



StoryScript Content Module (scm)

Tutorial & Documentation

Author: Martin Eden

Last Updated: 21/04/2007

*Making your own **adventure***

StoryScript puts in your hands the tools you need to create your very own text adventures, quickly *and* easily. This document will teach you everything from how to get started, to how to deal with the most advanced aspects of writing StoryScript adventures - or **modules**, as I call them.

StoryScript modules combine the power of a scripting language with a simplicity that's easy to get to grips with. If you've programmed before you'll find it even simpler, and if you haven't - don't worry! - this is much easier.

What you need

All you need to create your own adventure, is the Story Script engine (you can get the latest version off of the [website](#)) and a text editor. Notepad will work well for this, but you should avoid wordpad and Microsoft Word, as they won't save to plain text. My personal favourite is [ObjectEdit](#), but that's only available via Object Desktop.

Creating content

A StoryScript module consists of a number of files, of which the most important is the Content Module. This is the meat of the adventure, and contains all the actual content of the module (as opposed to graphical frills).

Everything in StoryScript is made up of three things: **Rooms**, **Objects** and **Doors**. These are then linked by the games built in commands, and the more complex **Scripts**. But we're getting ahead of ourselves. Let's quickly look at the three types of thing that we have:

Rooms

These are the containers of the game. The player travels between them, and interacts with the objects within them. Nothing in the game can exist outside a room. They can be a literal room in a castle or spaceship, or they can just be an area that you've split your world up into - like a part of a rainy street, a plain of grass, or coral reef that you swim up to. Ultimately, how large or small you make an area is up to you - and the player only 'sees' what you describe.

Objects

These are the stuff inside rooms that the player interacts with. Many can be picked up and carried around, others are fixed. They can be anything from a key, to a pool of water, to a book in a bookshelf, to the bookshelf itself.

Doors

These link rooms together, and allow you to travel from one to another. Some may present significant challenges in themselves, requiring the player to unlock them before they can be used.

With just these three you can build up complex environments to explore and interact with. So how do we go about doing that?

Keyword Content

Obviously you can't just tell the computer in English, "I want a room there, it should look like this...". Instead you have to use a formalized way of writing. First go to your StoryScript folder (probably C:\Program Files\StoryScript\) and open the modules folder. Create a new folder with the name of your new adventure. For example, "tutorial". Open this folder and create a new text file, called "tutorial.scm". This is a StoryScript content module.

First Keyword

Open up your new module and type

```
ModuleName:Tutorial
```

When you move on to creating your own modules you would use the name of your module instead of 'Tutorial'. This should always be identical to the name of the file and folder. There should not be a space before or after the colon.

So what have we done? StoryScript Content Modules are made up of a number of **Keywords**, followed by a **Value**. The keyword tells the computer what kind of information you're giving it, and the value is the actual information. In this case, we said we wanted to talk about the name of the module, and then, after the colon, we gave it a value - the name of the module, tutorial.

New Keyword

But we want to do more than just tell it what the game's called! We want to create a world. So it's time to learn another keyword. Type on a new line

```
New:Room
```

Simple as that! You just created a Room. If the value had been 'Object' or 'Door', then you would have created one of those instead.

Describing Things

Currently our Room is a blank slate. How do we give it a name and a description? *All* Things in StoryScript, have three properties in common: Name, Unique, and Description. Let's take a look at them:

On new lines below the last one, type:

```
Name:Alleyway  
Unique:Alleyway 1  
Description:You are standing in a dark alleyway. The paving stones are damp with rain, and a chill wind is blowing, rattling the lids of dustbins. The buildings above seem to be falling inwards, intent on
```

```
closing out the already narrow band of sky visible.
```

Breaking this down: The name of a thing is how you want the player to see it. When they want to interact with something they will use this value. For example 'examine alleyway'.

The Unique value, on the other hand, is *never* seen by the player. You use it in creating the module when you want to refer to this particular thing. It can be the same as the name but it can never be the same as any other thing's Unique identifier. In this case it seems likely that we might want to have more than one alleyway, so I called it Alleyway 1. If we have another alleyways it's *name* might also be Alleyway, but it's *unique* would have to be something different - like Alleyway 2.

Finally, the description: This is what the player will see when they enter this Room, or if they type 'look'. You should not put a new line in the description (by pressing 'Enter'). To check that it's all one line, turn text wrap off. (In notepad click 'Format', and then uncheck 'Word Wrap'). If the description is now all on one line then it doesn't have any breaks in it, and the engine will be able to understand your description.

Quick Reference Keyword Summary

- **Name** - Necessary, one value, name used for player interaction with the Thing.
- **Unique** - Necessary, one value, name used for 'behind the scenes' module interaction with the Thing.
- **Description** - Necessary, one value, text displayed when examine command used.

Adding a *Player*

So let's go try it out! There's just one more thing you have to do before this module will work: You have to tell the engine which room the player should start in. Go back up to the top of tutorial.scm, and add below the ModuleName line the following text:

```
CurrentRoom:Alleyway 1
```

This tells the engine that the starting room should be Alleyway 1. You notice that we use the Unique identifier for the room, rather than it's name.

Anyway, go fire up StoryScript. It should automatically detect your new module, and away you go. You'll see your description there. Your first StoryScripting!

Adding *Objects*

At the moment, your alleyway is rather boring. The player has nothing to interact with. So let's add some objects! How about a note on the floor? It could be an important clue in your adventure. Leave a line after your Room (not necessary, but it makes it clearer) and type:

```
New:Object  
Name:Note  
Unique>Note (Clue)  
Description:This is a scrap of paper, apparently dropped on the  
pavement. It is soaked and the paper is fragile, the ink beginning to  
run. Some of it is still legible.
```

So far things are familiar, right? Remember: Name, Unique and Description are common to all Rooms, Objects, and Doors.

Try running your adventure again. What do you see? Nothing different! Where's your

note? It doesn't appear because you didn't tell the game that it should be in the alleyway. Currently, it exists, but the player will never see it, because it's not anywhere - it has no location.

So time to add some more to your object:

```
AddObject:Alleyway 1
```

It's fairly obvious what this does, but there are some important lessons to be learned here: The keywords that make up a thing can go in any order, so long as you follow the following rules:

1. Keywords defining a Room, Object, or Door, always have to come after the 'New' keywords. We could have jumbled up AddObject, Name, Description, and Unique in any order, so long as the 'New' keyword came first.
2. The overall structure of the module should be Rooms, Objects, Doors, Scripts. Hopefully you can see why from the last blue box: If you hadn't described Alleyway 1 to the computer before Note (Clue), then the AddObject keyword would be trying to add your note to a Room that didn't exist yet.

So try running the module again! Hopefully you should now be able to see and examine your note.

You notice that the object is automatically listed in the 'You can see' section of the room description. However, there are more advanced things we can do. Try adding this line to your object:

```
Room_Extend:On the floor lies what at first seems to be just a scrap of paper. On closer inspection you can see it is a handwritten note.
```

Now, whenever the object is dropped these lines will be added to that room's description. To take the idea further, you can also add:

```
Hidden:True
```

Now the object will not be listed in the room by default. As well as letting you use room_extend as an alternative way of displaying objects, Hidden allows you to not explicitly tell the players about an object at all. Then they have to pick up from subtler clues.

What else can we do with objects? Let's look at another example. Leave a line and type:

```
New:Object  
Name:Trashcan  
Unique:Trashcan  
Description:These rusty trashcans are full of month old garbage. The stench is terrible and you don't want to go too close.
```

At the moment, the player could pick up the trashcan and carry it away from there. But what if you want to have an immovable object? Add:

```
Fixed:True
```

Finally, add the trashcan to the room.

```
AddObject:Alleyway 1
```

Try getting the trashcan. A default message is displayed, and - as simple as that - you have an immovable object. But what if you want something more interesting than the bog standard response? No problem! The keyword you want is 'No_Get'.

```
No_Get:Even if the trashcans weren't so heavy, would you really want to carry something that revolting around?
```

And that's objects done. Before we move on to doors, let's have a quick review of all the keywords for objects:

Quick Reference Keyword Summary

- **Room_Extend** - Optional, one value, text that extends the Room's description when the Object is in the Room
- **Hidden** - Optional, one true/false value, toggles whether Object is listed when Room is described, default: false
- **Fixed** - Optional, one true/false value, toggles whether Object is immovable, default: false
- **No_Get** - Optional, one value, text that is displayed when the player attempts to get a Fixed object, if empty a default message is displayed instead.

Adding Doors

But however many interesting objects you have, you're not going to want the player to sit in that one Room forever. So we need a way of getting from one to another. First, let's create another Room.

```
New:Room
Name:Subway Station
Unique:Station
Description:A small deserted subway station, dimly lit by overhanging
lights. There is a wind from the underground tunnel, and it sends
nearby pieces of paper and rubbish skidding across the dusty floor.
```

Remember that you should put this text with the other Room, before the Objects get described. Currently, this is cut off from the other Room, so let's make a Door.

```
New:Door
Name:Stairs
Unique:Subway Stairs
Description:These stairs are in poor repair, and lead into the
darkness of a subway station from the street.
```

So far, a Door is just like any other Thing in StoryScript. But now to create the link. Type:

```
AddExit:Alleyway 1|Down|D
AddExit:Station|Up|U
```

There's quite a lot new here, so let's take things slowly. Firstly: you can never have just one 'AddExit' keyword. You have to add the Door to two rooms, or it doesn't know where it should take the player when they step through.

Secondly, this is our first keyword that takes more than one value. AddExit actually has *three* values, and they are separated by the '|' character, or pipe. On a standard qwerty keyboard it should be next to 'Z', on the backslash ('\') key. To use it, hold shift and press backslash. Any time a keyword has multiple values, StoryScript separates them this way.

So what do the three values do? The first specifies which Room to add an exit to, same as the AddObject keyword. The next is the direction *from* that Room the Door is. To go down the stairs from the alleyway, the player would type 'Go Down'. The final value is a shorthand version of the direction. If you use this, then the player can just type 'Go d' (or whatever shorthand you assign). If you don't want to use a shorthand, just use the value 'Null'.

Try running it! You should be able to walk between your two Rooms easily. Also try examining a direction (like 'Examine down') or going somewhere by using the name of the door ('Go stairs'). You can often use these interchangeably.

There's more to be done with Doors, but that can wait until we've looked at Scripts.

Quick Reference Keyword Summary

- **AddExit** - Necessary, three values.
 1. Room Unique property. One of the rooms this door opens on to.
 2. The full name of the direction this door is in. eg. 'North', or 'Up'
 3. A shorthand form of the above property, generally the first letter. i.e. 'n' or 'u'. This can be the word 'Null' if don't want there to be a shorthand.

StoryScript *Scripts*

Scripts are little mini programs - sets of instructions that will only run when the right things happen. Script aren't 'Things' in the same way as everything else in the module, and are defined in a slightly different way.

So what can we do with Scripts? Pretty much anything! Scripts are the real power of the StoryScript module, and as people think of more things they want to be able to do, I'll add even more functionality. Let's start with a very simple example. Type:

```
New:Script  
ScriptID:Read note
```

This just creates a Script. Instead of 'Name' and 'Unique', Scripts have 'ScriptID'. This is because the player will never really know about the Script, so it doesn't need a name as such. Instead it needs a unique identifier that you can use to create the module. I generally try and choose an ID that will help me remember what the Script does.

Now on to the bulk of making a Script. There are three main parts to any script: Keys, Conditions, and Code - which, if you were an atrocious speller, you could call the three C's. They all have the right sound anyway!

Keys

Scripts are triggered when the player types in the right text. So what's the right text? That's what the Key keyword decides. In our case, we want to make a script that lets the player read our note. So they're probably going to try and type something like 'read note'. So let's add to our Script

```
Key:Read note
```

Keys are never case sensitive.

That's all well and good, but what if the player types 'read paper'? We can add options to our Key in the following way. Use this line instead:

```
Key:Read note|Read paper
```

Now if the player types either of those, then the Script's will accept it as a valid key. But the English language is a confusing thing, and this note is fairly obscured. What if a too clever player types 'decipher note'? We could, of course, simply use

```
Key:Read note|Read paper|Decipher note|Decipher paper
```

But this is starting to get a bit silly. There is a better way. Try the following:

```
Key:Read|Decipher  
Key:Note|Paper
```

Now, the script will only accept the player's action if it contains at least one of each Key set. So it must have either 'read' or 'decipher' *and* it must include either 'note' or 'paper'. You should notice that it only searches within the text for the individual words in any order. If the player typed 'The note is carefully deciphered', the Script would still accept this, as it contains one of each Key set.

Quick Reference Keyword Summary

- **ScriptID** - Necessary, one value, unique identifier of the script
- **Key** - Optional, list of values that denote words that trigger the script. This keyword can appear multiple times. If so, the script is triggered when the player's command contains at least one word from each key set, in any order.

Conditions

Even if the player types in the correct text, the Script may still not run. You can set Conditions, so that the script will only run in certain situations. For example, make it so that the player can only use a phone booth when they have coins in their inventory. Both Conditions and Code can be a little confusing, so read the following carefully.

Try typing:

```
Condition:IsObject:Inventory|Note (Clue)
```

What on earth is going on here? The thing you have to understand is that the value that goes with the keyword 'Condition', is itself actually an entire keyword/value pair.

The module is actually defined using three separate systems. The main module language, and then the two smaller Condition and Code languages. When the engine loads your module it acts upon the keywords in the module to create your adventure. But it just stores away the Conditions and Codes and only acts upon them when the Script is run.

So what does this particular Condition say? The condition keyword is 'IsObject'. This checks to see if an Object is present. It has two values. The first is the place to look for the Object. This can be 'Inventory', to look in the player's inventory; 'Here', to look in the current Room; '?', to look in both the inventory and the current Room, or the unique identifier of a Room, to look in a particular Room. The second value is the unique identifier of the Object you're checking for.

Unless all Conditions are met, the Script will not run. So in this case, unless the Player has the note in their inventory, they won't be able to try and decipher it.

Running Scripts

If you don't define any Conditions, the Script will run every time the player types the correct Keys in. If you have Conditions but no Keys, then every time the user type is anything the game will check to see if the Conditions are met. If they are, it'll run. If you don't give a Script any Conditions or Keys, it will never run.

Codes

So what happens when a Script runs? The game runs the stuff defined in Code. Again, there are all kinds of special Code keywords you can use. Later on, I'll list all of them. But for now, we want a way of making some text display when you read the note.

Try typing:

```
Code:Display:The paper is sodden and ripped, and much of the text is
either gone, or made indecipherable by running ink. After a few
seconds of squinting, you make out a few blurry words...
Code:Pause:
Code:Display:locked...check dustbins...danger
```

The first Code keyword we're using is 'Display'. It simply shows some text to the player. Notice that to have more than one code keyword, you simply have another Code line. The Code keyword 'Pause' makes the game halt until the player presses enter. Note that even though 'Pause' has no values, you still have to have a trailing colon.

One Script Down...

There you go! Your first Script is finished, and once you've got the hang of these you can do anything. One more finishing touch is needed:

```
AddScript:Room|Alleyway 1
```

The AddScript keyword has two values, and we'll see why soon. For now, just try running your module.

You should find that you can read (or decipher!) the note in the first room. The script will only run if the paper is in the player's inventory. Try experimenting with the conditions. Try replacing 'Inventory' with '?', and then go and drop the note in the station. Can you examine it while it's on the floor there? If you go back to the alleyway without the paper can you still examine it?

More advanced Scripting

Let's extend this example just a bit further, and then I'll give you the list of all the commands and let you try it for yourself. The note we can now read suggests that there is a key to be had - and that a little judicious searching in the dustbins will uncover it. We're going to be make this work. First of all, though, let's create the door we want to get through.

Locks

Doors in StoryScript have the property of being unlocked or locked, and default to being the former. That can be changed simply enough though. Add the following to your module:

```
New:Door
Name:Door
Unique:Station Door
Description:A plain door leading into the (now disused) station
office.
Locked:True
AddExit:Station|In|Null
```

Simple as that. Now if you try walking through the door you'll see a default message that tells you that way is impassable. To customize it slightly more you can add this keyword to your door:

```
No_Through:The door is locked and the key could be anywhere. The keyhole, however, is not as dusty as the surrounding door. Perhaps the key isn't far off?
```

Revealing Objects

We want to have a key hidden in the trashcans that will unlock this door. It needs to be revealed when we search the bins. We also don't want the player to be able to reveal the key unless the note - so until then the bins should still claim to very smelly and not do much else. How do we achieve all these things?

First off, create the key Object:

```
New:Object  
Name:Key  
Unique:Station Key  
Description:A plain and serviceable (if rather smelly) key
```

Note that I haven't added the 'AddObject' line. This key exists, but it's not accessible to the player yet.

Next we need the Script that reveals the key:

```
New:Script  
ScriptID:Find key in trash  
Key:Examine trashcan  
Code:AddObject:Here|Station Key
```

Two things in this Script should be new to you. First of all, the command AddObject. It is identical to the module keyword AddObject, except you cannot use '?'. You can still use 'Inventory', 'Here', or the unique identifier of a Room.

Secondly, the Key that triggers this Script is actually also a valid standard command in the game. The Script doesn't supplant the command, rather both will run, the standard command first.

Variables

Test this and see what happens. You should be able to examine the trashcan in order to reveal the key. However, it is clear we have two problems, that with your current knowledge of the engine will seem insurmountable: Firstly, the Script will run regardless of whether you've read the note. Secondly, and far worse, you can reveal the key more than once! In fact, if you pick up the key and go and drop it in the station, and then come back and examine the trashcan once more, the key will end up in two places at once.

Clearly we need some way of knowing what has happened previously to running the Script. The way we do this is with **variables**. Variables are values that Code keywords can set, and that Condition keywords can check for. The engine remembers them from room to room, allowing you to know whether certain things have happen before.

A variable consists of two things: A name, and a value. But you don't need to create them in the module as you do Things and Scripts. Simply when you check or set a variable, the engine will create it as it needs it. Until their value has been set, all variables equal 0 (zero).

So find your 'Read note' script and add this line:

```
Code:SetVar:Read Note|1
```

This sets the variable called 'Read Note' to equal 1. Similarly, add to 'Find key in

trashcan' the line:

```
Condition:CheckVar:Read Note|1
```

Now the reveal key Script will only run after the note has been deciphered - when 'Read Note' equals 1. By the same token you can add the following to the 'Find key in trashcan' Script, to make sure that it only runs once:

```
Condition:CheckVar:Found Key|0  
Code:SetVar:Found Key|1
```

So the first time this Script runs, it will set 'Found Key' to 1. After that it will never run again, as it requires 'Found Key' to be equal to 0.

Finishing Touches

There's just a little bit more to finish off the tutorial module. You need to make it so that the key can open the door. That's a simple Script, with a few new code and condition keywords:

```
New:Script  
ScriptID:Unlock Station Door  
Key:key  
Key:unlock door|open door|use  
Condition:IsDoor:Station Door  
Condition:IsLocked:Station Door  
Condition:IsObject:Inventory|Station Key  
Code:ToggleDoor:Station Door
```

The commands above should be fairly self explanatory by now. Note that there is no 'Unlock door' or 'Lock door' command. Instead you have ToggleDoor which sets it to be in the opposite state from its current, and the Condition keywords, 'IsLocked' and 'IsUnlocked'. So if you wanted to be able to lock the door again, it would be another Script.

You could add some text (using 'Display') to be displayed when the door is unlocked. However, the normal way to do this is using the Door keywords 'Locktext' and 'Unklocktext'. They are optional, and if you don't set them then there is a default message.

In this example we might add something like the following the Station Door:

```
UnlockText:The key turns easily in the lock and you hear the tumblers  
fall. Time to see what's on the other side.
```

And that's the tutorial done! If you want to continue you'll need to add another room, and add the Station Door to it so that it leads through.

I hope you found this tutorial helpful, and that you enjoy making text adventures in StoryScript. I'd love to hear about anything you make, so feel free to drop in over at my website: <http://www.geekroom.co.uk/> where you'll find StoryScript in the projects section.

Quick Reference Summary

Module Keywords

Keyword	Belongs to	Optional	No. of values	Description
ModuleName	Module	No	1	The title of the adventure module. Must be

				identical to filename and folder name.
CurrentRoom	Module	No	1	The unique identifier of the starting room.
BackgroundImage	Module	Yes	1	The path of an image to use the background
ColourPalette	Module	Yes	1	The path of a StoryScript palette file
New	Module	Yes	1	Specifies the beginning of either a Room, Object, Door, or Script declaration
Name	Thing ¹	No	1	The name of the Thing as it appears to the player
Unique	Thing ¹	No	1	The unique identifier used internally to refer to the Thing
Description	Thing ¹	No	1	The text displayed when the Thing is examined. Also displayed when entering a Room.
Hidden	Object	Yes	1	If set to 'True', the Object is not displayed in the Room's contents listing. Defaults to 'False'.
Fixed	Object	Yes	1	If set to 'True', the Object cannot be picked up. Defaults to 'False'.
No_Get	Object	Yes	1	Optional text to display when attempting to pick up a 'Fixed' Object. Otherwise default message.
Room_Extend	Object	Yes	1	Text that extends the Room's description when the object is in the room.
AddObject	Object	Yes ²	1	Unique identifier of the Room this Object should be in.
Locked	Door	Yes	1	If set to 'True', the Door is impassable. Defaults to 'False'.
LockText	Door	Yes	1	Text displayed when the Door is locked. Otherwise default message.
UnlockText	Door	Yes	1	Text displayed when the Door is unlocked. Otherwise default message.
No_Through	Door	Yes	1	Text displayed when the player attempts to go through the locked Door. Otherwise default message.
AddExit	Door	No ³	3	1: The Room to add the exit to. 2: The direction the door is in. 3: A shorthand form of the direction. Can be set to 'Null' if not needed.
ScriptID	Script	No	1	The unique internal identifier for the Script.
Key	Script	Yes ⁴	X	X words or phrases that will trigger off the Script. If one or more of these appears in the player's command at some point then the Script will run. If there is more than Key statement the input must contain at least one Key value from each Key statement.
Condition	Script	Yes ⁴	1	A valid Condition statement (see below)
Code	Script	No	1	A valid Code statement (see below)
AddScript	Script	No	2	1: The kind of Thing to add the Script to (either 'Room' or 'Object'). 2: The unique identifier of the thing the Script is being added to.

¹ A Room, Object, or Door

² Necessary for the object to be accessible, unless this is enabled by a script later on

³ Needed twice for the door to function correctly

⁴ Must have at least one of either Key or Condition for the Script to function

Condition Keywords

Keyword	No. of values	Description
IsObject	2	1: Where the Object can be. This can be 'Here' (the current Room), 'Inventory', '?' (for either of the previous places), or the unique identifier of a Room. 2: The unique identifier of the Object. Runs if the Object is present.
IsDoor	1	The Door to be checked for in the current Room. Runs if the Door is present.
IsLocked	1	The Door to be checked. Runs if the door is locked.
IsUnlocked	1	The Door to be checked. Runs if the door is unlocked.
CheckVar	2	1: The unique name of the variable. 2: The value it must be equal to. Runs if the variable is set to the value.

Code Keywords

Keyword	No. of values	Description
Display	1	Text to be output to the screen
Death	0	Causes the game to end with a gameover message
Pause	0	Prompts the player to press 'Enter'
AddObject	2	Adds an Object to the specified place. 1: Where the Object should be placed. This can be 'Here' (the current Room), 'Inventory', or the unique identifier of a Room. 2: The unique identifier of the Object.
AddDoor	4	Adds an exit to the specified Room. 1: The Room the exit should be added to. 2: The Door being used. 3: The direction the door is in. 4: The shorthand form of the direction. Can be 'Null'.
RemoveObject	1	Removes the given Object from either the Room its in, or the inventory, depending on its current location. 1: The unique identifier of the Object.
RemoveScript	2/0	Removes the given Script from the Room its currently in. 1: The Room to be removed from. 2: The Script to be removed. Optionally this may be called with no values, which causes the Script to remove itself from its current Room.
SetVar	2	Sets a variable to a particular value. 1: The unique name of the variable to be set. 2: The value it should be set to.
ToggleDoor	1	Toggles the specified Door between locked and unlocked. 1: The unique identifier of the Door.
FalseInput	1	Makes the game respond as if the user had just typed in a particular command. 1: A command such as the user might type in, eg. 'look'.