PyroCore API Revision 1.2 (2019-02-15)

Author : Levi Perez

Basic Structures

Behavior

- A C++ component attachable to a GameObject. Contains an arbitrary number of IScript objects. The following is just a global function adding a Python script called ‘MyBehavior’ to the Behavior component of a GameObject:

Example 1 : Attaching a Behavior

```
# EG. 1 : Attaching a Behavior (assumes MyBehavior is a defined IScript)
from PyroCore import *

def AttachMyBehavior(obj): # obj is a GameObject
  behavior = None
  if obj.HasComponent(Behavior):
    behavior = obj.GetComponent(Behavior)
  else:
    behavior = obj.AddComponent(Behavior)
  behavior.AddPyScript(MyBehavior)
```

IScript

- The main script class that you inherit from in Python. Equivalent to a Unity MonoBehavior. Currently, you may define any of the following callback functions, which work similarly enough to Unity:

Example 2 : Defining IScript Callbacks

```
# EG. 2 : Defining IScript Callbacks
def Start(this): # 'this' is equivalent to 'self' you will see online.
  pass # 'pass' tells Python to ignore this function.
      #  Having just an empty function will throw a syntax error.
def Update(this, dt): # dt == Time.deltaTime (which you may also call)
  pass
def LateUpdate(this, dt): # Called after everything has updated once.
  pass
def Shutdown(this): # Not in Unity, but called when the Behavior is destructed.
  pass              # Great for cleanup.
def OnCollisionEnter(this, thiscoll, thatcoll):
  # Note: 'this' is the IScript. 'thiscoll' and 'thatcoll' are IColliders.
  pass
```

These functions are indented underneath a class declaration inheriting from IScript (shown below).

Example 3 : IScript Callbacks (cont.)

```
# EG. 3 : IScript Callbacks (cont.)
from PyroCore import *

class MyBehavior(IScript): # This is how you inherit from a class.
  def Start(this):
    this.timer = this.gameObject.CreateAttribute("timer", Time.CreateTimer(5.0))
    # More on the Timer class in future sections.
```

Scripts may be dragged-and-dropped from the 'Scripts' tab in the Asset Selector. Pay attention to the mouse tooltip; dragging on top of an existing GameObject attaches the script to that GameObject, whereas dropping on nothing creates a new GameObject with the script attached. Tip: Hold ALT to see what you are hovering over, or to drag onto a GameObject that is tagged as 'Unselectable'.

Properties

The following members are available for you to access in Python:

+ `this.gameObject` - access to the GameObject that the IScript is attached to. This is a popular one.
+ `this.transform` - access to the GameObject's Transform component. Shorthand for this.gameObject.transform.

Anything else that you guys need shorthand for? Please write below.

IFactory

— The class that you inherit from in Python in order to create instantiation commands. Useful as a sort of 'Prefab' equivalent, though it uses code rather than the editor to store creation logic. Define the 'Init' and/or 'Create' functions to achieve this functionality. Additionally, you may use these functions to just run code at runtime. In other words, they don't *have* to return anything, nor do they *have* to create GameObjects. They can be dragged-and-dropped from the 'Factories' tab in the Asset Selector in-editor.

Example 4 : Writing an IFactory

```python
from PyroCore import *

# EG. 4 : Writing an IFactory
class ShitCreator(IFactory):
  isInit = False # if this doesn't exist, Init won't be called.
  def Init(engine):
    Debug.LogUser("Spitting out all the shit...")
    # 'Debug.LogUser' logs cyan text, which may be easier to differentiate.
    isInit = True

  def Create(system, name):
    gameobject = system.CreateGameObject(name)
    shit = system.CreateGameObject("shit")
    gameobject.AddChild(shit)
    return gameobject # returning is optional, but may be desired.
```

Parameters

'engine' - access to the C++ Engine instance currently running. More on the available functions of the Engine class later in this document.

'system' - access to the C++ GameObjectSystem of the current scene. More on the available functions of the GameObjectSystem class later in this document.

'name' - a string passed to the Create function, usually representing what to name a created GameObject. You can do anything with the string, or ignore it completely. Remark: will be equal to the name of the class plus some number if the factory is dragged-and-dropped from the Asset Selector.

GameObject

Public Properties

.name	The name of the GameObject. Read/Write.
.transform	The Transform (position, scale, rotation) of the GameObject.
.system	The GameObjectSystem (used primarily for creating new GameObjects) that the GameObject belongs to.

Inherited from IEntity

.uid	The ID (a number) associated with the GameObject. Guaranteed to be unique within the same GameObjectSystem[1]. Read only.
.isDeleted	Whether or not the GameObject is up for deletion. Read only.
.isEnabled	Whether or not the GameObject is enabled. Read only.

Public Methods

At(float, float)	A quick way to set the X-Y position of the GameObject.
HasTag(Tags)	Checks if the GameObject has the given Tag(s). To check for multiple tags, try OR-ing them together using the vertical bar symbol. EG: that.**HasTag(Tags.Player | Tags.Attacking)**
SetTag(Tags)	Sets the given Tags on the GameObject.
ClearTag(Tags)	Clears the given Tags from the GameObject.
GetTag()	Gets all Tags on the GameObject.
IsTrigger()	Shorthand for **HasTag(Tags.Trigger)**.
IsPersistent()	Shorthand for **HasTag(Tags.Persistent)**.
SetPersistent()	Shorthand for **SetTag(Tags.Persistent)**.

[1] Future implementations may guarantee that the UID is unique within the entire runtime of the game, however this is not the current behavior.

HasAttribute(string)	Whether or not the GameObject has an Attribute with the given label. See 'Attributes' section for more information.
CreateAttribute(string, obj)	Creates an Attribute with the given label and attaches it to the GameObject. Parameter 'obj' can be any type. See 'Attributes' section for more.
SetAttribute(string, obj)	Sets the value of the Attribute with the given label on the GameObject. Parameter 'obj' can be any type, though it is safest if it is the same type as was passed to **CreateAttribute**.
GetAttribute(string)	Gets the value of the Attribute with the given label on the GameObject. Returns None and prints an error to the logs if there is no such Attribute on the GameObject.
RemoveAttribute(string)	Removes the Attribute with the given label from the GameObject.
GetSystem()	Equivalent to accessing the **.system** property of the GameObject. Returns the GameObjectSystem that the GameObject belongs to.
HasComponent(class)	Whether or not the GameObject has a particular kind of component. Pass the name of the component's class (**no quotes**). EG: go.**HasComponent(AudioEmitter)**
AddComponent(class)	Adds the given component type to the GameObject. Pass the name of the component's class (**no quotes**). EG: go.**AddComponent(Physics)**
RemoveComponent(class)	Removes the given component type from the GameObject. Throws an error to the logs if no such component type exists. Pass the name of the component's class (**no quotes**). EG: go.**RemoveComponent(Colliders)**
GetComponent(class)	**This is a popular one.** Gets the given component type from the GameObject. Returns None and throws an error to the logs if no such component type exists. Pass the name of the component's class (**no quotes**). EG: sprite = this.gameObject.**GetComponent(Sprite)**
IsInBounds(Vector2)	Tests to see if a given position is in bounds of the GameObject. The functionality of this is limited at the moment, and should **NOT** be used in place of detecting a collision.
IsSceneRoot()	Whether or not the GameObject is the Scene's root

	manager object (UID == 0).
HasParent()	Whether or not the GameObject has a parent GameObject.
GetParent()	Gets the GameObject's parent. Returns None and throws an error to the logs if it has no parent.
HasChildren()	Whether or not the GameObject has child GameObject(s).
IsChildOf(GameObject)	Whether or not the GameObject is a child of the given other GameObject.
HasChild(string)	Whether or not the GameObject has a child with the given name.
FindChild(string)	Finds the GameObject's child that has the given name. Returns None and throws an error to the logs if no such GameObject exists.
FindChildRecursive (string)	**Unimplemented.** When implemented, this function will search all children, grandchildren, great-grandchildren, and so on until it finds the GameObject with the given name. For the time being, you are better off just not having child GameObjects that are that far nested.
AddChild(GameObject)	Adds the given GameObject as a child of this GameObject.
RemoveChild(string)	Unchilds the given GameObject from this GameObject, if this GameObject has a child with the given name.
Delete()	Marks this GameObject for deletion.
Check()	Checks if this GameObject is in a valid state.

Please write any additional functionality you would like here.

Attributes

Attributes are a lot like any member variables that you can create and use in a class. The key difference is that Attributes can be viewed and edited real-time in the editor, as well as serialised. Sure, they are currently clunky to use, but trust me, it is well worth the extra typing.

Supported In-Editor Types

The currently supported types that can be viewed in-editor are:

- Integers	(5)
- Floats	(5.0)
- Booleans	(True)
- Strings	(“Five”)
- Timers	(Time.CreateTimer(5.0))
- GameObjectLists	(GameObjectList.Create())

Relevant GameObject Functions

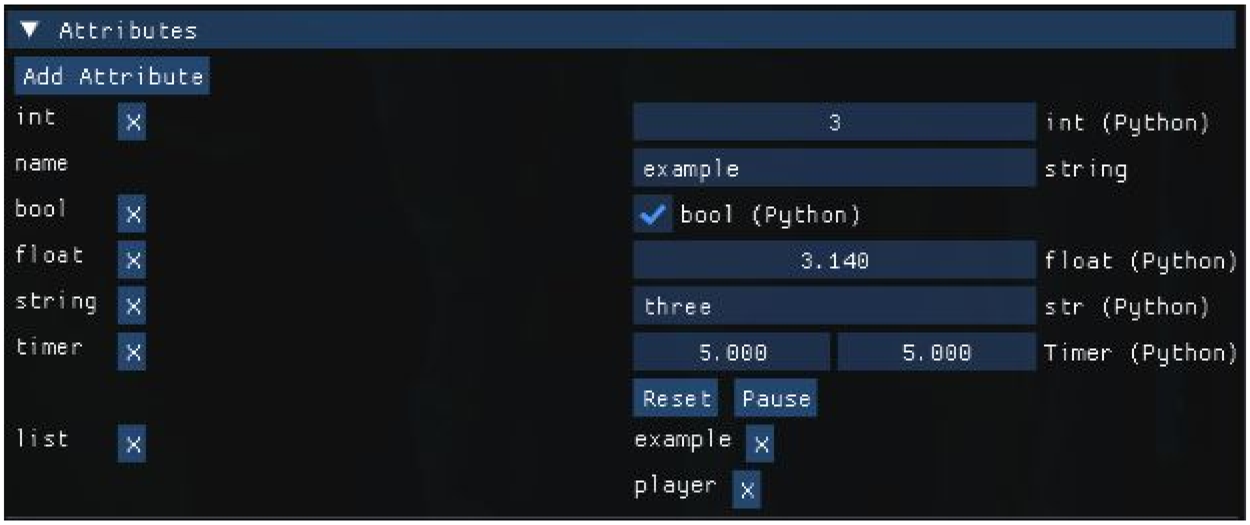
HasAttribute(string)	Whether or not the GameObject has an Attribute with the given label.
CreateAttribute(string, obj)	Creates an Attribute with the given label and attaches it to the GameObject. Parameter ‘obj’ can be any type.
SetAttribute(string, obj)	Sets the value of the Attribute with the given label on the GameObject. Parameter ‘obj’ can be any type, though it is safest if it is the same type as was passed to **CreateAttribute**.
GetAttribute(string)	Gets the value of the Attribute with the given label on the GameObject. Returns None and prints an error to the logs if there is no such Attribute on the GameObject.
RemoveAttribute(string)	Removes the Attribute with the given label from the GameObject.

Example 5 : Attributes

In Code:

```python
# EG. 5 : Attributes (assume the following is part of an IScript class)
def Start(this):
  this.integer = this.gameObject.CreateAttribute("int", 3)
  this.float = this.gameObject.CreateAttribute("float", 3.14)
  this.boole = this.gameObject.CreateAttribute("bool", True)
  this.string = this.gameObject.CreateAttribute("string", "three")
  this.timer = this.gameObject.CreateAttribute("timer", Time.CreateTimer(5.0))
  this.list = this.gameObject.CreateAttribute("list", GameObjectList.Create())
  # add some GameObjects to demonstrate GameObjectLists
  this.list.append(this.gameObject)
  # .append() can also be called with a capital 'A'
  this.list.Append(this.gameObject.system.FindWithName("player"))
```

In Editor: (after Start() is called)



```
▼ Attributes
Add Attribute
int     X                3                  int (Python)
name                     example            string
bool    X                ✓ bool (Python)
float   X                3.140              float (Python)
string  X                three              str (Python)
timer   X                5.000     5.000    Timer (Python)
                         Reset  Pause
list    X                example X
                         player X
```

Notes:

- In the left column are the names of the Attributes. The 'X' by each allows you to delete the Attribute. Notice how you cannot delete 'name'... bad idea!
- In the right column are value adjusters for the respective types. Next to the adjusters are the types of the Attribute, and it will say '(Python)' next to it if the Attribute was created in Python.
- The 'X's in the 'list' Attribute allow you to remove GameObjects from the list. If you hold CTRL, it will also delete the GameObject.
- The 'Add Attribute' button allows you to create new Attributes.

Components

AudioEmitter

The component that allows a GameObject to emit audible sound, either music or sound effects. Can be set to act in 3D space, or to be native stereo/mono (See method **SetRelative**). Note: will not be able to play any sound until **SetSound** is called. **SetSound** should always be the first method you call!

Inherited from IComponent

ToString()	Returns the name of the component as a string.
Suspend(bool)	Suspends the component from updating and playing.
IsSuspended()	Whether or not the component is suspended.
Enable(bool)	Does nothing (for now).
IsEnabled()	Whether or not the thing is enabled.
UID()	The unique ID of the parent GameObject.
.uid	The unique ID of the parent GameObject. Read only.
.gameObject	The GameObject this is attached to. Read only.

Public Properties

.name	The name of the sound, EG "Kora_Attack". Same as the filename, minus the extension. Read only.
.duration	The duration of the sound, represented by a TimeOffset object (see below). Read only.
.status	Playing status of the sound. Read only. Possible values: - PlayStatus.Playing - PlayStatus.Paused - PlayStatus.Stopped
.looping	Whether or not the sound loops. Read/Write.
.volume	The volume of the sound. Read/Write.
.pitch	The pitch of the sound. Read/Write.

.minDistance	The minimum distance at which the sound is audible. Read/Write.
.attenuation	The dampening applied to the sound's amplitude. Read/Write.
.isRelative	Whether or not the sound is relative to its listener. Read/Write.
.offset	The current playing offset (position in the audio file) of the sound. Read/Write.
.zOffset	A fucky thing. Play around with it if isRelative is set to True. Should be 100.0 if isRelative is False.

Public Methods

Play()	Plays playback from the current position.
Pause()	Pauses playback at the current position.
Stop()	Stops playback and resets the current position to the beginning.
SetSound(string, bool)	Sets the sound to play. Sound names are equal to their filename minus the extension. The second parameter is whether or not the sound is a sound effect; currently, all .wav files are considered sound effects and all .ogg files are considered music.
GetDuration()	Returns the duration of the sound as a TimeOffset object (see more on the TimeOffset class below).
GetPlayingStatus()	Returns the current playing status of the sound. Possible return values are: - PlayStatus.Playing - PlayStatus.Paused - PlayStatus.Stopped
GetLooping()	Returns whether or not the sound loops.
SetLooping(bool)	Sets whether or not the sound should loop.
GetVolume()	Returns the volume of the current sound.
SetVolume(float)	Sets the volume of the current sound.
GetPitch()	Gets the pitch of the current sound (the default is 1.0).
SetPitch(float)	Sets the pitch of the current sound.

GetMinDistance()	Gets the minimum distance at which the sound is audible, if the sound is considered relative. See GetRelative/SetRelative below.
SetMinDistance(float)	Sets the minimum distance at which the sound is audible. Has no effect if the sound is not relative.
GetAttenuation()	The dampening amount currently applied to the sound.
SetAttenuation(float)	Sets the dampening amount currently applied to the sound.
GetRelative()	Returns whether or not the sound is considered relative to the listener (spatial audio).
SetRelative(bool)	Sets whether or not the sound is to be considered relative to the listener (spatial audio).
GetTimeOffset()	Returns a TimeOffset object representing the current position in the sound.
SetTimeOffset (TimeOffset)	Sets the current position of the sound to the given TimeOffset (see below).
GetZOffset()	The current Z-offset. A fucky thing. Play around with it if isRelative is set to True. Should be 100.0 if isRelative is False.
SetZOffset(float)	Sets the Z-offset. A fucky thing. Play around with it if isRelative is set to True. Should be 100.0 if isRelative is False.
IsSFX()	Returns whether or not the sound represents a sound effect (.wav) or music (.ogg).

Any additional functionality requests? Please write below.

TimeOffset

TimeOffset is a class that represents a certain duration of time.

Creation

The following functions are ways of creating TimeOffset objects:

```
seconds = TimeOffset.FromSeconds(5.0)              # represents 5 whole seconds
milliseconds = TimeOffset.FromMilliseconds(5000)    # same as above
microseconds = TimeOffset.FromMicroseconds(5000000) # same as above
```

Conversion

The following functions help you convert values from a TimeOffset object:

```
# continuation from the above snippet
actualSeconds = seconds.AsSeconds()           # == 5.0
actualMilliseconds = seconds.AsMilliseconds() # == 5000
actualMicroseconds = seconds.AsMicroseconds() # == 5000000
```

Example 6 : Setting Up an AudioEmitter

```
# EG. 6 : Setting Up an AudioEmitter
from PyroCore import *

class MyAudioFactory(IFactory):
  def Create(system, name):
    obj = system.CreateGameObject(name)
    emitter = obj.AddComponent(AudioEmitter)
    emitter.SetSound("Heartbeat", True) # True because Heartbeat is a .wav
    # Note that SetSound must be called before any other setup!
    # You have been warned.
    emitter.volume = 95.0 # Values between 0.0 and 100.0 are valid.
    emitter.pitch = 0.75  # 1.0 is normal pitch.
    emitter.looping = True
    emitter.isRelative = True
    emitter.zOffset = 2.5 # This is a fucky one. 2.5 seems good.
    return obj
```

AudioListener

This is a sparse class, and will be of little use to you. The only use cases I can currently come up with is for when you create the player's GameObject, or when you create a camera GameObject (for cutscenes?). There should only be one active AudioListener in a scene at any one time.

Inherited from IComponent

ToString()	Returns the name of the component as a string. Static.
CloneInto(GameObject)	Clones this component into another GameObject. Generally, a bad idea with an AudioListener.
Suspend(bool)	Does nothing (AudioEmitters won't play anyway).
IsSuspended()	Whether or not the component is suspended.
Enable(bool)	Does nothing (for now).
IsEnabled()	Whether or not the thing is enabled.
UID()	The unique ID of the parent GameObject.
GetGameObject()	The GameObject this is attached to.
.uid	The unique ID of the parent GameObject. Read only.
.gameObject	The GameObject this is attached to. Read only.

Adding to a GameObject

Like with any component, this is how you add an AudioListener to a GameObject:

```
player = system.CreateGameObject("player")
player.AddComponent(AudioListener)
```

Behavior

The Behavior component is a container of all scripts (IScript inheritors) attached to a GameObject.

Public Methods

HasScript(class)	Whether or not the Behavior holds the specified class name.
AddPyScript(class)	Adds a given Python IScript to the Behavior component.
RemoveScript(class)	Removes a script from the Behavior component.
GetPyScript(class)	Gets the Python object holding the IScript instance. 'None' will be returned if no such script exists on the Behavior.
Reload()	Reloads all scripts attached to the Behavior. May also reload other instances of the script(s) if they are also attached to other GameObjects.

Additional methods/properties are inherited from IComponent.

Adding a Script

Taken verbatim from an above section:

```
# EG. 1 (assumes MyBehavior is a defined IScript)
from PyroCore import *

def AttachMyBehavior(obj): # obj is a GameObject
  behavior = None # checking if obj already has a Behavior is best practice
  if obj.HasComponent(Behavior):
    behavior = obj.GetComponent(Behavior)
  else:
    behavior = obj.AddComponent(Behavior)
  behavior.AddPyScript(MyBehavior)
```

Colliders

'Colliders' is a component that contains an arbitrary number of concrete colliders of various types. Currently, the only supported collider types are CircleCollider and SegmentCollider, which are documented later in this section.

Inherited from IComponent

ToString()	Returns the name of the component as a string.
CloneInto(GameObject)	Clones this component into another GameObject.
Suspend(bool)	Inhibits checking for collisions.
IsSuspended()	Whether or not the component is suspended.
Enable(bool)	Does nothing (for now).
IsEnabled()	Whether or not the thing is enabled.
UID()	The unique ID of the parent GameObject.
GetGameObject()	The GameObject this is attached to.
.uid	The unique ID of the parent GameObject. Read only.
.gameObject	The GameObject this is attached to. Read only.

Public Properties / Methods

.isDirty	Is the component awaiting calculation? Read/Write.
IsDirty()	Is the component awaiting calculation?
SetIsDirty(bool)	Sets the component to redo collision calculation.
AddCircleCollider (float)	Creates a new CircleCollider and adds it to the Colliders component. Its name will be 'new circle collider' plus some number (recommend you rename).
AddCircleCollider {float, string)	Same as above, but specifies a name for the new CircleCollider.
AddSegment(Vector2, Vector2)	Adds a new SegmentCollider to the Colliders component given two points. Its name will be 'new segment collider' plus some number (rename me!).

AddSegment(Vector2, Vector2, string)	Same as above, but specifies a name for the new SegmentCollider.
GetCollider(string)	Gets the collider with the given name.
RemoveCollider(coll)	Removes the given collider (probably gotten from **GetCollider**).
IsColliding()	Whether or not the Colliders is currently colliding with something else.
IsCollidingWith(string)	Whether or not the Colliders is currently colliding with another GameObject that has the given name.
GetCurrentCollider()	Gets the currently colliding collider, or the last collider to have collided with something. Returns None if this component has never collided with anything.
SetDebugDraw(bool)	Turns debug draw on or off.

ICollider Properties

Encompasses all types of colliders (Circle, Segment).

.gameObject	The most useful property. Gets the GameObject this ICollider is attached to.
.name	The name of the collider. You should probably set this if you didn’t set it at creation time. Read/Write.
.resolution	The Resolution object attached to the collider. You can get this to set the resolution function of the given collider. See below for more details. Read only.
.transform	The Transform associated with the collider. Will be the same as the Colliders component’s GameObject.
.collisionPoint	The last point of collision this collider encountered.

CircleCollider Properties

.radius	The radius of the circle. Read/Write.
.offset	The offset of the circle if the GameObject’s Z-rotation were 0.0. Read/Write.

SegmentCollider Properties

.p0	The first point (Vector2) of the segment. Read only.
.p1	The second point (Vector2) of the segment. Read only.
SetPoints(Vector2, Vector2)	Sets the two points of the segments.

Resolution

This class wraps the function that is called when a specific ICollider collides with something. By default, it uses a Physics-based collision resolution, which only does something if a Physics component is involved with the collision. Calling **Reset** on the Resolution object wipes this default behavior.

Enum : ResolutionType

There are three categories of resolution function types:

- **ResolutionType.None** : No resolution will occur on this collider’s end.
- **ResolutionType.Function** : A C++ function handles resolution.
- **ResolutionType.Python** : A Python function handles resolution.

Public Members

.type	The ResolutionType (see above) of the function.
SetFunction(func)	Sets a Python function to be the resolution of the collider this class is attached to.
Reset()	Resets the ResolutionType to be None.

Colliders Practical Examples

Example 7.1 : Changing ICollider Physics Resolution

```python
# EG. 7.1 - Changing ICollider Physics Resolution
from PyroCore import *

# The resolution function must take three arguments: two GameObjects and dt
def MyResolution(first, second, dt):
  Debug.LogUser("%s is colliding with %s!" % (first.name, second.name))

class MyFactory(IFactory):
  def Create(system, name):
    obj = system.CreateGameObject(name)
    coll = obj.AddComponent(Colliders)
    circle = coll.AddCircleCollider(5.0, "my circle")
    circle.resolution.SetFunction(MyResolution) # <-- the magic
    return obj
```

Example 7.2 : OnCollisionEnter / OnCollisionExit

```python
#EG. 7.2 - OnCollisionEnter / OnCollisionExit
from PyroCore import *

def CollisionReporter(IScript):
  # 'this' refers to this IScript behaviour, 'thiscoll' refers to the current
  # ICollider (CircleCollider, SegmentCollider) attached to this GameObject
  # that is involved in the collision, and 'thatcoll' refers to the other
  # ICollider.
  def OnCollisionEnter(this, thiscoll, thatcoll):
    Debug.LogSuccess("%s is colliding with %s!" % (this.gameObject.name,
      thatcoll.gameObject.name))

  def OnCollisionExit(this, thiscoll, thatcoll):
    Debug.LogSuccess("%s is no longer colliding with %s!" % (this.gameObject.name,
      thatcoll.gameObject.name))
```

Physics

Roughly equivalent to Unity’s Rigidbody2D, the Physics component handles physics-based movement and collision resolution of a GameObject.

Inherited from IComponent

ToString()	Returns the name of the component as a string.
CloneInto(GameObject)	Clones this component into another GameObject.
Suspend(bool)	Pauses physics calculations.
IsSuspended()	Whether or not the component is suspended.
Enable(bool)	Does nothing (for now).
IsEnabled()	Whether or not the thing is enabled.
UID()	The unique ID of the parent GameObject.
GetGameObject()	The GameObject this is attached to.
.uid	The unique ID of the parent GameObject. Read only.
.gameObject	The GameObject this is attached to. Read only.

Public Properties

.gravity	(Vector2) The current gravity being applied to the GameObject. Read/Write.
.velocity	(Vector2) The current velocity of the GameObject. Read/Write.
.acceleration	(Vector2) The current acceleration of the GameObject. Read/Write.
.force	(Vector2) The current force being applied to the GameObject. Read/Write.
.normalForce	(Vector2) The current normal force of the GameObject. Read/Write.
.restitution	(float) The restitution value of the GameObject. Read/Write.
.friction	(float) The friction modifier of the GameObject. Read/Write.
.energy	(float) The total energy of the fucking GameObject. Read/Write.
.kinematicEnergy	(float) The kinematic energy of the GameObject. Read/Write.
.potentialEnergy	(float) The potential energy of the GameObject. Read/Write.
.barycenter	(Vector2) The geometric barycenter of the GameObject. Read only.
.drag	(float) The drag applied to the GameObject.

Public Methods

Basically, getters and setters for all the above properties. For instance, **GetGravity** and **SetGravity** correspond with the **.gravity** property. Whether you use properties or methods does not matter. The only addendum is that **.barycenter** cannot be set, and therefore has no setter.

Example 8 : Physics-Based Movement

```
# EG. 8 : Physics-Based Movement
from PyroCore import *

class MyMovement(IScript):
  def Start(this):
    this.rb = this.gameObject.GetComponent(Physics)
    # using 'rb' to stand for Rigidbody (as you would in Unity)
    this.rb.gravity = Vector2(0, 0) # turn off gravity
    this.rb.drag = 0.0              # turn off drag

  def Update(this, dt):
    x = Input.GetAxis(Axis.LHorizontal) # more on Input in 'Static-ish Classes'
    y = Input.GetAxis(Axis.LVertical)

    if x != 0.0:
      this.rb.velocity += x * dt * 20 * Vector2(1, 0) # 20 is a magic number
    if y != 0.0:
      this.rb.velocity += y * dt * 20 * Vector2(0, 1)
    # and that’s it! A simple physics-based player controller script.
```

Sprite

The Sprite component encompasses all there is to know about a GameObject’s appearance. Please note that while this API is up-to-date, Andrey plans to vastly refactor the Sprite component and generally the entire way graphics work. With that in mind, just know that this particular section of the API is definitely going to change in the future.

Inherited from IComponent

ToString()	Returns the name of the component as a string.
CloneInto(GameObject)	Clones this component into another GameObject.
Suspend(bool)	Pauses animations.
IsSuspended()	Whether or not the component is suspended.
Enable(bool)	Does nothing (for now).
IsEnabled()	Whether or not the thing is enabled.
UID()	The unique ID of the parent GameObject.
GetGameObject()	The GameObject this is attached to.
.uid	The unique ID of the parent GameObject. Read only.
.gameObject	The GameObject this is attached to. Read only.

Public Methods

Some available methods are omitted here, either due to its likelihood of being refactored out or due to the overly-complex nature of using the method.

SetMesh(Mesh)	Sets the Sprite’s Mesh object, usually gotten from the RenderSystem or another Sprite.
SetMesh(string)	Sets the Sprite’s Mesh object, searching for a mesh with the given name.
GetMesh()	Gets the Sprite’s Mesh object.
HasMesh()	Whether or not the Sprite even has a Mesh.

GetTexture()	Gets the Texture object associated with the Sprite’s Mesh, if there is one. If there is not, None is returned and a warning is sent to the logs. See appropriate section on the Texture class for further usage.
GetAnimationState()	Gets the AnimationState object associated with the Sprite’s Mesh, if there is one. If there is not, None is returned and a warning is sent to the logs. See appropriate section on the AnimationState class for further usage.
GetSkeleton()	Gets the Skeleton object associate with the Sprite’s Mesh, if there is one. If there is not, None is returned and a warning is sent to the logs. See appropriate section on the Skeleton class for further usage.
IsAnimated()	Whether or not the Sprite has an animation.
SetGraphic(string)	Sets the Sprite’s graphic to the given graphic ID. See below for more about graphic IDs.
GetGraphicID()	Gets the Sprite’s current graphic ID.
ScaleToTexture()	Scales the Sprite’s GameObject to fit the aspect ratio of the associated Texture’s image file.
SetAnimation(string)	Sets the current animation by name. Defaults to track #0 and looping to true. Throws a warning to the logs if no AnimationStateData exists for the Sprite.
SetAnimation(string, bool)	Sets the current animation by name, and sets whether or not the animation should loop. Defaults to track #0.
SetAnimation(string, bool, int)	Sets the current animation be name, sets whether or not the animation should loop, and sets the track number of the animation.
SetAtlasRegion(string)	If the Sprite has an Atlas, sets the current region by name. If it does not, a warning is sent to the logs.
GetColor()	Returns a Color object representing the Sprite’s tint. By default, Sprites have a pure white tint (or Color(1.0, 1.0, 1.0, 1.0)).
SetColor(Color)	Sets the Sprite’s colour tint.

GetAlpha()	Gets only the Sprite's alpha value. Shorthand for **GetColor().a**.
SetAlpha(float)	Sets only the Sprite's alpha value. Values between 0.0 and 1.0 are considered valid.
GetUVTranslation()	(Vector2) Gets the UV translation of the Sprite's Texture.
SetUVTranslation (Vector2)	Sets the UV translation of the Sprite's Texture. Useful for creating scrolling parallaxes.
GetUVScale()	(Vector2) Gets the UV scale of the Sprite's Texture.
SetUVScale(Vector2)	Sets the UV scale of the Sprite's Texture.
SetUVTransform (translation, scale)	Sets both the UV translation and UV scale all in one. Takes two Vector2s.
GetLayer()	Gets the Sprite's layer. (Deprecated? See **Mesh.GetDrawPriority()**)
SetLayer(float)	Sets the Sprite's layer. (Deprecated? See **Mesh.SetDrawPriority(int)**)

Example 9.1 : Instantiating a 'Pure' Texture

```
# EG. 9.1 : Instantiating a Pure Texture
from PyroCore import *

class PureTextureFactory(IFactory):
  def Create(system, name):
    obj = system.CreateGameObject(name)
    sprite = obj.AddComponent(Sprite)
    sprite.SetGraphic("debug256") # <-- Sets Texture to debug256.png,
                                  #     and Mesh to the default square mesh.
    sprite.ScaleToTexture()       # <-- Not necessary since debug256.png is 1:1,
                                  #     but if it weren't this would be useful.
    return obj
```

This may be noted elsewhere, but because of the way graphic IDs work, it is important that there isn't more than one .png file in the assets file structure named 'debug256'; if there is, there will be ambiguity as to which one will be used.

Example 9.2 : Instantiating an Atlased Texture

```
# EG. 9.2 : Instantiating an Atlased Texture
from PyroCore import *

class AtlasedTextureFactory(IFactory):
  def Create(system, name):
    obj = system.CreateGameObject(name)
    sprite = obj.AddComponent(Sprite)
    sprite.SetGraphic("test_tileset_v1_001")
    # The graphic ID is the same as the .atlas and .png filenames,
    # minus the extension.
    sprite.ScaleToTexture()
    sprite.SetAtlasRegion("images/center_tile")
    # The best way to find Atlas region names is through the Asset Selector.
    # Alternatively, you can ask the artist or look through the .atlas file.
    return obj
```

Example 9.3 : Instantiating an Animation (Spine)

```
# EG. 9.3 : Instantiating an Animation (Spine)
from PyroCore import *

class AnimatedSpriteFactory(IFactory):
  def Create(system, name):
    obj = system.CreateGameObject(name)
    sprite = obj.AddComponent(Sprite)
    sprite.SetGraphic("kora") # Parsed from the file 'kora.anim'
    sprite.GetAnimationState().SetAnimation("swim", True)
    # See Asset Selector for valid animation names.

    # Bonus -- how to set skins through the Skeleton as well as draw priority
    sprite.GetSkeleton().SetSkin("scythe_mutation") # See Asset Selector for
                                                    # valid skin names.
    sprite.GetMesh().SetDrawPriority(10)
    return obj
```

Static-ish Classes

There are several static classes that have been made available to you.

- **Engine** : not really static, but has the static property **.main** that you can use to get the currently running Engine object from any function. It is essentially a singleton, meaning it walks and talks like a static class, though it is technically not.
- **Time** : contains most of the same useful functionality as Unity’s Time static class.
- **Input** : again, attempts to stay true to Unity’s API. Provides an interface to detect input.
- **Debug** : provides logging functions.
- Others that aren’t necessarily static but contain some static methods will be detailed in the ‘Math Classes’ or ‘Other Classes’ sections.

Engine

The Engine object is everything. It contains all systems, all scenes, the console window, and the game window.

Static Properties

.main	Gives you access to the main running Engine instance.
.frameCount	Shorthand for **engine.TimeSystem().frameCount** or an alternative to **Time.frameCount**. Why is this here? IDFK.

Public Methods

Kill(int)	Kills the Engine and therefore the game. Pass an exit code (in most cases, 0, unless an error occurred).
RenderSystem()	Gets the RenderSystem from the Engine instance.
SceneSystem()	Gets the SceneSystem from the Engine instance.
TimeSystem()	Gets the TimeSystem from the Engine instance. You are probably better off using the **Time** static class instead.
EngineEventSystem()	Gets the EngineEventSystem from the Engine instance. You are probably better of using the **Input** static class instead.
WorldGameObjects()	Gets the world GameObjectSystem of the currently running scene.
SetVerticalSync (bool)	Toggles Vertical Sync of the game window. Probably best to only call this in the .init script.
SetFramerateLimit (int)	Sets the framerate limit of the game window. Probably best to only call this in the .init script.
SetMaximiseOnStart (bool)	Toggles whether or not the game window should start maximised. Probably best to only call this in the .init script.
IsFullscreen()	Whether or not the window is currently fullscreen.
SetFullscreen(bool)	Sets whether or not the window is fullscreen.
GetAssetsPath()	Gets the path to the assets directory.

Time

Static Properties

.time	The current runtime of the game, in seconds. Read only.
.deltaTime	The time between now and the previous frame. Read only.
.realDeltaTime	The real time between now and the previous frame. That is, if there is a non-1.0 Delta Time Multiplier, this will return a value as if the multiplier was not applied. Read only.
.deltaTimeMultiplier	The multiplier applied to deltaTime. Read/Write.
.fixedDeltaTime	The time between the last fixed frame and the fixed frame before it. Read only.
.frameCount	The number of frames that have been executed since startup. VERY useful for optimisations. Read only.
.fixedFrameCount	The number of fixed frames since startup. Read only.
.frameRate	The framerate of the game Engine. Read only.
.fixedFrameRate	The framerate of fixed frames within the game Engine. Read only.
.triggerFixedUpdate	Whether or not this frame is a fixed frame. Read only.

Static Methods

CreateTimer(float)	Creates a Timer with the given start time. By default, the Timer starts immediately as if **Start** was called. See below for Timer class details.
PauseAllTimers(bool)	Sets whether or not to pause the updating of all Timer instances.
SetDeltaTimeMultiplier (float)	Sets the delta time multiplier. For instance, setting it to 2.0 would mean that things like animations run at double speed.
GetDeltaTimeMultiplier ()	Gets the delta time multiplier.
TimerInfo()	Returns a formatted string containing a summary table of all Timers in the system.

ToString()	Returns a formatted string containing a summary table of all properties in the system.

The Timer Class

Timers are extremely useful, especially if set as an Attribute of a GameObject (because you can view, change, and pause/play them in the editor). A common use case is if an enemy can attack only every few seconds, or if they spawn at a given interval. See the end of this section for a concrete example of this.

Static Methods

Create(float)	Creates a new Timer. By default, the Timer starts immediately as if **Start** was called. Equivalent to **Time.CreateTimer**.

Public Properties

.time	The maximum time the Timer starts at and resets to when **Reset** is called. Represented as a float in seconds. Read/Write.
.currentTime	The current time of the Timer. Can go negative if the Timer is not reset. Represented as a float in seconds. Read/Write.
.isDone	Whether or not the Timer is done (**.currentTime** <= 0.0).
.isPaused	Whether or not the Timer is in the 'paused' state. Read only (call **Pause** to pause).

Public Methods

Done()	Whether or not the Timer is done (**.currentTime** <= 0.0).
Reset()	Resets the current time to the maximum time.
Set(float)	Sets the maximum time.
SetCurrent(float)	Sets the current time.
Pause()	Pauses the Timer.
Start()	Starts the Timer if it is currently paused.
Time()	Returns the maximum time.
CurrentTime()	Returns the current time.

Example 10 : Optimising with the Time Class

```python
# EG. 10 : Optimising with the Time Class
from PyroCore import *

class MyEnemyBehaviour(IScript):
  def PathToPlayer(this, dt):
    # Pretend that this method actually path finds,
    # which is an expensive operation if we were to do it every frame.
    pass

  def Move(this, dt):
    # Pretend that this method applies movement to the enemy,
    # which we don't have to do every frame if it is Physics-based.
    pass

  def Update(this, dt):
    if Time.fixedFrameCount % 15 == 0: # True about every half second.
      this.PathToPlayer(dt)            # Avoids being called every frame.
    elif Time.triggerFixedUpdate:      # True about every 1/30th of a second.
      this.Move(dt)                    # Again, isn’t (usually) called every frame.
```

Example 11 : Utilising the Timer Class (EnemyController)

```
# EG. 11 : Utilising the Timer Class
from PyroCore import *

class EnemyController(IScript):
  def Start(this):
    # Create a 10-second Timer:
    this.spawnTimer = this.gameObject.CreateAttribute("spawnTimer",
      Timer.Create(10.0)) # Indent to multiline parentheses statements

    # Bonus : Example of using GameObjectLists!
    this.enemies = \ # Alternatively, use a backslash to multiline
      this.gameObject.CreateAttribute("enemies", GameObjectList.Create())

  def SpawnPredator(this):
    # Example of using a factory script from within another script:
    predator = PredatorFishFactory.Create(this.gameObject.system, "predator")
    # Do more setup here...
    # Do more setup here...
    # Do more setup here...
    this.enemies.append(predator)
    return predator

  def Update(this, dt):
    if this.spawnTimer.isDone:
      this.SpawnPredator()
      this.spawnTimer.Reset()
    # Bonus : Example of iterating through GameObjectLists!
    for enemy in this.enemies:
      if enemy.GetAttribute("isDead"):
        this.enemies.remove(enemy)
```

Input

This static class *attempts* to mime Unity's own **Input** static class. The primary difference is our use of Python enums (don't google that, because natively enums don't exist in Python; boost essentially added a new language feature that we utilise).

enum Axis

Values of this enum are:

- **Axis.LHorizontal** : The horizontal axis of the left joystick (or WASD).
- **Axis.LVertical** : The vertical axis of the left joystick (or WASD).
- **Axis.RHorizontal** : The horizontal axis of the right joystick (no keyboard support).
- **Axis.RVertical** : The vertical axis of the right joystick (no keyboard support).
- **Axis.LTrigger** : The left trigger (no keyboard support).
- **Axis.RTrigger** : The right trigger (no keyboard support).

enum Button

- **Button.RBumper** : The right bumper (or currently keyboard 'R').
- **Button.LBumper** : The left bumper (or currently keyboard 'F').
- **Button.Up** : Up on the D-Pad (or currently keyboard 'W').
- **Button.Down** : Down on the D-Pad (or currently keyboard 'S').
- **Button.Left** : Left on the D-Pad (or currently keyboard 'A').
- **Button.Right** : Right on the D-Pad (or currently keyboard 'D').
- **Button.A** : The A button on the gamepad (or keyboard SPACE).
- **Button.B** : The B button on the gamepad (or mouse right-click).
- **Button.X** : The X button on the gamepad (or mouse left-click).
- **Button.Y** : The Y button on the gamepad (or mouse middle-click).
- **Button.Pause** : The Pause button on the gamepad (or keyboard 'P').
- **Button.Back** : The Back button on the gamepad (or keyboard BACKSPACE).
- **Button.LClick** : The left joystick click on the gamepad (or mouse left-click).

- **Button.RClick** : The right joystick click on the gamepad (or mouse right-click).
- **Button.MClick** : Only the middle-click on the mouse.
- **Button.F1**
    - Through
- **Button.F12** : Only the keyboard F-keys.
- **Button.Escape** : Only the ESCAPE key on the keyboard.
- **Button.Jump** : The A button on the gamepad (or keyboard SPACE).
- **Button.Fire1** : The right bumper (or currently keyboard 'R').
- **Button.Fire2** : The left bumper (or currently keyboard 'F').

Static Properties

.anyKeyDown	Is *any* key currently pressed?
.anyJoystickConnected	Is *any* joystick connected?

Static Methods

GetAxis(Axis)	Gets a float value read in from the given axis. Values range from -1.0 to 1.0 for joysticks, and 0.0 and 1.0 for triggers.
GetButton(Button)	Gets whether or not a Button is being held.
GetButton(string)	Gets whether or not a key is being held. Valid values are like 'W', 'A', 'S', 'D', 'F1', ' ', etc.
GetButtonDown (Button)	Same as above, but only returns True the first frame the button is pressed.
GetButtonDown (string)	Same as above, but takes a string representing a key.
GetButtonUp (Button)	Same as above, but only returns True the first frame that a button is released.
GetButtonUp (string)	Same as above, but takes a string representing a key.
IsEnabled()	Whether or not Input is enabled for use by Python. (C++ code is unaffected.)
Enable(bool)	Toggles Input access from Python scripts. C++ code is unaffected.

Example 12 : Getting Input

```
# EG. 12 : Getting Input
from PyroCore import *

class InputReporter(IScript):
  def Update(this, dt):
    if Input.GetButtonUp(Button.Escape):
      # Kill the engine:
      this.gameObject.system.engine.Kill(0)
    if Input.GetButtonDown(Button.Fire1):
      Debug.LogUser("Fire1 triggered!")
    if Input.GetButtonDown("R"):
      # Currently, this if statement is equivalent to the above.
      Debug.LogUser("\"R\" key triggered!")
      # Both messages will print when R is pressed, but only Fire1
      # will print if the gamepad RBumper is pressed. Can you tell me why?

    x = Input.GetAxis(Axis.LHorizontal) # left is -1.0, right is 1.0
    y = Input.GetAxis(Axis.LVertical)   # down is -1.0,    up is 1.0

    if x != 0.0 or y != 0.0:
      Debug.Log("Joystick state = (%.2f,%.2f)" % (x, y))
```

Debug

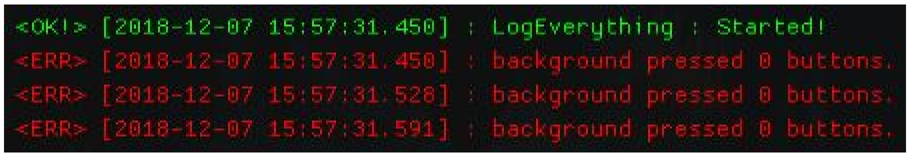
Static Methods

Log(string)	Prints a time-stamped string to the logs. Shows up as greyish-white.
LogSuccess(string)	Prints a time-stamped string to the logs. Shows up as bright green.
LogWarning(string)	Prints a time-stamped string to the logs. Shows up as bright yellow.
LogError(string)	Prints a time-stamped string to the logs. Shows up as bright red.
LogUser(string)	Prints a time-stamped string to the logs. Shows up as bright cyan.

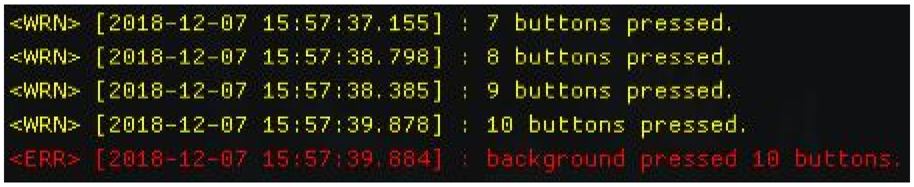
Example 13 : Logging Variables Embedded in Strings

```
# EG. 13 : Logging Variables Embedded in Strings
from PyroCore import *

class LogEverything(IScript):
  def Start(this):
    Debug.LogSuccess("LogEverything : %s" % "Started!”)
    # %s is replaced with 'Started!'
    this.buttonCount = 0

  def Update(this, dt):
    if Input.anyKeyDown:
      this.buttonCount += 1
      Debug.LogWarning("%d buttons pressed." % this.buttonCount)
      # Use the % operator with a string to replace %d with integers,
      # %f with floats, %s with strings, and %r with booleans.
    if this.buttonCount % 10 == 0:
      Debug.LogError("%s pressed %d buttons." % (this.gameObject.name,
        this.buttonCount))
      # Use what’s called a tuple (thing1, thing2) with the % operator if there are
      # multiple % symbols in the string.
```

Attaching to the GameObject named ‘background’ gives this output:



```
<OK!> [2018-12-07 15:57:31.450] : LogEverything : Started!
<ERR> [2018-12-07 15:57:31.450] : background pressed 0 buttons.
<ERR> [2018-12-07 15:57:31.528] : background pressed 0 buttons.
<ERR> [2018-12-07 15:57:31.591] : background pressed 0 buttons.
```



```
<WRN> [2018-12-07 15:57:37.155] : 7 buttons pressed.
<WRN> [2018-12-07 15:57:38.798] : 8 buttons pressed.
<WRN> [2018-12-07 15:57:38.385] : 9 buttons pressed.
<WRN> [2018-12-07 15:57:39.878] : 10 buttons pressed.
<ERR> [2018-12-07 15:57:39.884] : background pressed 10 buttons.
```

System Classes

The following systems (inheritors of ISystem) are available to you through Python:

- **EngineEventSystem** : has exactly the same interface as the static class **Input**. Use that instead. This system's specifications will be omitted here.
- **TimeSystem** : has exactly the same interface as the static class **Time**. Use that instead. This system's specifications will be omitted here.
- **RenderSystem** : used to get and create Meshes, Atlases, SkeletonDatas, and AnimationStateDatas. You do *not* currently need to worry about the last three functionalities, so they will be omitted here.
- **SceneSystem** : contains all the Scenes loaded into the Engine. Each one has its own GameObjectSystem and respective ComponentSystems (the latter is not bound to Python for various reasons).
- **GameObjectSystem** : one of these per Scene. Later implementations may include two of these per Scene -- one for world GameObjects, and the other for UI GameObjects.

Inherited from ISystem

All systems inherit the following from ISystem:

.engine	The Engine instance this system belongs to. Read only.
GetEngine()	Gets the Engine instance this system belongs to.
IsAwake()	Whether or not **Awake** has been called on the system yet.
IsSuspended()	Whether or not the system is considered suspended.
Suspend(bool)	Mark the system for suspension.

RenderSystem

Note that this class is up for refactoring. I will still document its current state here, but take heed: all of this is bound to change.
How to get the currently running RenderSystem from an IScript:

```
rs = this.gameObject.system.engine.RenderSystem()
```

Alternatively, from *anywhere*:

```
rs = Engine.main.RenderSystem()
```

Public Methods

GetDebugSquareMesh()	Gets a wireframe square mesh.
GetDebugCircleMesh()	Gets a wireframe circle mesh.
GetSquareMesh()	Gets the standard square mesh. Might want to clone it.
GetMesh(string)	Gets a mesh with the specified name. Not likely that you should use this.
CloneMesh(string, string)	Clones a mesh. The first string is the new mesh's name, the second string is the old mesh's name.
CloneMesh(string, Mesh)	Clones a mesh, giving it the name that is the first string.
GetAtlas(string)	Gets an atlas with the given graphic ID. Not likely you will need this.
GetSkeletonData(string)	Gets the SkeletonData from the given graphic ID.
GetAnimationStateData (string)	Gets the AnimationStateData with the given graphic ID.
GetMainCamera()	Gets the main Camera. See section 'Graphics Classes' for more information.
SetMainCamera(Camera)	Sets the main Camera. Probably no use for this in Python (yet).

SceneSystem

A simple system, this manages the current Scene and gives functionality for switching to other Scenes.

Public Methods

GetCurrentScene()	Gets the currently active Scene.
GotoScene(string)	Goes to the Scene with the given name. EG. **Engine.main.SceneSystem().GotoScene("world0")**

GameObjectSystem

A system that manages all GameObjects that it contains.
Worth noting that this is the type of system that is passed to IFactory **Create** methods:

```
class MyFactory(IFactory):
  def Create(system, name):
    pass
```

Public Properties

.currentWorld	Static. Gets the GameObjectSystem of the current world Scene. Call like **GameObjectSystem.currentWorld.**
.scene	Gets the Scene that this GameObjectSystem belongs to.

Public Methods

.CreateGameObject (string)	Creates and returns a new GameObject with the given name.
GetGameObject(ID)	Gets the GameObject with the given UID. Not really that useful in Python...
FindWithName(string)	Finds the first GameObject with the given name. Very useful.

Math Classes

Here you will find the following classes:

- **Transform** : not inherently a math class, but this is the best place to put it, as you will be referencing the rest of this section whenever you deal with Transforms.
- **Mathf** : an ever-growing float math library for you to use. Additional functionality will be added on an as-needed basis.
- **Random** : a random number generator. What more do you need to know?
- **Color** : not necessarily a math structure, but it is essentially a Vector4 and thus has **a lot** of math operations bound to it. Not entirely sure if this belongs more in the 'Graphics Classes' section, but here you go.
- **Vector4** : a four-dimensional vector. Only used much in terms of converting to-and-from Colors, and if you ever do any advanced matrix manipulation.
- **Vector3** : a three-dimensional vector, used mostly in terms of Transform scales and translations. Conversion between this type and Vector2 is common.
- **Vector2** : a two-dimensional vector, used mostly by Physics and Colliders components. Conversion between this type and Vector3 is common.
- **IVector2** : not going to be detailed here, as it is simply a Vector2 with integer components. Mainly gotten from things like Texture size.
- **Quaternion** : essentially a Vector4, but with slightly different math semantics. Used solely for Transform rotations.
- **Matrix4** : a 4x4 square matrix. Used with advanced Transform and Camera manipulation.

Transform

If you are familiar with Unity, then you know Transforms are highly important; if you can’t make shit move around on screen, then it’s hardly a game.

Public Properties

.globalPosition	Gets the global position vector of the Transform. Currently, this is read only (can’t set it). See **AddGlobalPos** if you want to set this member.
.globalScale	Gets the global scale vector of the Transform. Read/Write.
.position	Gets the local position of the Transform. Read/Write.
.scale	Gets the local scale of the Transform. Read/Write.
.rotation	Gets the local rotation of the Transform **as a Quaternion**. Read/Write.
.rotationF	Gets the local rotation of the Transform **as a float** (in the Z-axis). Read/Write.
.parent	Gets the parent Transform. Returns None if there is none.

Public Methods

AddLocalPos(float, float, float)	Adds to the x, y, and z components, respectively, of the Transform’s local position.
AddGlobalPos(float, float, float)	Adds to the x, y, and z components, respectively, of the Transform’s global position.
AddLocalRot(float, Vector3)	Adds the angle (in radians) to the Transform’s rotation in the given axis. (You probably don’t want to use this one... we are in 2D.)
AddLocalRot(float)	Adds the angle (in radians) to the Transform’s rotation in the Z axis. **Use me**.
GetOffsetFromParent()	Gets the offset from the parent Transform. Returns Vector3(0, 0, 0) if there is no parent.
RecalculateMatrix()	Recalculates the Transform’s matrix. Only use this if you know what you are doing.

Example 14.1 : PlayerController with Transforms

```
# EG. 14.1 : PlayerController with Transforms
from PyroCore import *

class TransformController(IScript):
  def Update(this, dt):
    x = Input.GetAxis(Axis.LHorizontal)
    y = Input.GetAxis(Axis.LVertical)

    # do movement
    if x != 0.0:
      this.transform.position += x * 10 * dt # 10 is arbitrary
    if y != 0.0:
      this.transform.position += y * 10 * dt

    # do rotation
    angle = Mathf.Atan2(y, x) - Mathf.PI # More on this in 'Mathf' section
    this.transform.rotation = Quaternion.Euler(Vector3(0, 0, angle))
```

Example 14.2 : Enemy Circling with Transforms

```
# EG. 14.2 : Enemy Circling with Transforms
from PyroCore import *

class EnemyMovement(IScript):
  def Start(this):
    # Probably have to flip this enemy's graphic:
    this.gameObject.GetComponent(Sprite).GetSkeleton().flipX = True
    # Set a target:
    this.target = this.gameObject.system.FindWithName("player")
    # create a speed Attribute:
    this.gameObject.CreateAttribute("speed", 10.0)
  def Update(this, dt):
    toTarget = this.target.transform.position - this.transform.position
    toTarget.Normalize() # More on this in 'Vector3' section
    toTarget *= dt
    normal = Vector3(toTarget.y, -1 * toTarget.x, 0)
    # the above is a vector maths trick; it creates a perpendicular vector
    normal *= this.gameObject.GetAttribute("speed")
    this.transform.position += normal

    angle = Mathf.Atan2(normal.y, normal.x)
    this.transform.rotation = Quaternion.Euler(Vector3(0, 0, angle))
```

Mathf

Static Properties

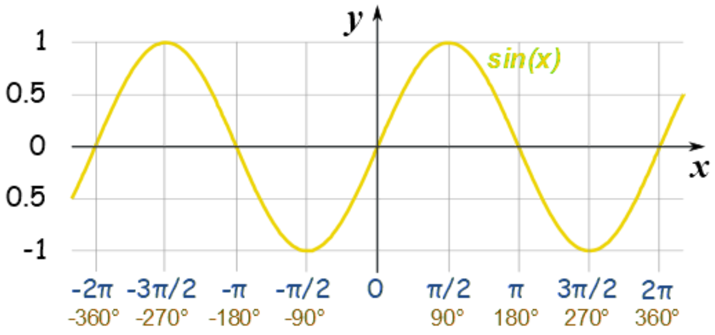
.PI	It’s pi. 3.14159265358979323846
.Epsilon	The smallest floating point value available. Tiny.
.Euler	Euler’s constant.
.Rad2Deg	The constant you multiply by to convert radians to degrees.
.Deg2Rad	The constant you multiply by to convert degrees to radians.

Static Methods

Clamp(val, min, max)	Clamps ‘val’ between ‘min’ and ‘max’, inclusive.
Atan2(y, x)	Returns the arctangent-squared of the given y and x values. Useful for calculating the angle of a given 2D vector (see examples 14.1 and 14.2 in the previous section).
Sin(rads)	Returns the sine of the given angle in radians. See below for more quick info on sine waves.
Cos(rads)	Returns the cosine of the given angle in radians. See below for more quick info on cosine waves.
Sign(val)	Returns -1 if ‘val’ is negative, +1 if it is positive, and 0 if it is 0. Useful for checking for facing directions.
Pow(base, exp)	Returns ‘base’ to the power of ‘exp’. EG. **Mathf.Pow**(2.0, 2.0) == 4.0
Log(of, base)	Returns the logarithm of ‘of’ with base ‘base’. EG. **Mathf.Log**(4.0, 2.0) == 2.0
Ln(of)	Returns the natural logarithm of ‘of’.
Abs(val)	Returns the absolute value of ‘val’.
Max(val1, val2)	Returns the higher of two values.
Min(val1, val2)	Returns the lower of two values.
Oscillate(min, max, t)	Oscillates between two values given some time-related value ‘t’.

More on Sine / Cosine (Mathf.Sin / Mathf.Cos)

Sine and cosine are very helpful continuous mathematics functions. They can be used to easily generate numbers that oscillate between two values, a min and a max. The basic graph for how sine looks shows at right.



Cosine is very similar to sine, but differs in that at x = 0, y = 1.

To oscillate between two values *a* and *b*, use the following formula:

val = (sin(x) + 1) / 2 * (b - a) + a

or in Python:

```
a = 4.0
b = 10.0
val = (Mathf.Sin(Time.time) + 1) / 2 * (b - a) + a
# val will oscillate between 4 and 10
```

NEW : Mathf.Oscillate does the above for you, and is much faster.
You can pass Time.time, or some other value calculated from dt.

Example 15 : Floating Effect with Sine

Below you will find a very simple example of how to use Mathf.Sin to achieve a floating effect.

```
# EG. 15 : Floating Effect with Sine
from PyroCore import *

class FloatingEffect(IScript):
  def Start(this):
    this.gameObject.CreateAttribute("floatness", 0.5)
  def Update(this, dt):
    amount = Mathf.Sin(Time.time) * this.gameObject.GetAttribute("floatness")
    # amount will oscillate between -0.5 and 0.5, or whatever 'floatness' is
    this.transform.AddLocalPos(0, amount * dt, 0)
    # multiply amount by dt to normalise it between frames
```

Random

Static Methods

Rand()	Returns a random value between 0.0 and 1.0.
Range(min, max)	Returns a random value between 'min' and 'max', inclusive.
Chance(percent)	Returns True the given 'percent' of the time (between 0.0 and 1.0). EG. **Random.Chance**(0.5) returns True 50% of the time.

Example 16 : Random Spawning

One of the most effective uses of the Random static class is for randomising spawn logic. Below, you will find an example that utilises the Random class to both select a random enemy *and* generate a random position to spawn it at.

```
# EG. 16 : Random Spawning
from PyroCore import *

class RandomFactory(IFactory):
  isInit = False
  fishFactory = None
  def Init(engine):
    if Random.Chance(0.5):
      # 50% chance to spawn a PredatorFish
      RandomFactory.fishFactory = PredatorFishFactory
    else:
      # 50% chance to spawn a ScytheFish
      RandomFactory.fishFactory = ScytheFishFactory
    # intentionally do not set isInit to True

  def Create(system, name):
    if RandomFactory.fishFactory is not None:
      fishy = RandomFactory.fishFactory.Create(system, name)
      # use Random.Range to randomise a position:
      pos = Vector3(Random.Range(-10, 10), Random.Range(-10, 10), 0)
      fishy.transform.position = pos
      return fishy
```

Color

This is one of the few classes you will instantiate inside PyroCore Python. It is implicitly convertible to and from a Vector4, which you may notice in some error log messages (if you fuck up, God forbid).

Constructors

Color()	Constructs the default White (1, 1, 1, 1) Color.
Color(float)	Constructs a Color with all values equal to the given float. EG. **Color**(0.5) == **Color**(0.5, 0.5, 0.5, 0.5)
Color(float, float, float)	Constructs a Color with the given red/green/blue values, defaulting the alpha to 1.0. EG. **Color**(0.6, 0.2, 0.2) == **Color**(0.6, 0.2, 0.2, 1.0)
Color(float, float, float, float)	Constructs a Color with the given red/green/blue/alpha values.
Color(Vector4)	Converts a Vector4 to a Color explicitly.
Color(Vector3)	Converts a Vector3 to a Color, defaulting the alpha to 1.0.
Color(Vector3, float)	Converts a Vector3 to a Color with the given alpha.

Static Properties

.White	Equal to **Color**(1, 1, 1, 1), but faster in calculations.
.Red	Equal to **Color**(1, 0, 0, 1), but faster in calculations.
.Green	Equal to **Color**(0, 1, 0, 1), but faster in calculations.
.Blue	Equal to **Color**(0, 0, 1, 1), but faster in calculations.
.Black	Equal to **Color**(0, 0, 0, 1), but faster in calculations.

Static Methods

Lerp(Color, Color, t)	Linearly interpolates between two Colors. EG, when t = 0, **Color.Lerp** returns the first Color. When t = 1, **Color.Lerp** returns the second Color. Anything in between is a linear interpolation between the two Colors.

Public Properties

.r	The red value of the Color.
.g	The green value of the Color.
.b	The blue value of the Color.
.a	The alpha value of the Color.

Public Methods

Lerp(Color, t)	The same as the static method described above, except called as a method on a particular Color.
	EG. color1 = Color(0.5)
	color2 = Color(1.0)
	color3 = color1.**Lerp**(color2, dt)

Operators

The Color class inherits all operators from Vector4. See the next section on Vector4 objects for the operators available to them.

Example 17.1 : Oscillating Colors

```
# EG. 17.1 : Oscillating Colors
from PyroCore import *

class ColourOscillator(IScript):
  toColour = Color(0.5, 1.0, 0.2, 1.0)

  def Start(this):
    this.origColour = this.gameObject.GetComponent(Sprite).GetColor()

  def Update(this, dt):
    colour = Color()
    colour.r = Mathf.Oscillate(toColour.r, this.origColour.r, Time.time)
    colour.g = Mathf.Oscillate(toColour.g, this.origColour.g, Time.time)
    colour.b = Mathf.Oscillate(toColour.b, this.origColour.b, Time.time)
    this.gameObject.GetComponent(Sprite).SetColor(colour)
```

Example 17.2 : Funky Colouring

```
# EG. 17.2 : Funky Colouring
from PyroCore import *

class FunkyColouring(IScript):
  def Start(this):
    this.sprite = this.gameObject.GetComponent(Sprite)

  def Update(this, dt):
    rot = this.transform.rotation # Returns a Quaternion
    rotVec = Vector3(rot.x, rot.y, rot.z).Normalize()
    color = Color(rotVec, 1.0)  # Generate a colour from rotation,
                                # give it 1.0 alpha.
    this.sprite.SetColor(color) # Generally, generates blue hues.
```

Vector4

If you're dealing with Vector4 objects, then you are probably actually working with Colors. A lot of methods in the PyroCore library return Vector4 objects to represent colours, and so the conversion between the two has been made implicit for you. Therefore, anything a Color can do, a Vector4 can also do, and vice versa.

Constructors

Vector4()	Constructs an empty Vector4, with all values initialised to 0.0. This differs from the default **Color()** constructor.
Vector4(Vector3, float)	Constructs a Vector4 from a Vector3, with the additional w component specified.
Vector4(float)	Constructs a Vector4 with all components initialised to the given value.
Vector4(float, float, float, float)	Constructs a Vector4 with the given x, y, z, and w values.

Static Methods

Lerp(Vector4, Vector4, t)	Linearly interpolates between two Vector4s. EG, when t = 0, **Vector4.Lerp** returns the first Vector4. When t = 1, **Vector4.Lerp** returns the second Vector4. Anything in between is a linear interpolation between the two Vector4s.

Public Properties

.x	The x component of the Vector4.
.y	The y component of the Vector4.
.z	The z component of the Vector4.
.w	The w component of the Vector4.
.r	Equivalent to (points to) the x component.

.g	Equivalent to (points to) the y component.
.b	Equivalent to (points to) the z component.
.a	Equivalent to (points to) the w component.

Public Methods

Lerp(Vector4, t)	The same as the static method described above, except called as a method on a particular Vector4.
	EG. vec1 = Vector4(0.5)
	vec2 = Vector4(1.0)
	vec3 = vec1.**Lerp**(vec2, dt)

Operators

Vector4 + Vector4 **Vector4 + Color** **Color + Vector4** **Color + Color**	Adds two Vector4s together component-wise, and returns a new Vector4. EG. Vector4(1) + Vector4(2) == Vector4(3, 3, 3, 3)
Vector4 + float **Color + float**	Adds a float to each component in the Vector4, and returns a new Vector4.
Vector4 += Vector4 **Vector4 += Color** **Color += Vector4** **Color += Color**	Adds the second Vector4 to the first, modifying the values held in the first.
Vector4 += float **Color += float**	Adds a float to each component in the Vector4, modifying it.
Vector4 - Vector4 **Vector4 - Color** **Color - Vector4** **Color - Color**	Subtracts the second Vector4 from the first component-wise, and returns a new Vector4. EG. Vector4(4) - Vector4(1) == Vector4(3, 3, 3, 3)
Vector4 - float **Color - float**	Subtracts a float from each component in the Vector4, and returns a new Vector4.
Vector4 -= Vector4 **Vector4 -= Color** **Color -= Vector4** **Color -= Color**	Subtracts the second Vector4 from the first, modifying the values held in the first.
Vector4 -= float **Color -= float**	Subtracts a float from each component in the Vector4, modifying it.
Vector4 * Vector4 **Vector4 * Color** **Color * Vector4** **Color * Color**	Multiplies two Vector4s together component-wise, and returns a new Vector4. EG. Vector4(2) * Vector4(5) == Vector4(10, 10, 10, 10)

Vector4 * float **Color * float**	Multiplies each component of a Vector4 by a float, and returns a new Vector4.
Vector4 *= Vector4 **Vector4 *= Color** **Color *= Vector4** **Color *= Color**	Multiplies two Vector4s together component-wise, modifying the first Vector4.
Vector4 *= float **Color *= float**	Multiplies a Vector4 by a float component-wise, modifying the Vector4.
Vector4 / Vector4 **Vector4 / Color** **Color / Vector4** **Color / Color**	Divides two Vector4s component-wise, and returns a new Vector4. EG. Vector4(10) / Vector4(5) == Vector4(2, 2, 2, 2)
Vector4 / float **Color / float**	Divides each component of a Vector4 by a float, and returns a new Vector4.
Vector4 /= Vector4 **Vector4 /= Color** **Color /= Vector4** **Color /= Color**	Divides two Vector4s component-wise, modifying the first Vector4.
Vector4 /= float **Color /= float**	Divides a Vector4 by a float component-wise, modifying the Vector4.
-Vector4 **-Color**	Multiplies each component of a Vector4 by -1, returning a new Vector4.
Vector4 == Vector4 **Vector4 == Color** **Color == Vector4** **Color == Color**	Returns True if the two Vector4s are equal.
Vector4 != Vector4 **Vector4 != Color** **Color != Vector4** **Color != Color**	Returns True if the two Vector4s are NOT equal.

Vector4[int]
Color[int]
	Subscript operator. Index must be between 0 and 3. EG:
	vec = Vector4(1, 2, 3, 4)
	vec[0] == vec.x == vec.r # True
	vec[1] == vec.y == vec.g # True
	vec[2] == vec.z == vec.b # True
	vec[3] == vec.w == vec.a # True

Vector3

Vector3s often represent Transform position and scale. Really, I can’t think of anything else you would use these for.

Constructors

Vector3()	Constructs an empty Vector3, with all values initialised to 0.0.
Vector3(Vector2, float)	Constructs a Vector3 from a Vector2, with the additional z component specified.
Vector3(float)	Constructs a Vector3 with all components initialised to the given value.
Vector3(float, float, float)	Constructs a Vector3 with the given x, y, and z values.
Vector3(Vector4)	Constructs a Vector3 from a Vector4, truncating the w component.

Static Properties

.up	Equivalent to Vector3(0, 1, 0), but faster in calculations.
.down	Equivalent to Vector3(0, -1, 0), but faster in calculations.
.left	Equivalent to Vector3(-1, 0, 0), but faster in calculations.
.right	Equivalent to Vector3(1, 0, 0), but faster in calculations.
.forward	Equivalent to Vector3(0, 0, -1), but faster in calculations. (-Z is our camera’s forward direction.)
.back	Equivalent to Vector3(0, 0, 1), but faster in calculations. (-Z is our camera’s forward directions.)

Public Properties

.x	The x component of the Vector3.
.y	The y component of the Vector3.
.z	The z component of the Vector3.
.length	The length of the Vector3. Inefficient for most calculations.
.lengthSqr	The square length of the Vector3. Much more efficient to calculate than **.length**, so use this where possible.

Public Methods

Rotate2D(float)	Rotates the Vector3 the given radians around the Z axis.
Normalize()	Normalises the Vector3 (makes its length 1.0).
DistanceTo (Vector3)	Calculates the distance to another Vector3. Makes use of a square-root operation, so it is somewhat inefficient.
SqrDistanceTo (Vector3)	Calculates the square distance to another Vector3. Much more efficient than **DistanceTo**, so use this where you can.
Lerp(Vector3, t)	Linearly interpolates between two vectors given t, returning a new Vector3.
ToDegrees()	Converts this Vector3, assuming it represents angles in radians, to degrees. Modifies self.
ToRadians()	Converts this Vector3, assuming it represents angles in degrees, to radians. Modifies self.
ToQuaternion()	Converts this Vector3, assuming it represents angles in radians, to a new Quaternion.

Operators

Vector3 + Vector3	Adds together two Vector3s component-wise, returning a new Vector3.
Vector3 + Vector2	Adds a Vector2 to a Vector3 component-wise, returning a new Vector3.
Vector3 + float	Adds a float to each component of a Vector3, returning a new Vector3.
Vector3 += Vector3	Adds the second Vector3 to the first (component-wise), modifying it.

Vector3 += float	Adds a float to the Vector3 component-wise, modifying it.
Vector3 - Vector3	Subtracts the second Vector3 from the first (component-wise), returning a new Vector3.
Vector3 - Vector2	Subtracts a Vector2 from a Vector3 (component-wise), returning a new Vector3.
Vector3 - float	Subtracts a float from each component of a Vector3, returning a new Vector3.
Vector3 -= Vector3	Subtracts the second Vector3 from the first (component-wise), modifying it.
Vector3 -= float	Subtracts a float from each component of a Vector3, modifying it.
Vector3 * Vector3	Multiplies two Vector3s component-wise, returning a new Vector3.
Vector3 * Vector2	Multiplies a Vector3 by a Vector2 component-wise, returning a new Vector3.
Vector3 * float	Multiplies each component of a Vector3 by a float, returning a new Vector3.
Vector3 *= Vector3	Multiplies two Vector3s component-wise, modifying the first Vector3.
Vector3 *= float	Multiplies each component of a Vector3 component-wise, modifying it.
Vector3 / Vector3	Divides the first Vector3 by the second component-wise, returning a new Vector3.
Vector3 / Vector2	Divides the Vector3 by a Vector2 component-wise, returning a new Vector3.
Vector3 / float	Divides each component of a Vector3 by a float, returning a new Vector3.
Vector3 /= Vector3	Divides the first Vector3 by the second component-wise, modifying the first.
Vector3 /= float	Divides each component of the Vector3 by a float, modifying it.
-Vector3	Negates each component of a Vector3, returning a new Vector3.
Vector3 == Vector3	Returns True if the two Vector3s are equal.
Vector3 != Vector3	Returns True if the two Vector3s are NOT equal.
Vector3[int]	Subscript operator. Index must be between 0 and 2. EG:

```
vec = Vector3(1, 2, 3)
vec[0] == vec.x
vec[1] == vec.y
vec[2] == vec.z
```

Vector2

Vector2s are often used in relation to Colliders or Physics components. It is common to convert to/from these and Vector3s.

Constructors

Vector2()	Constructs an empty Vector2, with all values initialised to 0.0.
Vector2(float)	Constructs a Vector2 with all components initialised to the given value.
Vector2(float, float)	Constructs a Vector2 with the given x and y values.
Vector2(Vector3)	Constructs a Vector2 from a Vector3, truncating the z component.

Static Properties

.up	Equivalent to Vector2(0, 1), but faster in calculations.
.down	Equivalent to Vector2(0, -1), but faster in calculations.
.left	Equivalent to Vector2(-1, 0), but faster in calculations.
.right	Equivalent to Vector2(1, 0), but faster in calculations.

Public Properties

.x	The x component of the Vector2.
.y	The y component of the Vector2.
.length	The length of the Vector2. Inefficient for most calculations.
.lengthSqr	The square length of the Vector2. Much more efficient to calculate than **.length**, so use this where possible.

Public Methods

Rotate2D(float)	Rotates the Vector3 the given radians around the Z axis.

Normalize()	Normalises the Vector3 (makes its length 1.0).
DistanceTo (Vector3)	Calculates the distance to another Vector3. Makes use of a square-root operation, so it is somewhat inefficient.
SqrDistanceTo (Vector3)	Calculates the square distance to another Vector3. Much more efficient than **DistanceTo**, so use this where you can.
Lerp(Vector3, t)	Linearly interpolates between two vectors given t, returning a new Vector3.

Operators

Vector2 + Vector2	Adds together two Vector2s component-wise, returning a new Vector2.
Vector2 + float	Adds a float to each component of a Vector2, returning a new Vector2.
Vector3 += Vector3	Adds the second Vector2 to the first (component-wise), modifying it.
Vector2 += float	Adds a float to the Vector2 component-wise, modifying it.
Vector2 - Vector2	Subtracts the second Vector2 from the first (component-wise), returning a new Vector2.
Vector2 - float	Subtracts a float from each component of a Vector2, returning a new Vector2.
Vector2 -= Vector2	Subtracts the second Vector2 from the first (component-wise), modifying it.
Vector2 -= float	Subtracts a float from each component of a Vector2, modifying it.
Vector2 * Vector2	Multiplies two Vector2s component-wise, returning a new Vector2.
Vector2 * float	Multiplies each component of a Vector2 by a float, returning a new Vector2.
Vector2 *= Vector2	Multiplies two Vector3s component-wise, modifying the first Vector2.
Vector2 *= float	Multiplies each component of a Vector2 component-wise, modifying it.
Vector2 / Vector2	Divides the first Vector2 by the second component-wise, returning a new Vector2.

Vector3 / float	Divides each component of a Vector2 by a float, returning a new Vector2.
Vector2 /= Vector2	Divides the first Vector2 by the second component-wise, modifying the first.
Vector2 /= float	Divides each component of the Vector2 by a float, modifying it.
-Vector2	Negates each component of a Vector2, returning a new Vector2.
Vector2 == Vector2	Returns True if the two Vector2s are equal.
Vector2 != Vector2	Returns True if the two Vector2s are NOT equal.
Vector2[int]	Subscript operator. Index must be 0 or 1. EG: vec = Vector3(1, 2) vec[0] == vec.x vec[1] == vec.y

Quaternion

Quaternions are a mysterious beast. If you understand them, you've got one on me. However, you probably know a trick or two learned from fucking around in Unity, plus there are numerous examples found on the web of how to use them to accomplish certain tasks. Therefore, I provide to you here an API interface for handling quaternions in as painless a way as possible.

Constructors

Quaternion()	Constructs an empty Quaternion, with all values initialised to 0.0.
Quaternion(float, float, float, float)	Constructs a Quaternion with the given x, y, z, and w values.
Quaternion(Vector3)	Constructs a Quaternion from the given Vector3. The w component will be initialised to 1.0. It will NOT, however, convert Euler angles to the Quaternion; use **Quaternion.Euler** for that.

Static Methods

Lerp(Quaternion, Quaternion, t)	Linearly interpolates between two Quaternions.
Slerp(Quaternion, Quaternion, t)	Performs a spherical linear interpolation between the given Quaternions, given parameter t. See Unity examples of this.
Euler(Vector3)	Takes in a Vector3 of Euler angles (in radians), returning the corresponding Quaternion.

Public Properties

.x	The x component of the Quaternion.
.y	The y component of the Quaternion.
.z	The z component of the Quaternion.
.w	The w component of the Quaternion.

Public Methods

Rotate2D(float)	Rotates the Quaternion the given radians around the Z axis.
Normalize()	Normalises the Quaternion (makes its length 1.0, whatever that might mean).
ToEulerAngles()	Converts this Quaternion to a Vector3 representing Euler angles in radians.
Lerp(Quaternion, t)	Member function version of the **Lerp** function described above.
Slerp(Quaternion, t)	Member function version of the **Slerp** function described above.

Operators

Quaternion + Quaternion	Adds two Quaternions together component-wise, returning a new Quaternion.
Quaternion + float	Adds a float to a Quaternion, returning a new Quaternion.

Quaternion += Quaternion	Adds two Quaternions together component-wise, modifying the first.
Quaternion - Quaternion	Subtracts the second Quaternion from the first, returning a new Quaternion.
Quaternion - float	Subtracts a float from each component of the Quaternion, returning a new Quaternion.
Quaternion -= Quaternion	Subtracts the second Quaternion from the first, modifying the first.
Quaternion * Quaternion	Performs Quaternion multiplication on the two Quaternions, returning a new Quaternion. **By the way**, multiplying two Quaternions essentially adds the two rotations represented by them together.
Quaternion * float	Multiplies each component of a Quaternion by a float, returning a new Quaternion.
Quaternion *= Quaternion	Multiplies two Quaternions, modifying the first.
Quaternion / Quaternion	Divides the first Quaternion by the second, returning a new Quaternion.
Quaternion / float	Divides each component of a Quaternion by a float, returning a new Quaternion.
Quaternion == Quaternion	Returns True if the two Quaternions are equal.
Quaternion != Quaternion	Returns True if the two Quaternions are NOT equal.

Example 18 : Fun with Quaternions!

```
# EG. 18 : Fun with Quaternions!
from PyroCore import *

class QuaternionExample(IScript):
  def Update(this, dt):
    addAngle = Quaternion.Euler(Vector3(0, 0, 10 * dt).ToRadians())
    this.transform.rotation *= addAngle # Multiplication adds rotation
    # And that's it! This adds 10 degrees to the current angle of rotation
    # per second.
```

Matrix4

Your uses of this class will be limited. Currently, not many functions available to us in C++ that take Matrix4s are bound to Python. The only ones that come to mind are the Camera functions, and it is not likely you will know what to do to make any use of those function. Nevertheless, I will document Matrix4s here.

Constructors

Matrix4()	Constructs an identity Matrix4. That is,
	`{ { 1, 0, 0, 0 },`
	`  { 0, 1, 0, 0 },`
	`  { 0, 0, 1, 0 },`
	`  { 0, 0, 0, 1 } }`

Matrix4(float)	Constructs a Matrix4 as if it were a scale matrix, with the identity values equal to the passed value. That is, with value X:
	`{ { X, 0, 0, 0 },`
	`  { 0, X, 0, 0 },`
	`  { 0, 0, X, 0 },`
	`  { 0, 0, 0, X } }`

Public Methods

At(c, r)	Gets the value at column ‘c’ and row ‘r’.

Operators

Matrix4 + Matrix4	Adds two Matrix4s together component-wise, returning a new Matrix4.
Matrix4 + float	Adds a float to each component of a Matrix4, returning a new Matrix4.
Matrix4 += Matrix4	Adds two Matrix4s together component-wise, modifying the first Matrix4.
Matrix4 += float	Adds a float to each component of a Matrix4, modifying it.
Matrix4 - Matrix4	Subtracts the second Matrix4 from the first Matrix4 component-wise, returning a new Matrix4.

Matrix4 - float	Subtracts a float from each component of a Matrix4, returning a new Matrix4.
Matrix4 -= Matrix4	Subtracts the second Matrix4 from the first Matrix4 component-wise, modifying the first.
Matrix4 -= float	Subtracts a float from each component of a Matrix4, modifying it.
Matrix4 * Matrix4	Performs matrix multiplication on two Matrix4s, returning a new Matrix4. Remember: not commutative!
Matrix4 * float	Multiplies each component of a Matrix4 by a float, returning a new Matrix4.
Matrix4 *= Matrix4	Performs matrix multiplication on two Matrix4s, modifying the first Matrix4.
Matrix4 *= float	Multiplies each component of a Matrix4 by a float, modifying it.
Matrix4 / Matrix4	Performs matrix division (the second is inverted) on two Matrix4s, returning a new Matrix4.
Matrix4 / float	Divides each component of a Matrix4 by a float, returning a new Matrix4.
Matrix4 /= Matrix4	Performs matrix division (the second is inverted) on two Matrix4s, modifying the first.
Matrix4 /= float	Divides each component of a Matrix4 by a float, modifying it.
Matrix4 == Matrix4	Returns True if the two Matrix4s are equal.
Matrix4 != Matrix4	Returns True if the two Matrix4s are NOT equal.
Vector4 * Matrix4	Performs vector-matrix multiplication (column-major), returning a new Vector4.
Matrix4 * Vector4	Performs matrix-vector multiplication (column-major), returning a new Vector4.
Matrix4[int]	Subscript operator. Returns the column at index (a Vector4). Index must be between 0 and 3 (inclusive). Note that the subscript operator with the Vector4 can be chained, such that: mat = Matrix4(1.0) mat[0][0] == 1.0 mat[1][0] == 0.0 mat[1][1] == 1.0 # etc.

Graphics Classes (+ Spine)

The following classes have exclusively to do with graphics and Spine animations and are available to you in Python:

- **Animation** : a specific animation playable by an AnimationState.
- **AnimationState** : the state of a particular Mesh’s animation.
- **AnimationStateData** : the shared data between all AnimationStates of a particular type. EG, all ScytheFish may have different AnimationStates, but they share an AnimationStateData.
- **TrackEntry** : the currently playing track of an AnimationState.
- **Event** : Spine animation events that are integral to creating event function callbacks.
- **Skeleton** : the thing that tells a Mesh how to animate.
- **SkeletonData** : shared data between all Skeletons of a given type.
- **Skin** : a skin for the Spine Skeleton to appear as.
- **Attachment** : additional add-ons for Skeletons.
- **Slot** : a slot in a Skeleton where an Attachment can go.
- **Atlas** : a Spine Atlas with defined (named) regions for you to access. Mostly used to get and pass to and from functions, though it has one somewhat useful member function: **FindRegion**.
- **AtlasRegion** : can be gotten from an Atlas. We’ve bound most of the potentially useful struct members to Python, but even then, I don’t think this will ever get used.
- **Mesh** : the actual thing with vertices that defines what to draw on screen.
- **Texture** : the thing that colours a Mesh.
- **Camera** : allows you to control Camera settings. Get from **Engine.main.RenderSystem().GetMainCamera()**.

Animation

This is a simple class, mostly used to pass around to other functions, unless you want to get information like the name or duration of the Animation.

Public Properties

.name	The name of the Animation. Read only.
.duration	The duration of the Animation, in seconds. Read only.

AnimationState

A much more important class, this is what you will interface with to change Animations, set listeners, get the current TrackEntry, and so on.

Public Properties

.data	The **AnimationStateData** this AnimationState is associated with. Read only.
.timeScale	The time scale this AnimationState animates by. Read/Write.

Public Methods

AddAnimation(string, bool)	Adds an Animation (by string name) to be the next playing track. The boolean flag is whether or not it should loop. Defaults to track 0 and 0.0 delay.
AddAnimation(string, bool, int, float)	Adds an Animation (by string name) to be the next playing track. The boolean flag is whether or not it should loop. The int is the track index (usually 0), and the float is a delay before the Animation should play.
AddAnimation(Animation, bool)	Adds an Animation (by object) to be the next playing track. The boolean flag is whether or not it should loop. Defaults to track 0 and 0.0 delay.
AddAnimation(Animation, bool, int, float)	Adds an Animation (by object) to be the next playing track. The boolean flag is whether or not it should loop. The int is the track index (usually 0), and the float is a delay before the Animation should play.

SetAnimation(string, bool)	Similar to the equivalent **AddAnimation**. **Not safe to call in AnimationState listeners!** Call the **AddAnimation** equivalent instead.
SetAnimation(string, bool, int)	Similar to the equivalent **AddAnimation**, minus the float delay. **Not safe to call in AnimationState listeners!** Call the **AddAnimation** equivalent instead.
SetAnimation(Animation, bool)	Similar to the equivalent **AddAnimation**. **Not safe to call in AnimationState listeners!** Call the **AddAnimation** equivalent instead.
SetAnimation(Animation, bool, int)	Similar to the equivalent **AddAnimation**, minus the float delay. **Not safe to call in AnimationState listeners!** Call the **AddAnimation** equivalent instead.
SetEmptyAnimation(int, float)	Sets the current Animation in the given track with the given mix duration to the empty Animation.
AddEmptyAnimation(int, float, float)	Adds the next Animation in the given track with the given mix duration and given float delay (in seconds) to an empty Animation.
IsAnimation(string)	Returns True if the currently playing Animation has the given string name.
GetCurrentTrack(int)	Gets the current TrackEntry for the given index. Generally, the index you pass will be 0 unless you know what you are doing.
ClearTrack(int)	Clears the TrackEntry at the given index.
ClearTracks()	Clears all TrackEntries regardless of index.
SetListener(Function, Object)	Sets a listener event callback for an AnimationState. That is, every time an event is thrown on the AnimationState, the Python function you pass will be called, and you can do some really cool stuff with that. The function object you pass should take four arguments: the first is the Event, the second is the AnimationState, the third is the TrackEntry, and the fourth is an optional (but still required to be there) argument that will be equal to the Python Object you pass to the **SetListener** function (this function). Object can be None if you have no use for this optional argument.

Example 19 : AnimationState Event Listeners

```python
# EG. 19 : AnimationState Event Listeners
from PyroCore import *

class ListenerExample(IScript):
  def SpineListener(event, animState, track, player):
    # player was what was passed into SetListener (in Start)
    if animState.IsAnimation("water_swim_scythe_attack"):
      if event.type == EventType.Complete:   # More on Events in the next section.
        animState.AddAnimation("swim", True) # HAVE to use AddAnimation;
                                             # cannot use SetAnimation!
        player.ClearTag(Tags.Attacking)
      elif event.name == "swoosh":
        player.SetTag(Tags.Attacking)

  def Start(this):
    animState = this.gameObject.GetComponent(Sprite).GetAnimationState()
    # This is how you pass a Python function as an object to another function:
    animState.SetListener(ListenerExample.SpineListener, this.gameObject)
    # Tada! The listener is now set up.
```

AnimationStateData

You won't have much use of this class (I don't think), but it's here anyway.

Public Properties

.defaultMix	The default mix applied to the Animations, if a mix is not defined between two Animations. Default is 0.0. Read/Write.
.skeletonData	The SkeletonData associated with this AnimationStateData, as defined by Spine. Read only.

Public Methods

GetMix(Animation, Animation)	Gets the mix currently being applied when the first Animation plays into the second. If there was never one defined, returns **.defaultMix**.
SetMix(Animation, Animation, float)	Sets the mix for when the first Animation fades into the second one.
SetMix(string, string, float)	Same as above, but instead of supplying Animation objest, you can just use their names here.

Example 20 : Where to Define Mix

```
# EG. 20 : Where to Define Mix
from PyroCore import *

class PlayerFactory(IFactory):
  def Create(system, name):
    player = system.CreateGameObject("player")
    sprite = player.AddComponent(Sprite)
    sprite.SetGraphic("kora")
    data = sprite.GetAnimationStateData()
    data.SetMix("swim", "swim_idle_test2", 0.2) # set the mix to be 0.2 seconds
    data.SetMix("swim_idle_test2", "swim", 0.2)
    # ...other setup here
    # ...
    return player
```

TrackEntry

One of the currently playing tracks in an AnimationState. Can be gotten from an **AnimationState** with the **GetCurrentTrack** method.

Public Properties

.animation	The Animation associated with this TrackEntry. Read only.
.trackIndex	The track index of this TrackEntry. Most of the time, this will be 0, unless you are purposefully utilising track mixing. Read only.
.mixTime	The current mix time of the TrackEntry. Read only.
.mixDuration	The total mix time that the TrackEntry will undergo. Read only.
.loop	Whether or not the TrackEntry is looping. Read only.
.trackTime	The current time the Animation has been playing. Read only.
.trackEnd	The end time of the Animation, plus mix times and whatnot. Read only.

Public Methods

SetListener(Function, Object)	The same as **AnimationState.SetListener**, except binds this listener to the specific TrackEntry. See the above section on similar usage.

Example 21 : Alternative to Listeners

```
# EG. 21 : Alternative to Listeners
# I don't personally recommend this method, it is simply to demonstrate how
# to access TrackEntry data.
from PyroCore import *

class PlayerController(IScript):

  def Start(this):
    this.sprite = this.gameObject.GetComponent(Sprite)
    # ... do more stuff

  def Update(this, dt):
    track = this.sprite.GetAnimationState().GetCurrentTrack(0)
    if track.animation.name == "water_swim_scythe_attack" and \
       track.trackTime >= track.animation.duration:
      # This is waaaaay dirtier than using a listener, by the way.
      this.sprite.GetAnimationState().SetAnimation("swim", True)
```

What the above script does is waits until the current Animation is “water_swim_scythe_attack” and until this Animation is done, and then switches the Animation to be “swim”. Again, I don’t recommend accomplishing this thing this way, but it is here for demonstration purposes.

Event

Events are what are passed to listener functions, and they give very helpful information. First, let's define the enumeration that tells us what type of event we're dealing with:

enum EventType:

EventType.Start	the event when the Animation first starts.
EventType.Interrupt	the event when the Animation is interrupted.
EventType.End	the event when the Animation is exiting.
EventType.Complete	the event when the Animation is finished.
EventType.Dispose	the event when the Animation is about to be destroyed.
EventType.Event	a special, animator-defined event that possibly contains special values.

Public Properties

.type	The type of the Event, the possible values of which are outlined above. Useful for fast, efficient checking.
.name	The name of the event. **Will only exist if event.type == EventType.Event.**
.int	An integer value the animator coded into the Event. **Will only exist if event.type == EventType.Event.**
.float	A float value the animator coded into the Event. **Will only exist if event.type == EventType.Event.**
.string	A string value the animator coded into the Event. **Will only exist if event.type == EventType.Event.**

See Example 19 for a sample of how to use Events.

Skeleton

This class has its uses. For one thing, it allows you to set the skin of a Spine graphic. Or, it allows you to flip the x or y axes of the graphic. These will be shown as examples at the end of this subsection.

Public Properties

.data	The **SkeletonData** associated with this Skeleton instance. Read only.
.flipX	If set to True, the graphic will be flipped in the x-axis. Read/Write.
.flipY	If set to True, the graphic will by flipped in the y-axis. Read/Write.
.skin	The current Skin applied to the Skeleton. Read only.

Public Methods

SetToSetupPose()	Sets the bones, constraints, and slots to their setup pose values.
SetBonesToSetupPose()	Sets the bones and their constraints to their setup pose values.
SetSlotsToSetupPose()	Sets the slots to their setup pose values.
SetSkin(Skin)	Sets the Skin on the Skeleton.
SetSkin(string)	Sets the Skin with the given name on the Skeleton.
GetAttachmentForSlotName (string, string)	Gets the Attachment with the given name (second parameter) for the Slot with the given name.
GetAttachmentForSlotIndex (int, string)	Gets the Attachment with the given name for the Slot with the given integer index.
SetAttachment(string, string)	Sets the Attachment with the given name (second parameter) on the Slot with the given name (first parameter).

Example 22 : Flipping an Enemy Based on Target

```python
# EG. 22 : Flipping an Enemy Based on Target
from PyroCore import *

class EnemyLookat(IScript):
  def Start(this):
    this.target = this.gameObject.system.FindWithName("player").transform
    this.skele = this.gameObject.GetComponent(Sprite).GetSkeleton()
  def Update(this, dt):
    totarget = this.target.position - this.transform.position
    angle = Mathf.Atan2(totarget.y, totarget.x)
    if this.transform.position.x < this.target.position.x:
      this.skele.flipX = True # <-- here is the magic
      this.transform.rotation = Quaternion.Euler(Vector3(0, 0, angle - Mathf.PI))
    else:
      this.skele.flipX = False
      this.transform.rotation = Quaternion.Euler(Vector3(0, 0, angle)
```

Example 23 : Spine Attachments

```python
# EG. 23 : Spine Attachments
# The following is untested and is mostly placeholder until we can test it.
from PyroCore import *

class MutatedPlayer(IFactory):
  def Create(system, name):
    player = system.CreateGameObject("player")
    sprite = player.AddComponent(Sprite)
    skele = sprite.SetGraphic("kora").GetSkeleton()
    skele.SetAttachment("rarm", "scythe") # These names are completely made up.
                                          # This just shows how you WOULD use it.
    # do more setup...
    # ...
    return player
```

SkeletonData

SkeletonData, for your purposes, is mostly for finding Spine objects, namely Skins, Events, Animations, and Slots.

Public Members

FindSkin(string)	Finds the Skin with the given name. Returns None if no such Skin exists.
FindEvent(string)	Finds the Event with the given name. Returns None if no such Event exists.
FindAnimation(string)	Finds the Animation with the given name. Returns None if no such Animation exists.
FindSlot(string)	Finds the Slot with the given name. Returns None of no such Slot exists.

Skin

Public Properties

.name	The name of the Skin. Read only.

Public Methods

AddAttachment(int, string, Attachment)	Adds an Attachment to the Skin at the given Slot index and gives it a name.
GetAttachment(int, string)	Gets the Attachment at the given Slot index with the given name. Returns None if no such Attachment exists.
GetAttachmentName (int, int)	Gets the name of the Attachment at the given Slot index and Attachment index. Returns empty string if no such Attachment exists.

Attachment

Attachments are a bit hairy. Spine uses some fucky C inheritance, and that makes it very difficult to bind to Python. For the time being, this class is here as a sort-of placeholder, though it is also bound so that you can get and pass it between functions.

Public Properties

.name	The name of the Attachment. Read only.
.type	The type of the Attachment. See below. Read only.

enum AttachmentType

- **AttachmentType.Region**
- **AttachmentType.BoundingBox**
- **AttachmentType.Mesh**
- **AttachmentType.LinkedMesh**
- **AttachmentType.Path**
- **AttachmentType.Point**
- **AttachmentType.Clipping**

Slot

Mostly used for passing between functions and getting the Slot index.

Public Properties

.name	The name of the Slot. Read only.
.index	The index of the Slot in its **SkeletonData**. Read only.
.attachmentName	The name of the attached Attachment. Read only.

Atlas

Public Methods

FindRegion(string)	Finds the **AtlasRegion** with the given name in the Atlas. Returns None if there is no such region.

AtlasRegion

Public Properties

.name	The name of the AtlasRegion. Read only.
.x	The x-position of the AtlasRegion in the Atlas. Read only.
.y	The y-position of the AtlasRegion in the Atlas. Read only.
.w	The width of the AtlasRegion. Read only.
.h	The height of the AtlasRegion. Read only.
.u	The u-coordinate of the AtlasRegion. Read only.
.v	The v-coordinate of the AtlasRegion. Read only.
.u2	The end u-coordinate of the AtlasRegion. Read only.
.v2	The end v-coordinate of the AtlasRegion. Read only.
.rotate	Whether or not the AtlasRegion is rotated from the original. Read only.
.flip	Whether or not the AtlasRegion is flipped from the original. Read only.

Mesh

The actual thing with vertices that defines what to draw on screen.

enum DrawMode

TODO : describe each DrawMode.

- **DrawMode.Points**
- **DrawMode.Lines**
- **DrawMode.LineStrip**
- **DrawMode.LineLoop**
- **DrawMode.Triangles**
- **DrawMode.TriangleStrip**
- **DrawMode.TriangleFan**
- **DrawMode.LinesAdjacency**
- **DrawMode.LineStripAdjacency**
- **DrawMode.TriangleStripAdjacency**
- **DrawMode.Patches**

enum Mesh.Flags

- **Mesh.Flags.None**	: no special flags are set on this Mesh.
- **Mesh.Flags.Dynamic**	: this Mesh is generated (i.e., from Spine).
- **Mesh.Flags.DeleteOnUnique**	: delete this Mesh if it is not used anywhere.
- **Mesh.Flags.Default**	: points to Mesh.Flags.DeleteOnUnique

Public Properties

.name	The name of the Mesh. Read only.
.drawMode	The **DrawMode** of the Mesh. See above. Read/Write.
.drawPriority	The drawing priority of the Mesh (think, layers). Can be set in the editor to test for good values. Read/Write.

Public Methods

GetTexture(int)
GetTexture()
	Gets the Texture, optionally with the specified index. Currently, the only valid index is 0, but this may change.

SetTexture(Texture)
SetTexture(Texture, int)
	Sets the Mesh's Texture. Currently, the int you pass in should be 0.

SetTexture(string)
SetTexture(string, int)
	Sets the Mesh's Texture, attempting to fetch the Texture by its name from the RenderSystem. Currently, the int you pass in should be 0.

EraseTexture(int)
EraseTexture()
	Deletes the Texture from the Mesh. Optionally, supply an index, though currently the only valid index is 0, but this may change.

GetAtlas()	Gets the Atlas attached to this Mesh. Returns None if one does not exist.
SetAtlas(Atlas)	Sets the provided Atlas on the Mesh.
SetAtlas(string)	Attempts to fetch the Atlas with the given name from the RenderSystem and set it on the Mesh. If there is no such Atlas, an error will be logged.
GetAnimationStateData ()	Gets the AnimationStateData of the Mesh. Returns None if there is no such AnimationStateData.
SetAnimationStateData (AnimationStateData)	Sets the provided AnimationStateData on the Mesh.
SetAnimationStateData (string)	Attempts to fetch the AnimationStateData with the given name from the RenderSystem and set it on the Mesh. If there is no such AnimationStateData, an error will be logged.
GetSkeleton()	Gets the Skeleton, if there is one. Else, returns None.
IsDynamic()	Whether or not this Mesh is considered dynamic (i.e., was created by Spine).

Texture

The thing that colours a **Mesh.**

enum Texture.Wrapping

- **Texture.Wrapping.Repeat**
- **Texture.Wrapping.MirroredRepeat**
- **Texture.Wrapping.ClampToEdge**
- **Texture.Wrapping.ClampToBorder**

enum Texture.MinFilter

- **Texture.MinFilter.Nearest**
- **Texture.MinFilter.Linear**
- **Texture.MinFilter.NearestMipmapNearest**
- **Texture.MinFilter.LinearMipmapNearest**
- **Texture.MinFilter.NearestMipmapLinear**
- **Texture.MinFilter.LinearMipmapLinear**

enum Texture.MagFilter

- **Texture.MagFilter.Nearest**
- **Texture.MagFilter.Linear**

Public Properties

.name	The name of the Texture. Read only.
.size	The size of the Texture. Read only.

Public Methods

SetWrapping(Wrapping)	Sets the UV wrapping technique used on the Texture.
SetFilter(MinFilter, MagFilter)	Sets the MinFilter and MagFilter of the Texture.

Camera

Static Properties

.DefaultEye	The default ‘eye’ vector.	(Vector3(0, 0, 50))
.DefaultCenter	The default ‘center’ vector.	(Vector3(0, 0, -1))
.DefaultUp	The default ‘up’ vector.	(Vector3(0, 1, 0))
.DefaultView	The default ‘view’ matrix.	

Static Methods

Create(fovY, ratio, nearDist, farDist, eye, centre, up)	Creates a new Camera. The parameters are finicky, but try using the defaults listed above as a starting point.
Create(fovY, ratio, nearDist, farDist, view)	Creates a new Camera. ‘view’ is a Matrix4, which you have to get from somewhere else (like the default listed above), or else calculate it yourself.

Public Properties

.position	The position of the Camera in world space. Cannot be modified directly, but can be set. Read/Write.
.ratio	The aspect ratio of the Camera. Read/Write.
.fovY	The field-of-view of the Camera. Read/Write.
.farDist	The far clipping plane distance of the Camera. Read/Write.
.nearDist	The near clipping plane distance of the Camera. Read/Write.

Public Methods

GetView()	Gets the view Matrix4 from the Camera.
GetProjection()	Gets the projection Matrix4 from the Camera.
GetRatio()	Gets the aspect ratio from the Camera.
SetRatio(float)	Sets the aspect ratio on the Camera.
GetFovY()	Gets the field-of-view of the Camera.
SetFovY(float)	Sets the field-of-view on the Camera.
GetFarDist()	Gets the far clipping plane distance of the Camera.
SetFarDist(float)	Sets the far clipping plane distance of the Camera.
GetNearDist()	Gets the near clipping plane distance of the Camera.
SetNearDist(float)	Sets the near clipping plane distance of the Camera.
GetPosition()	Gets the Vector3 position of the Camera in world space.
SetPosition(Vector3)	Sets the Vector3 position of the Camera in world space.
AddPosition(Vector3)	Adds a Vector3 to the Camera’s current position in world space.
Rotate(Quaternion)	Rotates the Camera by a given Quaternion.
Lookat(Matrix4)	Resets the view to the transform described by the Matrix4.
Lookat(eye, center, up)	Resets the view to the transform given by the Vector3s ‘eye’, ‘center’, and ‘up’.

Other Classes

Here is where all the misfits lie;

- **GameObjectList** : A very useful container, which can only hold GameObjects. If added as an Attribute, it is viewable and editable in the editor!

- **Timestamp** : Creates a unique timestamp upon construction. It can be pretty-printed and whatnot. Kind of useless, seeing as printing a log message prints a Timestamp for you anyway. Because of this, I will save my time and omit it here. Just know it exists and is bound!

GameObjectList

As mentioned, GameObjectLists are useful because they can be viewed and edited in the editor.

Bonus feature : if a GameObject gets deleted anywhere else in the code, it is also removed from the list. This means you don’t have to check every frame if the GameObject is deleted and should be removed!

More code examples of GameObjectLists can be found throughout this document, such as Examples 5 & 11.

Static Methods

Create()	The only way you should (and can) create GameObjectLists. Call like **GameObjectList.Create()**. Likely, you’ll want to call it within a **CreateAttribute** call, as such: **this.gameObject.CreateAttribute(“list”, GameObjectList.Create())**

Public Methods

ToString()	Generates a formatted string for you to output to the logs.

Size()
size()
	Returns the number of GameObjects contained in the list.

PushBack(GameObject)
append(GameObject()
	Pushes a GameObject to the back of the list.

PushFront(GameObject)
prepend(GameObject)
	Pushes a GameObject to the front of the list.

PopBack()	Pops the GameObject at the back of the list out of the list.
PopFront()	Pops the GameObject at the front of the list out of the list.

Remove(GameObject)
remove(GameObject)
	Removes the given GameObject from the list.

Clear()
clear()
	Clears the list.

Iterable

GameObjectLists are considered *iterable*; that means, you can iterate through them with a for-loop like so:

```
list = GameObjectList.Create()
list.append(fef1)
list.append(fef2)
list.append(fef3)
...
for obj in list:
  Debug.LogUser(obj.name + " is in the list.")
```

Gives the following output, assuming the fefs have the same names as their variables:

```
<YOU> [2018-12-19 14:05:32.587] : fef1 is in the list.
<YOU> [2018-12-19 14:05:32.587] : fef2 is in the list.
<YOU> [2018-12-19 14:05:32.587] : fef3 is in the list.
```

Appendix A : Python Errors

Python errors do not (usually) crash the game, but they can lead to some very weird behaviour. This can often be due to the fact that a script might run up until it hits a line with an error, and then the rest of the script will not run. This can leave some GameObjects only half-constructed, or some behaviours half-executed. Luckily, we have logging in the Terminal window.

Hint: If you get a wall of errors, try pausing execution and scroll to the very top of them. The topmost error usually causes the subsequent errors, so fix that one first.

Hint: you can especially utilise the string filter functionality in the Terminal to filter out exactly what error output you are looking for. Simply scroll the outermost scroll bar in the window up until you see the filter input box at the top, then type in “<PyErr>”. This should give you only Python error messages, which you can much easier sift through.

Without further ado, I will now present some common esoteric and ambiguous Python errors and what causes them.

Error 1 : SyntaxError

A lot of things can cause this one. Luckily, these are caught before the script even runs; they are caught at registry-time, that is, when the game first loads.

Cause 1 : Mismatched Parentheses

I will start with the simplest: mismatched parentheses. Consider the following script:

```
 1  # ERROR 1 : SyntaxError    Cause 1 : Mismatched Parentheses
 2  from PyroCore import *
 3
 4  class ErrorBehaviour(IScript):
 5    def Start(this):
 6      this.sprite = this.gameObject.GetComponent(Sprite)
 7      this.enemies = \                                          # ▼
 8        this.gameObject.CreateAttribute("enemies", GameObjectList.Create()
 9      this.timer = this.gameObject.CreateAttribute("timer",     # ▲
10        Timer.Create(10.0))
11
12    def Update(this, dt):
13      if this.timer.isDone:
14        this.timer.Reset()
15        pred = PredatorFishFactory.Create(this.gameObject.system,
16          "predator%d" % this.enemies.size())
17        this.enemies.append(pred)
```

Can you spot where the error is? It spits up the following output upon registering the script:

```
<ERR> [2018-12-10 16:13:14.934] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 9

<ERR> [2018-12-10 16:13:14.934] : <PyErr> this.timer = this.gameObject.CreateAttribute("timer",

<ERR> [2018-12-10 16:13:14.935] : <PyErr> ^

<ERR> [2018-12-10 16:13:14.936] : <PyErr> SyntaxError
<ERR> [2018-12-10 16:13:14.938] : <PyErr> :
<ERR> [2018-12-10 16:13:14.938] : <PyErr> invalid syntax
```

But this line looks just fine? Where is the syntax error?
Well, my friend, you must look at the line above it. `this.enemies = \ this.gameObject.CreateAttribute("enemies", GameObjectList.Create()` is missing one more parenthesis at the end, so when Python parses the script, it fucks up when it hits the beginning of the next line. To fix this issue, simply check all

your parentheses and that they are all matched. This is easy to do in editors such as Notepad++, which highlights matching parentheses.

Note: If the mismatched parenthesis is at the end of the file, you may get an error like this instead:

```
<ERR> [2018-12-10 16:29:14.164] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 16

<ERR> [2018-12-10 16:29:14.165] : <PyErr>
<ERR> [2018-12-10 16:29:14.166] : <PyErr> ^

<ERR> [2018-12-10 16:29:14.167] : <PyErr> SyntaxError
<ERR> [2018-12-10 16:29:14.168] : <PyErr> :
<ERR> [2018-12-10 16:29:14.168] : <PyErr> unexpected EOF while parsing
```

This is because the Python interpreter was expecting to read in more code at the open of the first parenthesis, but since it never found the closing parenthesis before End-Of-File was reached, it throws 'unexpected EOF while parsing'. So far, a missing parenthesis or a trailing backslash at the end of the file are the only things I've encountered that throw this error specifically.

Cause 2 : Weird Symbols in Weird Places

The error should be obvious since I’m pointing it out, but shit happens in real life.

The script:

```
 1  # ERROR 1 : SyntaxError    Cause 2 : Weird Symbols in Weird Places
 2
 3  class ErrorBehaviour(IScript):
 4    def Start(this):
 5      this.sprite = this.gameObject.GetComponent(Sprite)
 6      this.enemies = \
 7        this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
 8      this.timer = this.gameObject.CreateAttribute("timer",
 9        Timer.Create(10.0))
10      \ # <-- this is weird.
11
12    def Update(this, dt):
13      if this.timer.isDone:
14        this.timer.Reset()
15        pred = PredatorFishFactory.Create(this.gameObject.system,
16          "predator%d" % this.enemies.size())
17        this.enemies.append(pred)
```

The Output:

```
<ERR> [2018-12-10 16:35:52.751] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 11

<ERR> [2018-12-10 16:35:52.753] : <PyErr> ^

<ERR> [2018-12-10 16:35:52.753] : <PyErr> SyntaxError
<ERR> [2018-12-10 16:35:52.754] : <PyErr> :
<ERR> [2018-12-10 16:35:52.754] : <PyErr> invalid syntax
```

Not very descriptive, especially if the weird symbol is not so obvious. But, at least the output gave us a line number. The trick with SyntaxErrors is to look at the line above them, as that is (usually) where the problem lies. Of course, we know that the error here is the stray backslash ‘\’ on line 10.

Error 2 : AttributeError

This is another *really* common one. It happens when a particular object has no member (or 'attribute') with the given name. It is most often due to typos or due to uninitialised variables (such as referencing 'this.timer' before 'this.timer = ...' is invoked).

Cause 1 : Capitalisation Typo

Consider the following script:

```
 1  # ERROR 2 : AttributeError    Cause 1 : Capitalisation Typo
 2  from PyroCore import *
 3
 4  class ErrorBehaviour(IScript):
 5    def Start(this):
 6      this.sprite = this.GameObject.GetComponent(Sprite)
 7      this.enemies = \
 8        this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
 9      this.timer = this.gameObject.CreateAttribute("timer",
10        Timer.Create(10.0))
11
12    def Update(this, dt):
13      if this.timer.isDone:
14        this.timer.Reset()
15        pred = PredatorFishFactory.Create(this.gameObject.system,
16          "predator%d" % this.enemies.size())
17        this.enemies.append(pred)
```

Can you spot the error? It's on line 6, where 'this.GameObject' should be 'this.gameObject'. It gives this output in the console:

```
<ERR> [2018-12-10 16:07:27.388] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 16:07:27.388] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 6, in
Start

<ERR> [2018-12-10 16:07:27.389] : <PyErr> this.sprite = this.GameObject.GetComponent(Sprite)
<ERR> [2018-12-10 16:07:27.390] : <PyErr> AttributeError
<ERR> [2018-12-10 16:07:27.390] : <PyErr> :
<ERR> [2018-12-10 16:07:27.391] : <PyErr> 'ErrorBehaviour' object has no attribute 'GameObject'
<ERR> [2018-12-10 16:07:27.392] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 16:07:27.392] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 13, in
Update

<ERR> [2018-12-10 16:07:27.394] : <PyErr> if this.timer.isDone:
<ERR> [2018-12-10 16:07:27.395] : <PyErr> AttributeError
<ERR> [2018-12-10 16:07:27.395] : <PyErr> :
<ERR> [2018-12-10 16:07:27.396] : <PyErr> 'ErrorBehaviour' object has no attribute 'timer'
```

The error repeats every frame, and that is because the **Start** function failed at line 6, so this.timer (and this.enemies) never get initialised. What we have here is an AttributeError inside of an AttributeError. Because of this, when the **Update** function is called, every frame there will be another AttributeError related to the fact that this.timer does not exist. The fix in this case? Find the typo, and kill it. Kill it good.

Cause 2 : Referencing a Variable Before it Exists

We already saw above that if a variable was not ever set (which can happen due to some other, earlier error), it can lead to AttributeError if it is referenced too early.
The following script attempts to set a new Timer when the current Timer is done; however note that we never created one in **Start**!

```
 1  # ERROR 2 : AttributeError    Cause 2 : Referencing Variable Before Exists
 2
 3  class ErrorBehaviour(IScript):
 4    def Start(this):
 5      this.sprite = this.gameObject.GetComponent(Sprite)
 6      this.enemies = \
 7        this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
 8
 9    def Update(this, dt):
10      if this.timer.isDone:
11        this.timer = this.gameObject.SetAttribute("timer",
12          Timer.Create(10.0))
13        pred = PredatorFishFactory.Create(this.gameObject.system,
14          "predator%d" % this.enemies.size())
15        this.enemies.append(pred)
```

Output:

```
<ERR> [2018-12-10 16:24:05.441] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 16:24:05.442] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 10, in
Update

<ERR> [2018-12-10 16:24:05.442] : <PyErr> if this.timer.isDone:
<ERR> [2018-12-10 16:24:05.443] : <PyErr> AttributeError
<ERR> [2018-12-10 16:24:05.443] : <PyErr> :
<ERR> [2018-12-10 16:24:05.444] : <PyErr> 'ErrorBehaviour' object has no attribute 'timer'
```

Solution? Set up 'this.timer' in **Start** first. Also, you should probably not reset a Timer the way it is done in this example; just call **Timer.Reset()**.

Cause 3 : Wrong Method Name

I've done this a million times. Matter of fact, I did by accident in the writing of this section. What happens is, you go to call a method, say, 'this.gameObject.CreateAttribute'... but instead you call it like 'this.gameObject.CreateComponent'. Oops.

The following script throws an AttributeError:

```
 1  # ERROR 2 : AttributeError    Cause 3 : Wrong Method Name
 2  from PyroCore import *
 3
 4  class ErrorBehaviour(IScript):
 5    def Start(this):
 6      this.sprite = this.gameObject.GetComponent(Sprite)
 7      this.enemies = \
 8        this.gameObject.CreateComponent("enemies", GameObjectList.Create())
 9      this.timer = this.gameObject.CreateComponent("timer",
10        Timer.Create(10.0))
11
12    def Update(this, dt):
13      if this.timer.isDone:
14        this.timer.Reset()
15        pred = PredatorFishFactory.Create(this.gameObject.system,
16          "predator%d" % this.enemies.size())
17        this.enemies.append(pred)
```

And the log output:

```
<ERR> [2018-12-10 15:54:04.008] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 15:54:04.009] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 8, in
Start

<ERR> [2018-12-10 15:54:04.013] : <PyErr> this.gameObject.CreateComponent("enemies",
GameObjectList.Create())
<ERR> [2018-12-10 15:54:04.013] : <PyErr> AttributeError
<ERR> [2018-12-10 15:54:04.014] : <PyErr> :
<ERR> [2018-12-10 15:54:04.016] : <PyErr> 'GameObject' object has no attribute 'CreateComponent'
<ERR> [2018-12-10 15:54:04.017] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 15:54:04.018] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 11, in
Update

<ERR> [2018-12-10 15:54:04.019] : <PyErr> if this.timer.isDone:
<ERR> [2018-12-10 15:54:04.019] : <PyErr> AttributeError
<ERR> [2018-12-10 15:54:04.019] : <PyErr> :
<ERR> [2018-12-10 15:54:04.020] : <PyErr> 'ErrorBehaviour3' object has no attribute 'timer'
```

Lines 8 and 9 need to use 'this.gameObject.CreateAttribute' instead.

Error 3 : NameError

Also quite common and *very* similar to AttributeErrors, a NameError occurs when you use a symbol that isn’t defined anywhere, such as a class name that doesn’t exist. Commonly, this can happen due to refactoring code, where old classes are deleted or renamed but not all of their references reflect that. But, the most common cause...

Cause 1 : MORE TYPOS

```
 1  # ERROR 3 : NameError    Cause 1 : MORE TYPOS
 2  
 3  class ErrorBehaviour(IScript):
 4    def Start(this):
 5      this.sprite = this.gameObject.GetComponent(aprite)
 6      this.enemies = \
 7        this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
 8      this.timer = this.gameObject.CreateAttribute("timer",
 9        Timer.Create(10.0))
10  
11    def Update(this, dt):
12      if this.timer.isDone:
13        this.timer.Reset()
14        pred = PredatorFishFactory.Create(this.gameObject.system,
15          "predator%d" % this.enemies.size())
16        this.enemies.append(pred)
```

The fuck is a ‘aprite’? Did you mean ‘Sprite’? You must’ve, because the error logs are telling me this:

```
<ERR> [2018-12-10 16:48:07.443] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 16:48:07.443] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 5, in
Start

<ERR> [2018-12-10 16:48:07.445] : <PyErr> this.sprite = this.gameObject.GetComponent(aprite)
<ERR> [2018-12-10 16:48:07.445] : <PyErr> NameError
<ERR> [2018-12-10 16:48:07.445] : <PyErr> :
<ERR> [2018-12-10 16:48:07.446] : <PyErr> name 'aprite' is not defined
<ERR> [2018-12-10 16:48:07.447] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 16:48:07.447] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 12, in
Update

<ERR> [2018-12-10 16:48:07.449] : <PyErr> if this.timer.isDone:
<ERR> [2018-12-10 16:48:07.449] : <PyErr> AttributeError
<ERR> [2018-12-10 16:48:07.450] : <PyErr> :
<ERR> [2018-12-10 16:48:07.450] : <PyErr> 'ErrorBehaviour' object has no attribute 'timer'
```

Again, notice the second error. Because of the NameError with 'aprite', more errors cascade as the rest of the **Start** function cannot be called properly. This will be a recurring thing you will notice with Python errors.

Always scroll to the top of the error logs!

Cause 2 : Refactored Class Names

Say we had the class we've been using all along, 'ErrorBehaviour'. Now, say we renamed it to 'ErrorBehaviourRenamed'. What if there was a method or property variable stored in the originally named class? That means everywhere those properties/methods are referenced, this change has to be reflected. If it is not, it's a NameError.

```python
# ERROR 3 : NameError    Cause 2 : Refactored Class Names

class ErrorBehaviourRenamed(IScript):

  timesStarted = 0

  def Start(this):
    this.sprite = this.gameObject.GetComponent(Sprite)
    this.enemies = \
      this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
    this.timer = this.gameObject.CreateAttribute("timer",
      Timer.Create(10.0))

    EnemyBehaviour.timesStarted += 1
    # should be EnemyBehaviourRenamed.timesStarted += 1

  def Update(this, dt):
    if this.timer.isDone:
      this.timer.Reset()
      pred = PredatorFishFactory.Create(this.gameObject.system,
        "predator%d" % this.enemies.size())
      this.enemies.append(pred)
```

Log output and fix on overleaf.

The Output:

```
<ERR> [2018-12-10 17:52:59.301] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 17:52:59.302] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 14, in
Start

<ERR> [2018-12-10 17:52:59.303] : <PyErr> EnemyBehaviour.timesStarted += 1
<ERR> [2018-12-10 17:52:59.303] : <PyErr> NameError
<ERR> [2018-12-10 17:52:59.304] : <PyErr> :
<ERR> [2018-12-10 17:52:59.304] : <PyErr> name 'EnemyBehaviour' is not defined
```

The Fix:

Simple. Either undo the refactor, or change 'ErrorBehaviour.timesStarted' to 'ErrorBehaviourRenamed.timesStarted' as well as in all other relevant occurrences.

Error 4 : UnboundLocalError

This error is quite specific. It happens when you use a local variable before it is ever declared.

Example:

```python
# ERROR 4 : UnboundLocalError

class ErrorBehaviour(IScript):
  def Start(this):
    this.sprite = this.gameObject.GetComponent(Sprite)
    this.enemies = \
      this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
    this.timer = this.gameObject.CreateAttribute("timer",
      Timer.Create(10.0))

  def Update(this, dt):
    if this.timer.isDone:
      count += 1 # wut.
      this.timer.Reset()
      pred = PredatorFishFactory.Create(this.gameObject.system,
        "predator%d" % this.enemies.size())
      this.enemies.append(pred)
```

Output:

```
<ERR> [2018-12-10 16:58:22.444] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 16:58:22.447] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 13, in
Update

<ERR> [2018-12-10 16:58:22.448] : <PyErr> count += 1
<ERR> [2018-12-10 16:58:22.449] : <PyErr> UnboundLocalError
<ERR> [2018-12-10 16:58:22.449] : <PyErr> :
<ERR> [2018-12-10 16:58:22.450] : <PyErr> local variable 'count' referenced before assignment
```

Finally! An actually useful error message!

Because this one is so self-explanatory (just declare the fucking variable first), I will leave it at that.

Error 5 : ArgumentError

This is another common one, and it happens when **you don't read the docs.**
In other words, it happens when you pass the wrong types or number of arguments to a function.
Let's say you do this:

```
 1  # ERROR 5 : ArgumentError
 2
 3  class ErrorBehaviour(IScript):
 4    def Start(this):
 5      this.sprite = this.gameObject.GetComponent(Sprite)
 6      this.enemies = \
 7        this.gameObject.CreateAttribute("enemies", GameObjectList.Create(10))
 8      this.timer = this.gameObject.CreateAttribute("timer",
 9        Timer.Create(10.0))
10
11    def Update(this, dt):
12      if this.timer.isDone:
13        this.timer.Reset()
14        pred = PredatorFishFactory.Create(this.gameObject.system,
15          "predator%d" % this.enemies.size())
16        this.enemies.append(pred)
```

Look at line 7, specifically the '10' at the end. When you do this, an error like this will be spit in your face:

```
<ERR> [2018-12-10 17:03:30.539] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 17:03:30.540] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 7, in
Start

<ERR> [2018-12-10 17:03:30.541] : <PyErr> this.gameObject.CreateAttribute("enemies",
GameObjectList.Create(10))
<ERR> [2018-12-10 17:03:30.542] : <PyErr> Boost.Python
<ERR> [2018-12-10 17:03:30.542] : <PyErr> .
<ERR> [2018-12-10 17:03:30.543] : <PyErr> ArgumentError
<ERR> [2018-12-10 17:03:30.543] : <PyErr> :
<ERR> [2018-12-10 17:03:30.543] : <PyErr> Python argument types in
    None.Create(int)
did not match C++ signature:
    Create(void)
```

GameObjectList.Create takes 0 arguments! Passing the wrong number or types of arguments will result in this error, and luckily, the error message is descriptive enough to help us fix it.

Error 6 : IndentationError / TabError

This one happens because you're a dingus.

Cause 1 : Too Many Spaces

The Script:

```
 1  # ERROR 6 : IndentationError    Cause 1 : Too Many Spaces
 2
 3  class ErrorBehaviour(IScript):
 4    def Start(this):
 5      this.sprite = this.gameObject.GetComponent(Sprite)
 6      this.enemies = \
 7        this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
 8      this.timer = this.gameObject.CreateAttribute("timer",
 9        Timer.Create(10.0))
10
11    def Update(this, dt):
12        if this.timer.isDone: # 2 too many spaces here...
13        this.timer.Reset()    # or 2 too few in the subsequent lines
14        pred = PredatorFishFactory.Create(this.gameObject.system,
15          "predator%d" % this.enemies.size())
16        this.enemies.append(pred)
```

The Output:

```
<ERR> [2018-12-10 17:16:48.251] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 13

<ERR> [2018-12-10 17:16:48.251] : <PyErr> this.timer.Reset()

<ERR> [2018-12-10 17:16:48.252] : <PyErr> ^

<ERR> [2018-12-10 17:16:48.253] : <PyErr> IndentationError
<ERR> [2018-12-10 17:16:48.253] : <PyErr> :
<ERR> [2018-12-10 17:16:48.253] : <PyErr> expected an indented block
```

The Fix:

Delete two spaces in front of the 'if' statement on line 12!

Cause 2 : Mixing Tabs and Spaces

If you do this I will hunt you down and shovel kitty litter up your arsehole so that you have the driest shits imaginable.

```
 1  # ERROR 6 : IndentationError    Cause 2 : Mixing Tabs and Spaces
 2
 3  class ErrorBehaviour(IScript):
 4    def Start(this):
 5      this.sprite = this.gameObject.GetComponent(Sprite)
 6      this.enemies = \
 7        this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
 8      this.timer = this.gameObject.CreateAttribute("timer",
 9        Timer.Create(10.0))
10
11    def Update(this, dt):
12              if this.timer.isDone: # There are tabs in this line!
13        this.timer.Reset()
14        pred = PredatorFishFactory.Create(this.gameObject.system,
15          "predator%d" % this.enemies.size())
16        this.enemies.append(pred)
```

The Output:

```
<ERR> [2018-12-10 17:26:36.429] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 12

<ERR> [2018-12-10 17:26:36.431] : <PyErr> if this.timer.isDone:

<ERR> [2018-12-10 17:26:36.431] : <PyErr> ^

<ERR> [2018-12-10 17:26:36.432] : <PyErr> TabError
<ERR> [2018-12-10 17:26:36.432] : <PyErr> :
<ERR> [2018-12-10 17:26:36.432] : <PyErr> inconsistent use of tabs and spaces in indentation
```

The Fix:

Get your act together. Go to your text editor settings, put ‘insert spaces instead of tabs’, and choose tab width to be 2. That is our standard, our style guide. No excuses.

Error 7 : ArithmeticError

Not very common, but it can happen. I wouldn't quite blame you if it does, because it can catch you with your guard down. Most likely, you divided by 0 somewhere, as in with the following example.

Cause 1 : Divide By Zero (ZeroDivisionError)

The following shows a for-loop that will eventually have i == 0, and we are dividing by i. It is very easy for this to happen.

```
 1  # ERROR 7 : ArithmeticError    Cause 1 : Divide By Zero (ZeroDivisionError)
 2
 3  class ErrorBehaviour(IScript):
 4    def Start(this):
 5      this.sprite = this.gameObject.GetComponent(Sprite)
 6      this.enemies = \
 7        this.gameObject.CreateAttribute("enemies", GameObjectList.Create())
 8      this.timer = this.gameObject.CreateAttribute("timer",
 9        Timer.Create(10.0))
10
11      for i in range(5, -1, -1):
12        Debug.LogUser("%f" % (1.0 / i)) # will eventually divide by 0
13
14    def Update(this, dt):
15      if this.timer.isDone:
16        this.timer.Reset()
17        pred = PredatorFishFactory.Create(this.gameObject.system,
18          "predator%d" % this.enemies.size())
19        this.enemies.append(pred)
```

The Output:

```
<YOU> [2018-12-10 17:36:24.492] : 0.200000
<YOU> [2018-12-10 17:36:24.493] : 0.250000
<YOU> [2018-12-10 17:36:24.493] : 0.333333
<YOU> [2018-12-10 17:36:24.494] : 0.500000
<YOU> [2018-12-10 17:36:24.494] : 1.000000
<ERR> [2018-12-10 17:36:24.495] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 17:36:24.496] : <PyErr>   File "./assets/scripts/behaviors/examples.py", line 12, in
Start

<ERR> [2018-12-10 17:36:24.496] : <PyErr> Debug.LogUser("%f" % (1.0 / i))
<ERR> [2018-12-10 17:36:24.497] : <PyErr> ZeroDivisionError
<ERR> [2018-12-10 17:36:24.498] : <PyErr> :
<ERR> [2018-12-10 17:36:24.499] : <PyErr> float division by zero
```

The Fix:

Either limit your loops to not reach 0, or have an if-check to make sure that your divisor is not 0. These errors can happen at really random times, so you can never be too safe.

Error 8 : TypeError

Abusing objects and using their types unlike they are meant to be used will result in a TypeError.

Cause 1 : Invalid Operators Applied

This happens when **you don't read the documentation** (particularly the 'Math' section) and don't know what operators work on what.

```
# ERROR 8 : TypeError    Cause 1 : Invalid Operators Applied

class ErrorBehaviour(IScript):

  def Start(this):
    this.strn = "fef"
    this.strn /= 5
    # you cannot divide a string by an int!
```

The Output:

```
<ERR> [2018-12-10 20:14:05.996] : <PyErr> Traceback (most recent call last):

<ERR> [2018-12-10 20:14:05.999] : <PyErr>   File "./assets/scripts/behaviors/examples.py",
line 7, in Start

<ERR> [2018-12-10 20:14:05.001] : <PyErr> this.strn /= 5
<ERR> [2018-12-10 20:14:05.001] : <PyErr> TypeError
<ERR> [2018-12-10 20:14:05.002] : <PyErr> :
<ERR> [2018-12-10 20:14:05.002] : <PyErr> unsupported operand type(s) for /=: 'str' and
'int'
```

The Fix:

Just don't do it in the first place. Read the documentation, either here or on the Python official page, for what operators you may use.