Prof. Monster's

BGE Guide to

Python Coding

by Monster
Version 1.0
Date 22 Mar 2012

Hello and welcome!

This guide will give you a brief introduction how to use Python within the Blender Game Engine. The purpose of this guide is to enable you to start with Python coding and to avoid typical obstacles.

This guide will **NOT** teach you Python coding. If you want to learn Python please look for appropriate Python coding tutorials (e.g. ByteOfPython).

When writing Python code for the BGE I recommend to keep the Blender Python API open as reference. You can find it at www.blender.org.

Please be aware the BGE does not support the Blender API called **bpy**. You can access it when starting your game from within Blender. But it is not available in your final game.

Do not be scared of Python coding. It is not that difficult. It is a bit like writing a shopping list. This guide will help you with the first steps.

Introduction

The game logic are the rules of the behavior of your game.

The game logic is defined by

- Implicit logic of the BGE framework e.g.
  - Scene management
  - Rendering
  - Input management
  - Animation system
  - etc.
- Logic bricks setup via
  - GUI
  - Python

Your Python code will be part of the BGE GameLoop. I recommend to read "BGE guide to the GameLoop" first.

You apply your Python code via the Python controller. That means your code will be part of a controller. That makes it fit into the BGE logic.

The BGE expects a controller to continue as fast as possible.

This means your code should not unnecessarily waste processing time or block the GameLoop by not returning at all.

There is no need to simulate existing logic bricks with Python code. Compiled logic bricks are always faster.

Classification

The Python controller acts as controller as all other controllers. That means it collects the input from sensors and activates/deactivates corresponding actuators.

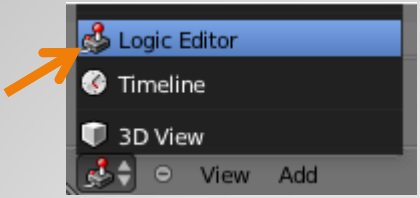
Example: On key press activate an animation

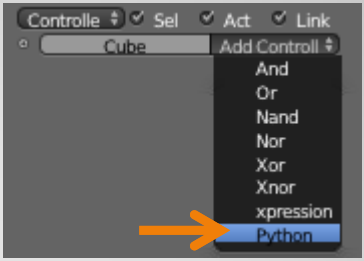
The Python controller acts as an actuator too. That means it can apply changes to the Game without the usage of actuators.

Example: you can let the BGE place an object at another position

It is important to know although the Python controller can act as actuator it does not run with the other actuators. It will execute and run together with the Controllers. This is a very tiny difference but can become very important.

You can add a Python controller at the Logic Editor:





Please do not forget to connect your Python controller with one or more sensors.

Python controller

The Python controller has two modes:

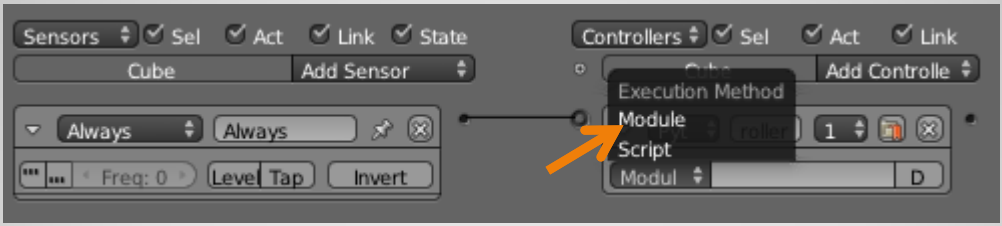
- Script mode (default)
- Module mode

The Module mode has some benefits over the Script mode:

- The Python file can reside internal AND external
- The Python code gets precompiled once (faster).
- The Python code will be written in module form (Python standard)
- You get multiple entry points into the same Python file (better organization)
- Initialization at module level
- It is easy to convert a Script into a Module

For historical reasons the Script mode is the default. I recommend to use the Module mode only.

At the Python controller you select the Mode “**Module**”



Python Modes

Python File

Before you can finish the setup for your Python controller you need a Python file.

The easiest way to create one is to open the Text editor and create a new text block.

Alternative you can create a text file in the folder of your blend file. In both cases the name of the file should end with **“.py”**.

The name of your module should enable the reader to guess what it does.

Good filenames:

- enemyBehavior.py
- mouse.py
- findObjects.py
- actionManager.py
- selector.py
- sceneManager.py

Bad filenames:

- main.py
- python.py
- process.py
- execute.py
- do.py
- perform.py

A Python module is a Python file (text file with extension ".py)" that contains Python code.

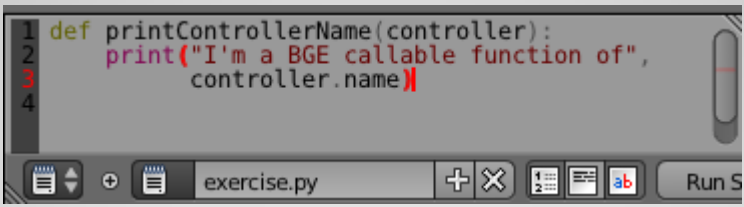
Any Python code will do it. To create a BGE callable Module you need to define one or more BGE entry points.

BGE entry points are functions with one or none formal parameter:

```
def printControllerName(controller):
    print("I'm a BGE callable function of",
          controller.name)
```

If a formal parameter is given (regardless how you name it) the BGE places a reference to the calling controller into it.

With the above lines you have written you first Python module. Place it into a Python file e.g.: **execise.py**.



Python Module

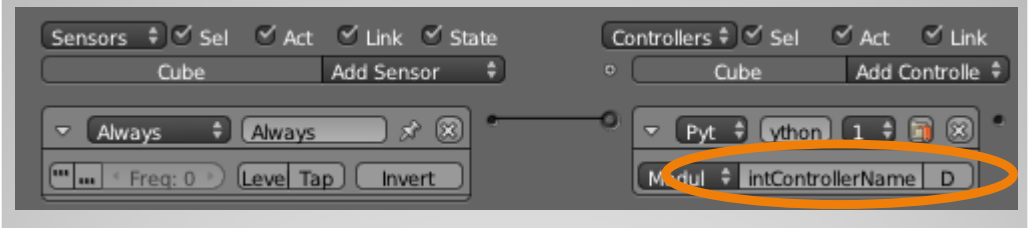
At the Python controller we can now enter the BGE entry point.
It has following form:

- **moduleName.<entryPointName>**

The module name does not include the .py
The entry point name does not include the parenthesis.
(The module name can include a package name)

In our example we configure the Python Controller with:

exercise.printControllerName

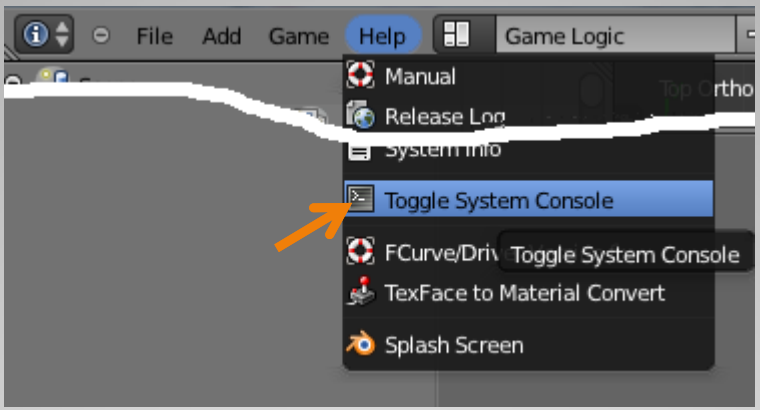


Entry Point

Start your game. The Python controller will be executed when one of the connected sensors triggers.

Simply press <P> within the 3D View

Toggle to your system console



You should see:

```
Blender Game Engine Started
I'm a BGE callable function of Python
```

Congratulations you ran your first BGE Python code

Execute Python code

How does the BGE run the module?

With the first call to a module,

the BGE tells the Python interpreter to load the module.

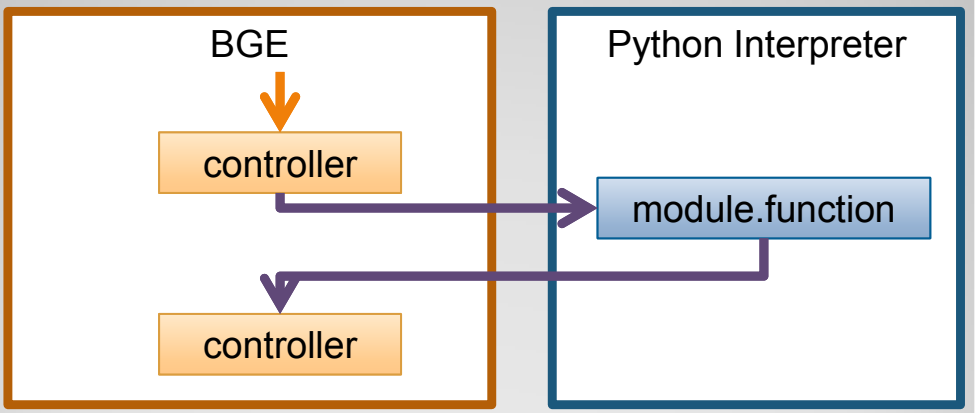
That means:

- The module gets compiled into byte code (if not already)
- The module executes its module level code
  - This is the code without indentation
  - This code will be executed exactly once!

With each execution of the Python controller:

The Python interpreter calls the BGE entry point which is the function mentioned in the module field of the Python controller. The call will provide a reference to the currently executed controller as the one possible parameter.

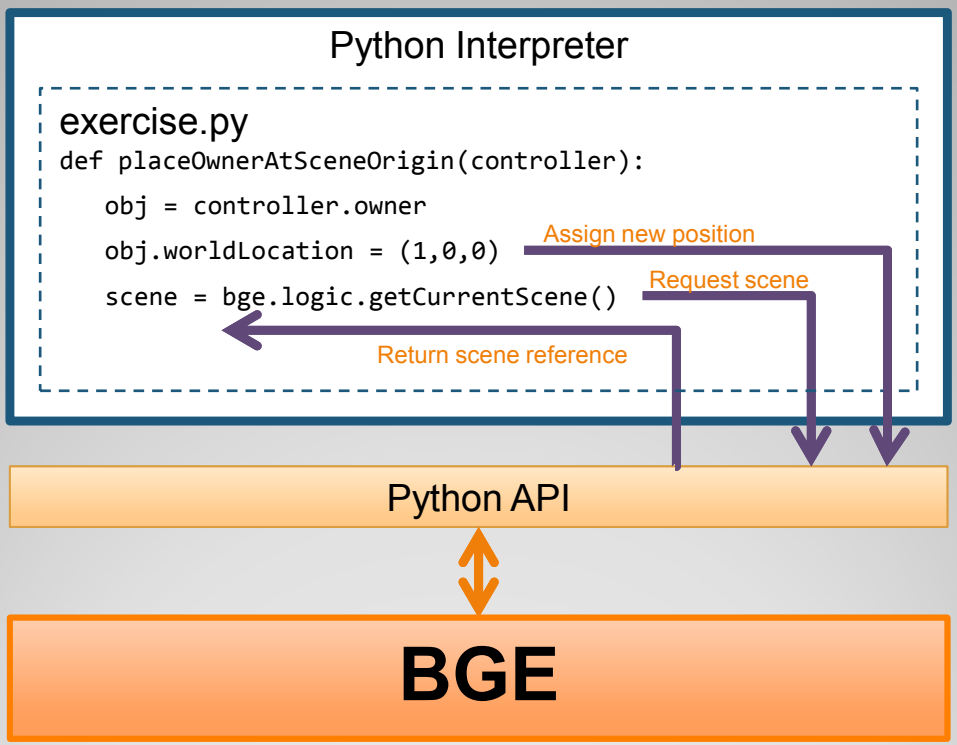
The BGE continues when the Python interpreter returns from this function call.



How it works

Calling just a Python module would allow you to perform custom processing. You can do nearly everything that you can do with a standalone Python application. This is pretty much but what you really need is to interact with the BGE.

Therefore the BGE provides you with an interface. It is called API (Application Programming Interface). Whatever you do with your Python code it does not run in Python. Python tells the BGE to perform the actions you requested.



BGE API

Lets continue with some important objects we get.

The first thing that you will find in nearly any BGE Python code is the current controller (often named "**cont**").

Remember you can have any number of controllers calling the same exact same code. The current controller is the controller that executes the current code.

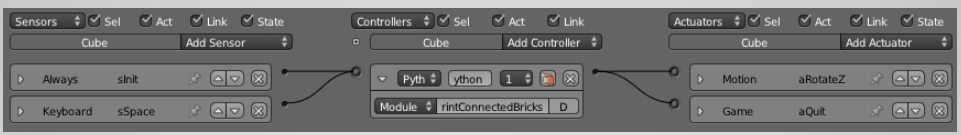
You can access all the connected logic bricks from the controller.

These are the

- connected sensors
- connected actuators

Lets print the names of all the logic bricks

```
def printConnectedBricks(cont):
            print ("sensors:", str(cont.sensors))
            print ("actuators:", str(cont.actuators))
```



the result is this:

```
Blender Game Engine Started
sensors: [sInit, sSpace]
actuators: [aRotateZ, aQuit]
```

Current Controller

As each other controller your code should evaluate the connected sensors first.

A very frequent asked question is:

- *“Why is my code executed twice when I press a key?”*

The answer is simple:

- “Because the controller executes on each trigger, this does not imply the sensors are positive or not”.

In other words your code has to care the sensors state. The current controller gives you a list of sensors. You have to pick the right one.

```
def printFirstSensorStatus(cont):
    firstSensor = cont.sensors[0]
    if firstSensor.positive:
        print ("The status of the first sensor is True")
    else:
        print ("The status of the first sensor is False")
```

Please keep in mind that the user might change the order of the logic bricks. You can access the sensors by name:

```
def printAlwaysSensorStatus(cont):
    sensor = cont.sensors["Always"]
    if sensor.positive:
        print ("The status of the ‘Always’ sensor is True")
    else:
        print ("The status of the ‘Always’ sensor is False")
```

Connected Sensors

If you want your code evaluate like an AND controller you can use this code:

```
def evaluateLikeAND(cont):
    if not allPositive(cont):
        print ("Not all sensors are positive")
        return
    print ("All sensors are positive")

def allPositive(cont):
    for sensor in cont.sensors:
        if not sensor.positive:
            return False
    return True
```

If you want your code evaluate like an OR controller you can use this code:

```
def evaluateLikeOR(cont):
    if not onePositive(cont):
        print ("Not one sensor is positive")
        return
    print ("At least one sensor is positive")

def onePositive(cont):
    for sensor in cont.sensors:
        if sensor.positive:
            return True
    return False
```

AND/OR Evaluation

Beside the status and the trigger sensors provide much more data. These are usually complex data that simple controllers can't process.

For example the near sensor provides a list of all sensed objects and the message sensor provides a list of all received messages.

With Python you can access this evaluation data.

```
def printSensedObjects(cont):
    sensor = cont.sensors[0]
    sensedObjects = sensor.hitObjectList
    if not sensedObjects:
        print ("no objects sensed")
        return

    print ("The sensed objects are", sensedObjects)
```

Have a look at the BGE API what you can get from the sensors.

Evaluation Data

After the sensors are evaluated your code can care the actuators. At this stage you should know if the connected actuators should be

- activated,
- deactivated or
- remain as they are.

Similar to the sensors you get a list of actuators. You can access them by index or by name:

```
def printNameOfFirstActuator(cont):
    actuator = cont.actuators[0]
    print ("The first actuator is", actuator.name)
```

The controller sends an activation signal to an actuator with:

```
def activateFirstActuator(cont):
    actuator = cont.actuators[0]
    cont.activate(actuator)
```

The controller sends an de-activation signal to an actuator with:

```
def deactivateFirstActuator(cont):
    actuator = cont.actuators[0]
    cont.deactivate(actuator)
```

If the status of the actuator should not change simply do nothing.

Be aware if an actuator receives activation and deactivation signals (e.g. from multiple controllers) the actuator acts like receiving an activation signal only.

Connected Actuators

If you want to activate all connected actuators at once:

```
def activateAllActuators(cont):
    if not allPositive(cont):
        return
    activateAll(cont)

def activateAll(cont):
    for actuator in cont.actuators:
        cont.activate(actuator)
```

If you want to deactivate all connected actuators at once:

```
def deactivateAllActuators(cont):
    if not allPositive(cont):
        return
    deactivateAll(cont)

def deactivateAll(cont):
    for actuator in cont.actuators:
        cont.deactivate(actuator)
```

BTW: You can only activate or deactivate connected actuators!

All Actuators

Configuration data

Each logic brick has configuration data. This is the data that you can set via GUI. With Python you can evaluate this configuration data on the fly.

I suggest to change preconfigured data only if you need to configure calculated data that is not known before the game starts.

For example if you want to track a dynamically added object. The TrackTo Actuator does not know the added objects and will remain to track the object set up at game start (which is usually the inactive object at an hidden layer).

This means you are forced to

- Identify the object to track
- Update the TrackTo Actuator with the identified object

```
def trackToSensedObject(cont):
    sensor = cont.sensors["Near"]
    sensedObjects = sensor.hitObjectList
    if not sensedObjects:
        return

    #select the first object in the list
    objectToTrack = sensedObjects[0]

    actuator = cont.actuators[0]
    actuator.object = objectToTrack
    cont.activate(actuator)
```

Have a look at the BGE API what you can get set at the sensors and actuators.

In a lot of cases it is important to know the game object with the current controller. The controller knows this game object:

```
def printCurrentObject(cont):
    if not allPositive(cont):
        return

    print ("The current object is", cont.owner.name)
```

All logic bricks have an owner. This is because sensors and actuators do not need to reside at the same object as the controller.

A game object provides a lot of data like position, parent, children etc..

Additional the game object allows various operations like positional changes, re-parenting etc. .

Some operations are executed immediately e.g. positional changes. Others are placed in the processing queue of the BGE and executed later e.g. playing an action.

In difference to an actuator all operations last one frame only. (A frame is one cycle in the GameLoop). If you need operations to be executed over a period of several frames you have to trigger your code at each frame of this period.

You can find all attributes of a game object in the BGE API documentation under KX_GameObject.

Current Game Object

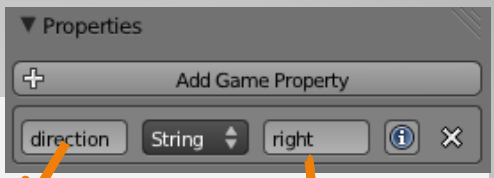
Now you know how to add BGE entry points to your module. You know how to call them from within the BGE logic.

It would became handy if you could assign parameters to the BGE entry point.

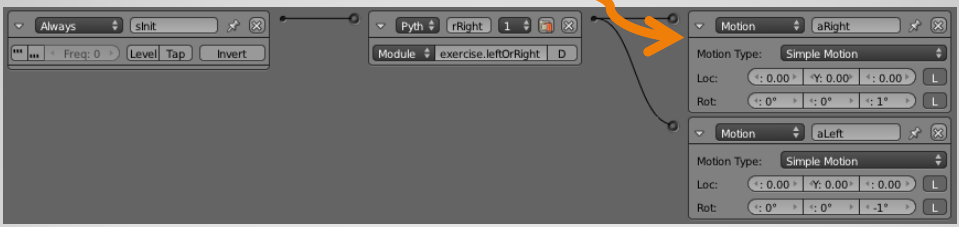
These parameters are the **properties** of the current game object.

The user can configure the properties with individual values. The Python controller can directly read the properties without the need of a Property Sensor (see next page).

Your code can act according to the given properties of the current game object.




```
def leftOrRight(cont):
    if not allPositive(cont):
        return
    direction = cont.owner["direction"]
    if direction == "left":
        cont.activate("aLeft")
    elif direction == "right":
        cont.activate("aRight")
```



Entry Parameters

Properties

Properties can be more than just simple configuration parameters.

Your code can modify properties, add and remove properties on the fly. You can see them as game object persistent variables. They live and die with their game object (if not explicit removed).

The user does not even need to see them. Properties created on the fly are **internal properties**.

Be aware internal properties are not recognized by property sensors!

Properties might not be present when reading them in this case your code exits with an KeyError.

To avoid that there are three methods to avoid such an error:

- Pre-check the existence with "**in**"
- Catch the KeyError with "**try**:..**except** KeyError:"
- Get the property value with "**get**()"

Remarks:

The method get() always returns an value. If the property is not present it does not add a property. The method returns the default value:

```
get(propertyName, defaultValue)
```

Without a default value get() defaults to None.

If you set a default value make sure it is a simple type. Because the default value will always be created regardless if used or not. This costs processing time and memory.

```
def readWithPreCheck(cont):
    own = cont.owner
    if "prop" in own:
        print("property", own["prop"])
    else:
        print("'prop' in",own.name,"does not exist")

def readWithTryExcept(cont):
    own = cont.owner
    try:
        print("property", own["prop"])
    except KeyError:
        print("'prop' in",own.name,"does not exist")

def readWithGet(cont):
    own = cont.owner
    value = own.get("prop")
    if value is not None:
        print("property", value)
    else:
        print("'prop' in",own.name,"does not exist")
```

You add or change properties by assigning a value to it. Existing property values will be overwritten.

```
def assignProperty(cont):
    own = cont.owner
    propertyName = "prop"
    own[propertyName] = "my value"
    print("'{0}' in {1} is '{2}'".format(
            propertyName, own.name, own[propertyName]))
```

Property access

The BGE allows you to have multiple independent scenes at the same time. From time to time it is necessary to access data of the scene where the current game object lives in.

Now we leave the first time the known world of our controller. For whatever reason the controller and the game objects do not know what scene they are living in.

So we need help from the BGE API. The API provides us the module **GameLogic**. Since 2.5 there is an alias called **bge** with the submodule **logic**. Both GameLogic and bge.logic refer to the same API. That means you can use whatever name you like most.

Before we can use this API module we need to make it known to you own module. So we add an import statement:

```
import bge
```

The usual place is the top of the module. Now we can access all public attributes of the module:

```
def printCurrentScene(cont):
    scene = bge.logic.getCurrentScene()
    print ("The current scene is", scene)
```

Again you can find all attributes at the Blender API documentation.

Current Scene

As written earlier a module will be loaded and initialized with the first call to one of it's BGE entry points.

This allows you to place some operations in the module you would like to be performed before your BGE entry point is called.

Please do not confuse

Module initialization with **Game Object Initialization**

Within the module initialization you can import other modules, run one time operations or define

- default values,
- "constants" (e.g. property names),
- classes,
- functions

Remarks:

A common mistake is to store a reference to the current scene or (even more worse) to the current controller in a module variable. (in the most cases this is just laziness). Any access to this variables returns the previously stored references which are incorrect if this BGE entry point is called by another game object or from another scene.

Better get the according references from the BGE API directly.

Module initialization

Scripts execute completely beginning with the first statement at the main level. Typical script code looks like this:

```
import bge
def init():
   #do some initialization
def doSomething():
    #do something

cont = bge.logic.getCurrentController()
own = cont.owner
if not “init” in own:
   init()
   own[“init”] = True
doSomething()
#do more processing
```

unnecessary

All you need to do is to place all processing code into a BGE entry point. Existing functions can be kept unchanged. New BGE entry points can be added.

```
import bge
def init(cont):
    #do some initialization
def doMoreProcessing(cont):
    own = cont.owner
    doSomething()
    #do more processing

def doSomething():
    #do something
```

Can be called separately

New BGE entry point

Unchanged but placed later

Script to Module

Performance

Processing logic with Python code is quite fast. It is not faster than the logic bricks when compared directly.

You should **avoid any unnecessary processing**. A lot of wasted processing can sum up to a game lag.

An important tool for performance increase are the sensors. Ensure that sensors avoid to trigger controllers when there is nothing to do. E.g. An always sensor with True pulse wastes a lot of processing time.

Example:

The controller is waiting for keyboard input. If the controller is triggered at each single frame the controller is executed unnecessary for 99% of the frames. You can't type that fast.

Better use a keyboard sensor to trigger the controller.

If your code knows it should not do anything (e.g. because a sensor is not positive) it should exit immediately skipping any other code.

Avoid long running loops (especially endless loops). You can use loops and recursions but they should be quite short.

When improving performance concentrate on frequent running code.

Don't be afraid to add further functions/methods.

You need to find a balance between efficient and readable code.

Recommendations

- Never expect a user to change your code!
- Place information important to the reader at the top of the module (Doc strings, Property names, Logic brick names, BGE Entry Points) – you know attention span is short!
- Avoid dependencies on game object names (better use property names)
- Avoid large comments – if you need them your code need a redesign
- Avoid short or meaningless identifiers – the reader should easily assume what the identifier means (e.g. a, o, l, g or main() have no natural meaning)
- Do not be afraid to type longer names – you have a computer and you can copy&paste. Other readers will thank you. (I'm sure you still want to understand what you wrote 3 weeks ago.)
- Check the PEP#008 Python Style Guide.
- Choose a naming convention and keep it consistent
- Keep operations that belongs together in one module (less files)
- Separate operations that do not belong together in separate modules (independent files).
- Avoid to mangle build-in names (e.g. print, range, dict)
- Avoid to give imported modules another name (e.g. import GameLogic as GL) – a reader knows GameLogic but not GL (and GL has no natural meaning).

I want to end the guide here. It should already provide you enough details to write your own Python modules.

I'm pretty sure you still have a lot of questions. You can get a lot of help at the Game Engine forums of blenderartists.org.

You will find a lot of tutorials and code snippets.

You can find me under blenderartists.org as Monster

Any feedback is welcome

Thanks for reading

And good luck

Thank you