a list of what are called "internal variables" using the internalVar command in both MEL and Python. (I haven't been able to find an official definition of this term. The following is what I gathered from context):

> All of the internal variables you can access with internalVar are paths to certain entities in the Maya context.
>
> The list of viable inputs for the internalVar is predefined. For said list, see Autodesk's documentation of the internalVar command itself.
>
> **Internal variables, as their name suggests, are used internally by the application. Scripts are not able to modify them.**

What you can modify are the script interpreter's environment variables.

> **These are manipulatable using the standard "os" and "sys" libraries in Python and the commands: getenv and putenv in MEL.**
>
> > **NOTE:** Despite the similar naming convention, getenv and putenv interact with environment variables while env lists the MEL interpreter's global variables.
>
> You can also configure environment variables using a Maya.env file in the User App Directory.
>
> **Some internal variables have values that overlap with environment variables. However, no matter how you modify environment variables, the internal variables will remain the same.** Think of internal variables like the default values for any environment variables they overlap with.

Section 12: Executables

I mentioned previously that the User Application Directory is distinct from the Maya.app, or the Maya Install Directory.

Even though they're hidden by most OSs, the install directory does, in fact, have subfolders.

The content of these folders is, broadly speaking, source code. That is to say it isn't really intended for user customization. That's what the User App Directory is typically for.

However, there are useful files in Maya.app directory, most notably, shell executables for interacting with Maya without its user interface.

You may frequently see “headless mode” used as an umbrella term for all interaction methods of this type.

NOTE: There is a *large* volume of shell executables included in subdirectories of Maya.app and the vast majority of them are entirely undocumented. The most we can assume is that these undocumented executables are there for internal use or compatibility with legacy code.

To avoid being overwhelmed, assume you are only responsible for what is explicitly documented for the version of Maya you are using. I have heard this is standard practice in the industry as well.

The main executables relevant to Maya scripting are “maya” for MEL and “mayapy” for Python.

NOTE: The path to these executables are not part of the default path for shell executables ($PATH). You will need to somehow change your environment to include the paths or use the entire path to an executable when you invoke it.

The path you’re looking for should be readable via either of these entities:

- The MAYA_LOCATION environment variable + “/bin/<name>”
- The internal variable, mayaInstallDir(mid) + “Maya.app/Contents/bin/<name>”.

The Maya sessions created via launching an executable will be distinct from a session created via launching the application itself.

Concurrent Maya sessions do not communicate with each other.

When you open a scene with a Maya session, it will only have access to the most recently saved iteration of the said scene.

Therefore, if you open the same scene in two different Maya sessions at once, you will only be able to keep the changes made in one of those sessions.
Even if you save in both sessions, the session you save from second will overwrite changes saved from the session prior.

To be precise: While the maya executable uses the same internal MEL interpreter as the script editor, **mayapy uses a separate external python interpreter installed with Maya.**

However, because the maya executable launches a new session, encapsulating the mel commands you invoke, I have yet to find a case

where mayapy's external environment makes it distinct in terms of scope or threading.

There is also an external environment for python2 specifically, which can be launched with mayapy2.

The one tangible difference between python in the script interpreter and python in mayapy is that **mayapy requires the Maya.standalone module to interact with Maya files.**

The maya executable has two notable modes: "batch" and "prompt."

These are either separate executables in the same directory ("mayabatch" and "mayaprompt") or flags for the maya executable depending on your OS.

It should be mentioned here that using the "maya" executable without a special mode launches the software with the standard GUI.

There is no variation of the mayapy command that will do this.

Be careful about closing the command line you used to launch Maya without terminating the process, which will cause Maya to crash. The method for doing so varies by OS.

Prompt Mode: Prompt mode is a special way to launch Maya without its graphical user interface via your OS's command line.

The mayapy counterpart for prompt mode is the executable's default.

Much like when you launch Maya with the GUI, you can specify a file to open at launch.

Scripting in prompt mode is identical to using the script editor save for a few missing commands that require the user interface.

Batch Mode: Batch mode very similar to prompt mode, except it requires every line in the script you want to execute up-front.

The mayapy counterpart for prompt mode is invoked when you specify a specific python file to run.

As mentioned earlier, script jobs are not compatible with batch mode, so they can be ruled out as a source of unexpected behavior for batch mode debugging.

Section 13: Network Protocols

A particularly useful Maya command that deserves its own section is commandPort (available in Python and MEL).

This command is one of the few multithreaded, but still synchronous uses of Maya's scripting environment that I alluded to earlier.

What commandPort does is allow the script editor to intake commands from a "localhost" web address, specified when you invoke the command.

It opens up a lot of functionality for piping external, dynamically-generated commands into Maya through whatever server you choose.

If you want to interact with Maya like you would in batch or prompt mode, but see the real-time consequences in Maya's GUI, this is the solution for you.

Beyond Scripting

Scripting is just one facet of a programmer's ability to interact with Maya.

It's generally considered the best starting place for new Maya programmers because it's relatively high-level, and many other "inroads" to Maya are slight variations on scripts.

Disclaimer: I haven't done rigorous enough research to cover the topics beyond this point in full, but wanted to mention them in this document to make sure you have a more complete overview.

Section 14: The DAG and DG

DAG and DG stand for **"directed a-cyclical graph"** and **"dependency graph"** respectively.

The dependency graph is what Maya uses internally to describe and calculate all information specific to a scene. You can accomplish some very advanced functionality just by manipulating these nodes in Maya's GUI.

> **Note:** You can get some idea of how the DG will evaluate upon opening a scene by investigating the scene in the Maya ASCII (.ma) format with a text editor.
>
> **Nodes in the dependency graph represent all the different operations needed to create a scene, along with all its data "from scratch."**

These operations can be tangible, like the creation or modification of an object. Others are strictly computational and may have nothing to do with any specific object at all.

Because operations in the DG are evaluated near-continuously, most can be thought of as dynamic relationships or processes.

Therefore, nodes representing the creation of a given <object> are named <object>, instead of <create object>.

Here's how the DG is evaluated: When an attribute of any node in the dependency graph changes, Maya traverses the graph to find all other attributes the change will effect and flags them as "dirty." This process is called dirty propagation. Maya then only updates the information in these nodes.

If you want to take a look at the dependency graph with Maya's UI, head over to the hypergraph tool.

The directed a-cyclical graph, also known as the "transformation hierarchy" is a subset of nodes in DG that represent objects (shape nodes) and any number of transformations effecting them (transform nodes).

If you've looked at an object's data in the node editor before, the shape nodes default to the naming convention "<identifier>Shape" and transform nodes only include the corresponding identifier.

Shape nodes are always the leaf nodes in the DAG. Anything upstream of them, which you may see in the node editor, is just for utility and part of the DG.

This is actually why it's important to clear the history of objects in your scene. The history of a shape node is represented by a chain of upstream transform nodes in the DAG, which can get quite costly to maintain.

Clearing an object's history removes this chain and make the values it outputs equal to an expression (the generic syntactic entity, not anything specific to Maya) or a constant.

Section 15: Expression Nodes

Refer the first mention of these nodes in section 8, E for an introduction to these entities.

<u>**Expression**</u> **nodes exist at the DG level, as do script nodes, and contain scripts that will execute when certain conditions are met.**

There is no way to reference local entities outside of a given expression node's scope, but print statements will still output to the terminal in the script editor.

The scripts can only be written with a special superset of MEL that contains new grammar for a special "expression" entity (not to be confused with the expression node).

Expressions declaratively define a relationship between attributes of nodes in the DG.

Expression nodes will not evaluate if their script does not contain an expression, and not compile if the expression is not the last line in said script.

Expression nodes are directly tied to objects in the scene and, therefore, evaluate "dynamically."

To be more precise:
Expression nodes always evaluate once when you create them or finalize edits to them (unlike script nodes).

When an expression node is set to evaluate "on demand" it will evaluate whenever the input value changes.

When it is set to "always" it will **also** evaluate whenever you change the current animation frame.

Be careful evoking standard MEL commands in both expression and script nodes. Certain conditions will cause the scripts to evaluate continuously and you will get behavior akin to an infinite loop.

Section 16: DASH

<u>**DASH**</u> **is a very small language that you can use in lieu of expression nodes for simple operations.**

It only supports inline operations and is not for creating dynamic relationships between DG objects.

I am giving it a brief mention here because it is technically a third language relevant to programming in Maya's GUI.

Section 17: Plugins and The API

It would be remiss of me not to at least mention Maya's API.

This is a whole different, lower level of Maya programming that customizes the behavior of the software itself.

As I mentioned earlier in this guide, scripting, as it relates to Maya, is built on top of a series of commands. When these commands pertain to a scene, they are manipulating nodes in the DG.

Notice that both Maya commands and DG nodes have behavior that has been defined at a lower level.

> **The API allows third-party developers to create and customize these entities as well as manipulate other low-level operations.** It is as close as you can get to Maya's source code itself (without violating the terms of service).

API code is typically added to the software as a pre-packaged "plugin."

> **Most professional-grade plugins come with easy installation scripts that anyone can run, regardless of programming experience.**
>
> **However to write your own, you need to download and configure the Maya SDK (software development kit).**
>
> Plugins can be written in C++ or Python. In other words, those are the two languages that have access to the API.

Resources

From Autodesk:

Maya, MEL, Python, Nodes, Pymel, API 2.0 (Python)

> **Note:** The "Nodes" URL is not working but the documentation can be found in the general Maya doc under 'Technical Documentation' > 'Nodes'

Location of Maya configuration folder (by OS)

Python API overview

Third Party:

Introductory:

- Maya TD Primer
- Maya Scripting Primer
- Recourse Directory

Executables:

- Network Protocol Instructions
- Headless Batch Processing
- More Headless Batch Processing

Scenes:

- List of All Maya Environment Variables
- Maya Scene Data
- DG Intro
- DG Video Explanation #1
- DG Video Explanation #2

Expressions:

- Intro to Expressions
- Expressions In-Depth (old)

Misc:

- Mayabatch & Mayapy Comparison
- Some very helpful GUI Gists