Python in NakedMud: More Than a Scripting Language

Geoff Hollis
hollisgf@email.uc.edu

November 18, 2006

Contents

1 Introduction

The original motivation for embedding Python into NakedMud was as a scripting language. However, it quickly became evident that Python could serve greater purposes for the codebase. It is easy to load Python modules and packages from within a C application, and with basic access to the mud's functions and data, Python can basically act as a secondary language for programming in NakedMud. This tutorial will cover how Python works, with respect to NakedMud. We'll do this by writing example triggers. However, it should be noted that Python is used for much more. For example: rooms, objects, and characters are actually *programmed* as Python scripts. In addition, new modules can be written in Python and added to the mud, just like in C. The information here can just as easily be applied to building or programming new modules in python.

We do cover some of the Python basics like importing and initializing a package, *this is not a tutorial on Python*. If you have never programmed in Python before, it is suggested you pick up an introductory book on Python, or do some Python tutorials to get a hang of the language's syntax. http://www.python.org is a great source for guides, tutorials, and documentation on the Python programming language.

2 Triggers

Python was originally intended to function as a scripting language for writing triggers in NakedMud (similar to how DG Scripts work for CircleMUDs). Python was chosen because of it's relatively clear and simple syntax, and easy learning curve, even for someone with minimal or no programming experience - the types of people (i.e. builders) we might expect to see using NakedMud's scripting language the most. A handful of trigger types are provided with the core NakedMud release. These range from triggers that are run when an object or mobile is loaded, triggers that run when someone speaks or enters a command, triggers when someone enters or leaves a room, and a handful of others. Here we will cover the basic trigger types, and provide sample code for each.

As a quick detour before we dive into the trigger types, it would be good to know how to edit and attach triggers. Assuming you are using NakedMud's standard text OLC, you can create triggers with the tedit <key> command. You can attach triggers to mobiles, objects, and rooms by editing the thing (i.e. medit <key>, oedit <key>, redit <key>, respectively) and then going to the Trigger menu.

2.1 Reset Triggers

Reset triggers can be attached to rooms, and are run every time the zone a room belongs to is reset. Every reset trigger has a variable called *me*, which is a pointer to the room being reset. Below is an example trigger that will hopefully elucidate ways in which initialization triggers can be used:

```
##############################################################################
# Reset trigger no. 1 - resetting a room. This trigger will attempt to load a
# mobile with with key 'mysty' into the room if she exists nowhere else in the
# game. Once mysty is handled, the trigger will check if there are any tables
# in the room. If there are less than 4 tables in the room, tables will be
# loaded until there are 4.
####

# Load Mysty if there are no copies of her in the game
if count_mobs("mysty@examples") < 1:
  load_mob("mysty@examples", me)
```

```
# Load a few tables into the room
num_tables = count_objs("table@examples", me)
while num_tables < 4:
  load_obj("table@examples", me)
  num_tables = num_tables + 1
```

2.2 Speech Triggers

Speech triggers are intended for use with mobiles and room. Any time someone uses the 'say' or 'ask' command in the same room as a mobile with a speech trigger, the trigger will be executed. Any time someone uses 'say' in a room with a speech trigger, the trigger will be executed. Speech triggers have 3 variables: *me, ch, and arg*. The *me* variable points to the mobile or room with the attached trigger. The *ch* variable points to the character doing the speaking. The *arg* variable is the actual speech that started the trigger.

```
##############################################################################
# If this trigger is attached to a mobile, any time someone in the same room
# as the mobile says hello hi, or how do you do, the mobile will reciprocally
# greet the original speaker.
####

if arg == "hi" or arg == "how do you do?":
  # wait a second, and then say hi back
  me.act("delay 1 say hello, " + ch.name + ". " + \
         "What brings you to " + me.room.name + "?")
```

2.3 Drop Triggers

There are two ways in which drop triggers can be used. The first case is attaching a drop trigger to an object. Whenever the object is dropped, the trigger will go off. The second case is attaching the trigger to a room. In this case, whenever something is dropped into the room, the trigger will execute. Drop triggers have three arguments: *me, ch, and obj*. The me variable is a pointer to whatever the trigger is attached to (object being dropped, or room the object is dropped to). The *ch* variable is a pointer to the character doing the dropping. The *obj* variable is a pointer to the object being dropped. In the case that the drop trigger is attached to

an object, the value of this variable is the same as the value of *me*. Below are two example drop triggers - one for a room and one for an object:

```
##############################################################################
# Drop trigger no. 1 - dropping objects. This trigger is attached to an object
# in order to 'curse' it. Any time a player drops the object, it will return
# itself to that player's inventory!
####

# send messages out to let people know the item is returning
message(ch, None, me, None, False, "to_char", \
    "$o resists being released! As soon as you let go of it, it flies " \
    "right back into your hands!")
message(ch, None, me, None, True, "to_room", \
    "As soon as $n lets go of $o, it flies right back into $s hands!")

# transfer the item back
me.carrier = ch



##############################################################################
# Drop trigger no. 2 - rooms with dropped objects. When an object is dropped to
# any room with the attached trigger, a sticky glue will fuse the object to the
# ground, preventing it from being moved.
####

# let everyone in the room know what is happening
me.send("As soon as " + obj.name + " touches the ground, a sticky substance " \
        "bubbles up from the floor and fastens it in place!")

# set the notake flag, so noone can get the object
obj.bits = 'notake, ' + obj.bits
```

2.4 Give Triggers

There are two ways in which give triggers can be used. Case one is when the give trigger is attached to an object. When this happens, the trigger will execute any

time someone gives the object to another person. Case two is when the give trigger is attached to a person. When this happens, the trigger will execute any time the mobile gives an object. Give triggers have four variables: *me, ch, obj, and recv*. The *recv* variable is the person the object is being given to. The *me* variable is the thing that the trigger is attached to. The *ch* variable is the character doing the giving. The *obj* variable is the object that is being given. In the case that the give trigger is attached to an object, *obj* will be the same value as *me*. In the case that the give trigger is attached to a mobile, *ch* will be the same value as *me*. Below is a simple give trigger that can be attached to an object:

```
##############################################################################
# Give trigger no. 1 - give trigger for an object. Like the demo drop trigger,
# this trigger will 'curse' an object. Any time the object is given by someone
# to another person, the object will return to the giver. If possible, it will
# equip to the person. Failing that, it will go back to the person's inventory.
####

# let everyone in the room know what is happening
message(ch, None, me, None, False, "to_char", \
    "$o resists being released! As soon as you let go of it, it flies " \
    "right back into your hands!")
message(ch, None, me, None, True, "to_room", \
    "As soon as $n lets go of $o, it flies right back into $s hands!")

# transfer the item back, first trying to equip it. If that fails, then just try
# to put it to our inventory
try:
  ch.equip(me)
except:
  me.carrier = ch
```

2.5 Receive Triggers

Receive triggers are very similar to give triggers. When a person gives someone else an object, a receive trigger is also executed. Receive triggers have 3 variables: *me, ch, and obj*. Me is the person receiving the object. Ch is the person giving the object. Obj is the object being given. Below is a simple receive trigger:

```
################################################################################
# Receive trigger no. 1 - receiving an item. If the item being given is a pint
# of alcohol, the receiver thanks the giver.
####

# check the vnum to see if it's a pint of beer
if obj.isinstance("pint@examples"):
  # let's set this up as a delayed action to add a bit of spice.
  def act_thanks(ch, data = None, arg = None):
    ch.act("say thanks, mate!")

  # queue up the action so it will execute in 1 second. We can provide
  # 3 more args if we so choose. The first is another method that would
  # execute if the action was interrupted. This method takes the same form as
  # act_thanks. The second is any arbitrary variable. The third is a string
  # argument. They would fill in the 2nd and 3rd variables to the completion or
  # interrupt function, when the action is completed or interrupted. None is an
  # acceptable value for any of these 3 optional arguments.
  me.startAction(1, act_thanks)

  # all of this was equivalent to: me.act("delay 1 say thanks, mate!")
```

2.6 Enter Triggers

Enter triggers can be used in two ways. The first is to attach the trigger to a room. Any time a character enters the room, the trigger will execute. The second way is to attach the enter trigger to a mobile. Any time someone enters the same room as the mobile, the trigger will execute. Both types of enter triggers have two variables assocciated with them: *me and ch. Me* is the room or person the trigger is attached to. The *ch* variable is the person who is doing the entering. Below are two example enter scripts - one for a room and one for a character:

```
################################################################################
# Enter script no. 1 - entering a scripted room. If a character stays in the
# room for more than 5 seconds, he will be randomly teleported to another room.
####

# We'll need a function to handle the delayed transportation event. Owner is
```

```
# the room that is doing the teleporting, and ch is the person being moved
def event_func(owner, ch, arg = None):
  # if we've stayed in the same room, transfer us somewhere new
  if ch.room == owner:
    ch.send("You reel in dizziness for a moment, and suddenly the " \
            "world looks very different.")
    message(ch, None, None, None, False, "to_room", \
            "$n disappears in a puff of smoke!")
    ch.room = random.choice(["by_stage",        \
                             "tavern_entrance", \
                             "the_bar",         \
                             "the_fireplace"]) + "@examples"
    ch.act("look")


# set up a new event to trigger in five seconds.
start_event(me, 5, event_func, ch)



################################################################################
# Enter trigger no. 2 - greeting characters. This script is intended to be
# attached to mysty@examples. When she meets someone for the first time,
# she will greet that person and flag him/her as having been met.
####

# check to make sure that:
#   a) this is a PC
#   b) we've never met the person before
if ch.is_pc and not ch.hasvar("met_mysty"):

  # send different message to different gendered characters
  if ch.sex == "male":
    me.act("say Well hello there, darlin'. I've never had the pleasure of " \
           "seeing your pretty face around these parts. If you need a hand, " \
           "or perhaps a drink, just ask me for help.")
  else:
    me.act("say Welcome to the Stuck Swine! I hope it's to your liking here. " \
```

```
    "If you need a hand, or perhaps a drink, just ask me for help.")

  # set a variable so we know we've met Mysty before
  ch.setvar("met_mysty", 1)
```

2.7 Exit Triggers

Like enter triggers, exit triggers can be attached to rooms and mobiles. If an exit trigger is attached to a room, the trigger will execute any time someone leaves the room. If the trigger is attached to the mobile, it will execute any time someone leaves the room the mobile is in (excluding the mobile itself). Exit triggers have four variables associated with them: *me, ch, ex*. The *me* variable points to the thing that has the trigger attached to it. The *ch* variable is the person leaving. The *ex* variable is the exit being left through. Below are some sample exit scripts:

```
##############################################################################
# Exit trigger no. 1 - leaving a room. When a character leaves any room with the
# attached trigger, he will trip and fall, spilling all of his items over the
# ground in the old room.
#####

# assign the inv to a local variable so it doesn't have to be rebuilt
# when we need it the second time
inv = ch.inv

# depending on whether or not we have items, send different messages
if len(inv) > 0:
  ch.send("{yAs you leave " + me.name + ", you trip and drop all of your " \
          "items on the ground!")
  message(ch, None, None, None, False, "to_room", \
          "$n trips and falls as $e leaves " + me.name + ", dropping all " \
          "of $s items in the process!")
else:
  ch.send("You trip and fall!")
  message(ch, None, None, None, False, "to_room", \
          "$n trips and falls as $e leaves " + me.name + "!")

# now, transfer all of the items to the ground
```

```
for obj in inv:
  obj.room = me



################################################################################
# Exit Trigger no. 2 - following characters. When attached to a mobile, this
# trigger will make that mobile follow the first person who leaves the room it
# is in. Once the mobile finds a person to follow, it will continue to follow
# that person every time he tries to move.
#####

# See if we've found someone to follow yet. If not, pick this person
if not me.hasvar("char_to_follow"):
  me.setvar("char_to_follow", ch.uid)

# See if the person leaving the room is our target. If it is, follow
if me.getvar("char_to_follow") == ch.uid:
  # go through the same exit that the person we are following did
  for name in me.room.exnames:
    if ex == me.room.exit(name):
      me.act(name)
      break
```

3 Modules

As the title of this article suggests, Python is used for more than just scripting. In fact, it can be used for *real* programming as well. About 95% of the things you can do in C, you can do in Python if you wish. You might say that NakedMud is *bilingual*; it can be arbitrarily extended in two different languages. However, there is a caveat to this feature: it is a rather cumbersome process to give C access to aspects of the mud written in Python. It is slightly less cumbersome (but cumbersome nonetheless) to give Python access to aspects of the mud written in C. The moral of this caveat is that, if you intend to do any programming in Python, you will seriously want to consider how the code you write will interact with code you write in C. For example's sake, let's consider writing a combat system in Python. Depending on the theme of your mud, a combat system will probably be expected to interact with a magic system, a clan system, a skill system, etc... choosing to write the combat system in Python will pigeon-hole you into also writing these other systems in Python; it is much more cumbersome to translate from Python to C than it is to translate from C to Python. Before writing anything in Python, you will want to ask yourself: what other aspects of my mud will interact with this system, and do I want to write these aspects in Python as well?

So the practical question arises: when do I want to program something in Python, and when do I want to program something in C? There are many ways you can answer this question - some good, some bad. I will outline a few I think are good answers to the question. Of course, I am not a programming guru by any means, so approach my point of view critically with your own knowledge.

The first situation where you might want to program in Python is when you are *prototyping* a new idea. Programming in Python is much quicker than programming in C. This makes Python a very useful tool for testing out a new idea you have. If you are skeptical as to whether the idea will work well in practice or not, and you would not like to risk the time writing it in C if you are not absolutely certain it will work, you can turn to Python to quickly get it prototyped for evaluation. If you decide you like how it works, you can then translate the code into a NakedMud C module.

The second situation is when you are confident the code you intend to write will never interact with any other aspect of the mud. For instance, I am currently working on an inter-mud communication module (i.e. people on different NakedMuds will be able to communicate with each other). There is little reason to suspect such a feature

would ever need to interact with anything else in a mud. The socket handling is also much easier to deal with in Python than it is in C, so why not do it in Python? When you are certain what you want to do will never have to interact with any other aspect of the mud, writing it in Python is probably a good choice.

Some people find it useful to separate the *driver* and the *library* of a mud. In our situation, a good way to define the library of NakedMud would be all of the core functions, action and event handlers, game mechanics, data structures, etc... that are central to the functioning of the mud. The driver would be everything else (e.g. player commands, mobile AI, minigames, etc...). The library would be similar to your game engine, and the driver would be a collection of all the ways in which things instantiated within the gameworld interact with the game engine. Drawing the "what should I write in C and what should I write in Python" line at the separation between the library and the driver of your mud would probably work well. Your mud library should not need to access your mud driver, so you should never need to translate between languages beyond the point of giving Python access to the core functionality of your MUD. Such a setup would also give you the opportunity to do a great deal of your programming Python, which will undoubtedly speed the development process.

The fourth situation when you might want to write something in Python is when you commit yourself to writing *everything* in Python. Even though the majority of NakedMud has been written in C, it is quite conceivable at this point in its development for a person to completely switch languages and do everything else in Python. This is an option some people might want to entertain - especially those who are not well-versed in C programming. Python is a much easier language to program in, and quite a bit more newbie-friendly than C. Simple programming errors are almost always non-fatal to your program (instead, an exception will be thrown and reported). Python code can also be dynamically reloaded into the mud, cutting down on the need for reboots or copyovers. Undoubtedly, there will be a need to do a little bit of work in C, but this will largely concern revealing more bits of NakedMud to Python so you can continue to do the 'real' programming in Python.

Hopefully these opinions will help you decide how you will answer the question of "what will I write in Python and what will I write in C"? These are by no means the only ways you might want to answer the question - they may not even be the best ways, or consistently reliable ways. I am by not an expert (or even a veteran) at answering these types of questions. Heck, I might even have the terminology wrong! But, the meaning is there and (hopefully) intelligible. I warn you, give this topic some serious consideration before you commit yourself to any decision. Outline what

it is exactly you wish to do with your mud, and try to identify any potential problems that might arise from writing any aspect of it in Python or C.

3.1 Loading Python Modules and Packages

If you've decided you would like to write a module in Python, adding it to NakedMud is a relatively painless process. Simply drop it into the *lib/pymodules* directory. The next time your mud restarts, the module will automagically be loaded into NakedMud. You can also load modules into the mud while it is running through use of the *pyload* command. Specify the name of a module, and it will be (re)loaded into the mud. You can also (re)load Python packages (i.e. directories of python modules) in the same way. WARNING: This may be very dangerous if you are loading or reloading a module or package that installs auxiliary data on something.

3.2 Auxiliary Data and Storage Sets

In the article entitled Extending NakedMud: An Introduction to Modules, Storage Sets, and Auxiliary Data, I (hopefully!) pointed out how important it is to understand the mechanics of auxiliary data and storage sets. They are fundamental aspects of of the codebase; without understanding how they work, you will not get very far in the development process. The aforementioned tutorial only covered their usage on the C end of things. Storage sets and auxiliary data are equally important on the Python end of things. However, the syntax employed to use them is slightly different than it is in C. that is what this section intends to cover - syntactic differences. For an actual tutorial on how/why to use auxiliary data and storage sets, you are referred to the aforementioned tutorial. All of the concepts covered in that article (sans actual implementation) are equally applicable to Python programming as they are to C programming. As an exercise to the reader, it may be worthwhile to try and implement the mail module covered in the C tutorial on storage sets and auxiliary data as a Python module.

A list of the methods available to Storage Sets is available in the reference section of this manual. Below is a short example of how one might go about implementing auxiliary data in Python.

```
##############################################################################
#
```

```
# auxiliary_example.py
#
# Provides a simple example of how one might install and interact with
# auxiliary data in python. This module installs a new piece of auxiliary data,
# and sets up two new commands that allow people to interact with that auxiliary
# data.
#
##############################################################################
from mud import add_cmd
import auxiliary
import storage

# Example auxiliary data class. Holds a single string variable that
# people are allowed to get and set the value of
class ExampleAux:
  # Create a new instance of the auxiliary data. If a storage set is supplied,
  # read our values from that
  def __init__(self, set = None):
    if not set:
      self.val = "abcxyz"
    else:
      self.val = set.readString("val")

  # copy the variables in this auxiliary data to another auxiliary data
  def copyTo(self, to):
    to.val = self.val

  # create a duplicate of this auxiliary data
  def copy(self):
    newVal = ExampleAux()
    newVal.val = self.val
    return newVal

  # returns a storage set representation of the auxiliary data
  def store(self):
    set = storage.StorageSet()
    set.storeString("val", self.val)
    return set
```

```
# allows people to peek at the value stored in their ExampleAux data
def cmd_getaux(ch, cmd, arg):
  aux = ch.getAuxiliary("example_aux")
  ch.send("The val is " + aux.val)

# allows people to set the value stored in their ExampleAux data
def cmd_setaux(ch, cmd, arg):
  aux = ch.getAuxiliary("example_aux")
  aux.val = arg
  ch.send("val set to " + arg)

# install our auxiliary data on characters when this module is loaded.
# auxiliary data can also be installed onto rooms and objects. You can install
# auxiliary data onto more than one type of thing by comma-separating them in
# the third argument of this method.
auxiliary.install("example_aux", ExampleAux, "character")

# add in our two commands
add_cmd("getaux", None, cmd_getaux, "unconcious", "flying", "admin",
        False, False)
add_cmd("setaux", None, cmd_setaux, "unconcious", "flying", "admin",
        False, False)
```

As can be seen, auxiliary data in Python is not that much different from auxiliary data in C. The only big difference is that auxiliary data in Python takes the form of a class with methods rather than a struct with functions. There is also no delete method (Python does its own garbage collection). Nor is there a read method. Instead, storage sets can be provided as an optional argument to the auxiliary data __init__ method. Hopefully it is obvious that, aside from these superficial differences, auxiliary data in C and Python is exactly the same!

4 Module, Method, and Variable Reference

4.1 Mud Module

This module contains a smattering of miscellaneous methods that do not really fit in anywhere else. For importing, this module's name is simply *mud*.

4.1.1 Methods

def set_global(key, val)
This function maps a key to a val in the global variable table. Both keys and vals can be any arbitrary Python object.

def get_global(key)
Returns the val that key maps to in the global variable table. If key does not have a corresponding val, returns None.

def erase_global(key)
Removes the given key and its corresponding val from the global variable table.

def add_cmd(name, sortAs, func, minPos, maxPos, grp, mobOk, interrupts)
Adds a new command to the game. *Name* is the thing players will have to type to activate this command. Commands are normally sorted in the command table by their name, but there are some situations where this will lead to undesirable shortforms of commands. So, for instance, "n" activating "nod" instead of "north". The sortAs variable can be used to force a shortform on a command (i.e. "n" for "north"). Normally, this value should be *None*. The func variable is a pointer to the method that instantiates this command. Methods that are acting as commands should have 3 parameters: *ch, cmd, and arg*. The minPos and maxPos are string-form positions that the character is allowed to use the command from. The grp variable is the user group this command belongs to. The mobOk variable is a boolean value that says whether or not non-pc characters can use this command. The interrupts variable is a boolean value that says whether or not this command interrupts a character's action handler.

def message(ch, vict, obj, tobj, hideNoSee, to, message)
Sends a message out to the specified types of people. Works identically to the message command defined in the inform.h header file, with the exception that to is a comma-

spearated list of scope types this message has. To can include *to_room, to_char, and to_vict*.

def format_string(string)
Returns a new copy of the string that has been formatted to act as a description, helpfile, and other blocks of text that should be in a paragraph form.

def extract(thing)
Extract an object or mobile from the game. Note that this method is extremely volatile, and should never be called from a script, as the function that called the script will more likely than naught still need the thing being extracted. Use with care!

def ite(outcome, if_true, if_else)
ite (if/then/else) is a functional form of the an if/then/else block. This has been supplied so that builders can embed scripts within descriptions (scripts must be surrounded by [and]).

4.2 Character Module

A module containing character-related utilities (e.g. building a list of all characters in game, loading a new character instance to game, counting occurrences of a character instance in game, etc...). This module is also home to the Char class, which is the Python representation of NakedMud's PCs and NPCs. For importing purposes, this module's name is *char*.

4.2.1 Methods

def char_list(self)
Returns a list of all the characters currently in the game.

def socket_list(self)
Returns a list of all the characters with attached sockets currently in the game.

def load_mob(self, key, where, position = "standing")
Load an instance of the mobile with the given key into the game. Where can either be a room or room key, or an object that is a piece of furniture. The character's position can also be set at loadtime, to one of the valid positions. By default, characters load

to the standing position. This method returns a pointer to the mobile it loads into the game.

def count_mobs(self, key, where = char_list())
Returns a count of all the mobiles that are an instance of the mob key in the given scope. Where can be a room or room key, or a piece of furniture. If where is None, every character in the game is checked.

4.3 Char Class

This class is a Python wrapper around the CHAR_DATA C struct. Contains various methods and variables for interacting with the actual character that this Python class wraps around. The Char class is contained within the char module.

4.3.1 Methods

def __init__(self, uid)
The initializer for the Char class. Takes in a uid for the character we wish to create a Class around. Note that new characters cannot be loaded into game with this initializer. To do that, you will want to look at the load_mob function.

def attach(self, script_vnum)
Attach a script with the given vnum to the character.

def detach(self, script_vnum)
Detach a script with the given vnum from the character.

def send(self, mssg)
Send the character a message. A newline will be appended to the end of the message.

def sendaround(self, mssg)
Send a message to everyone in the character's room except for the character himself. Appends a newline to the end of the message.

def act(self, cmd)
Force a character to perform the specified command

def setvar(self, key, val)
Set the value of a character's variable. Keys must be strings, and vals can be strings,

ints, or doubles.

def getvar(self, key)
Returns the value associated with the given key in the character's variable table. Keys must be strings, and vals can be strings, ints, or doubles. If a character does not have a variable with the given key, None is returned.

def hasvar(self, key)
Returns true if the character has a variable with the given key.

def deletevar(self, key)
Delete a variable with the given key from the character's variable table.

def equip(self, object, where = None)
Equips a character with the specified object. If where is not supplied, the item will equip to the first available positions. If where is supplied, the object will try to be equipped to the given position names.

def isActing(self)
Returns true of the character is in the midst of performing a delayed action.

def startAction(self, delay, complete, interrupt = None, data = None, arg = None)
Start a new delayed action of the specified length. The complete parameter is a pointer to the method that will be executed when the delay finishes counting down. This function must take 3 arguments: a pointer to the character performing the action, a pointer to data (even if it is None), and a pointer to arg (even if it is None). Interrupt is an optional function of the same form that is called if the character's action is ever interrupted. Data is an optional variable of any type. Arg is an optional string variable.

def interrupt(self)
Interrupts the character's current action, if he has one in waiting.

def getAuxiliary(self, key)
Returns the character's auxiliary data with the given key. If the character has no auxiliary data with the given key, None is returned.

def cansee(self, thing)
Returns whether or not the character can see the given thing (which can be another character, or an object).

def page(self, string)
Sends a long piece of text to the character in a way that allows the person to go back

forth through the text (like a book).

def isinstance(self, prototype)
Returns whether or not the character is inherited from the given prototype.

4.3.2 Variables

Characters have various variables associated with them. Some are settable, and some are not. below is a list of those variables, and details of what each is for.

Variable	Settable	Description
inv		A list of objects in the character's inventory.
objs		Same as inv
name	√	The character's name
mname	√	The character's name when multiple instances are in the room.
desc	√	The paragraph description of the character when looked at.
rdesc	√	The description of the character when seen in a room.
mdesc	√	The character's room description when multiple instances are in the room.
keywords	√	A comma-separated list of named the character can be referenced by.
sex	√	The character's gender, in string form. Will be male, female, or neutral.
gender	√	same as sex.
race	√	The character's race, in string form.
pos	√	The character's current position, in string form.
position	√	Same as pos
room	√	A pointer to the room the character is in.
last_room		A pointer to the previous room the character was in. Or None.
on	√	A pointer to the piece of furniture the character is currently on. To remove a character from a piece of furniture, set on to None.
uid		The character's unique identification number.
prototypes		A comma-separated list of prototypes the character inherits.
is_npc		True if the char is an NPC. False otherwise.
is_pc		True if the char is a PC. False otherwise.
hisher		"his" if the character is male, "her" if the character is female, and "its" if the character is genderless.
himher		"him" if the character is male, "her" if the character is female, and "it" if the character is genderless.
heshe		"he" if the character is male, "she" if the character is female, and "it" if the character is genderless. Not settable.

4.4 Room Module

A module containing room-related utilities. This module is also home to the Room class, which is the Python representation of NakedMud's rooms. For importing purposes, this module's name is *room*.

4.4.1 Methods

Currently, there are no methods in the room module.

4.5 Room Class

This class is a Python wrapper around the ROOM_DATA C struct. Contains various methods and variables for interacting with the actual room that this Python class wraps around. The Room class is contained within the room module.

4.5.1 Methods

def __init__(self, vnum)
The initializer for the Room class. Takes in a vnum for the room we wish to create a Class around.

def attach(self, script_vnum)
Attach a script with the given vnum to the room.

def detach(self, script_vnum)
Detach a script with the given vnum from the room.

def send(self, mssg)
Send a message to every character in this room. A newline will be appended to the end of the message.

def dig(self, dir, dest)
Creates a new exit in the specified direction to the destination. Dest can be another room, or a room key.

def fill(self, dir)
Delete an exit in the specified direction

def exit(self, dir)
Returns the exit in the specified direction. None if none exists.

def isinstance(self, prototype)
Returns whether or not the room is inherited from the given prototype.

def edesc(self, words, desc)
Adds a new extra description to the room, tied to the specified words. Words must be a comma-separated list of things that can be looked at in the room, which will show the extra description.

def add_cmd(self, cmd, abbrev, func, min_pos, max_pos, group, mob_ok, interrupts)
The same as add_cmd as described in the documentation for the mud module, but the command only works in the room.

4.5.2 Variables

Rooms have various variables associated with them. Some are settable, and some are not. below is a list of those variables, and details of what each is for.

Variable	Settable	Description
name	√	The room's name
desc	√	The paragraph description of the room when looked at.
terrain	√	The type of terrain the room is.
uid		The room's unique identification number.
class		The room's most immediate prototype.
chars		A list of characters within the room.
objs		A list of objects within the room.
contents		Same as objs.
exnames		A list of the exit names this room has.

4.6 Object Module

A module containing object-related utilities (e.g. building a list of all objects in game, loading a new object instance to game, counting occurrences of an object instance in game, etc...). This module is also home to the Obj class, which is the Python representation of NakedMud's objects. For importing purposes, this module's name is *obj*.

4.6.1 Methods

def obj_list(self)
Returns a list of all the objects currently in the game.

def load_obj(self, prototype, where, equipto = None)
Load an instance of the object with the given prototype into the game. Where can either be a room or a room key, a container, or a character. If where is a character, equipto can be None to load the object to the character's inventory, an empty string to equip the object to the character in the first available slots, or a comma-separated list of body positions the object should equip to. Returns a pointer to the object just loaded into the game.

def count_objs(self, prototype, where = obj_list())
Returns a count of all the objects descended from the prototype in the given scope. Where can be a room or room key, a container, or a character. If where is not supplied, every object in the game is checked.

4.7 Obj Class

This class is a Python wrapper around the OBJ_DATA C struct. Contains various methods and variables for interacting with the actual object that this Python class wraps around. The Obj class is contained within the obj module.

4.7.1 Methods

def __init__(self, vnum)
The initializer for the Obj class. Takes in a uid for the obj we wish to create a Class around. Note that objects cannot be loaded into game by calling the Obj init function. To do this, you will want to use the load_obj method in the Obj module.

def attach(self, script_vnum)
Attach a script with the given vnum to the object.

def detach(self, script_vnum)
Detach a script with the given vnum from the object.

def isinstance(self, prototype)
Returns whether or not the object is inherited from the given prototype.

def istype(self, type)
Returns whether or not the object is of the given item type (i.e. container, furniture).

def settype(self, type)
Makes the object an instance of the specified item type.

def edesc(self, words, desc)
Attaches a new extra description to the object. Words is a comma-separated list of words that will show the extra description. Desc is the extra description.

4.7.2 Variables

Objects have various variables associated with them. Some are settable, and some are not. below is a list of those variables, and details of what each is for.

Variable	Settable	Description
name	√	The object's name
mname	√	The object's name when multiple instances are in the room.
desc	√	The paragraph description of the object when looked at.
rdesc	√	The description of the object when seen in a room.
mdesc	√	The object's room description when multiple instances are in the room.
keywords	√	A comma-separated list of named the object can be referenced by.
contents		A list of objects contained within this one.
objs		Same as contents
chars		A list of people sitting on this object.
weight	√	The weight of the object, excluding contents.
uid		The universal identification number for the object.
prototypes		A comma-separated list of the prototypes this object inherits from.
bits	√	A comma-separated list of bits currently set on this object.
carrier	√	The person who currently has this item in their inventory.
room	√	The room this object is currently in.
container	√	The container this object is currently in.

4.8 Event Module

Events are basically delayed calls to functions. Events could have a wide range of uses. For instance, events might be useful for preparing reboots or copyovers at prescheduled times, global quests, recurrent calls to functions (e.g. a combat handler), etc.... The event module is very simple, containing only 3 methods. It is imported as *event*.

4.8.1 Methods

def start_event(owner, delay, func, data = None, arg = None)
Prepares to call func in delay seconds. Func must be a python method that takes three arguments: the event's owner, the event's data, and the event's argument. The data and arg variables are optional, and can have values of None.

def start_update(owner, delay, method, data = None, arg = None)
Exactly the same as start_event, but the event will be added back into the event handler whenever it completes.

def interrupt_events_involving(thing)
interrupts any events that have thing as their owner.

4.9 Storage Module

The storage module has has two classes - StorageSet and StorageList. These are wrappers for C storage set and storage lists, respectively. StorageSets are designed to hold and convert between integers, strings, and double values (i.e. the types that make up everything in the game). Storagelists are lists of StorageSets.

4.9.1 Storage Set Methods

def __init__(fname = None)
Create a new storage set. If fname is provided, parse the storage set from the filename provided.

def readString(key)
Read the string value associated with the given key in the storage set. Keys must

always be strings.

def readInt(key)
Read the integer value associated with the given key in the storage set. Keys must always be strings.

def readDouble(key)
Read the double value associated with the given key in the storage set. Keys must always be strings.

def readBool(key)
Read the boolean value associated with the given key in the storage set. Keys must always be strings.

def readList(key)
Read the StorageList value associated with the given key in the storage set. Keys must always be strings.

def readSet(key)
Read the StorageSet value associated with the given key in the storage set. Keys must always be strings.

def storeString(key, val)
Store a key:string-val pair in the storage set. Keys must always be strings.

def storeInt(key, val)
Store a key:integer-val pair in the storage set. Keys must always be strings.

def storeDouble(key, val)
Store a key:double-val pair in the storage set. Keys must always be strings.

def storeBool(key, val)
Store a key:boolean-val pair in the storage set. Keys must always be strings.

def storeList(key, val)
Store a key:list-val pair in the storage set. Keys must always be strings.

def storeSet(key, val)
Store a key:set-val pair in the storage set. Keys must always be strings.

def write(fname)
Write the storage set to a file with the given name.

def close()
Close and free the storage set from memory, along with all storage sets and storage

lists stored within the set being closed.

def contains(key)
Returns true if the storage set contains a value with the given key.

4.9.2 Storage List Methods

def __init__()
Create a new storage list.

def sets()
Returns a list of all the storage sets contained within this storage list.

def add(set)
Add a new storage set to the storage list.

4.10 Auxiliary Data Module

The Auxiliary Module has a single method called install, which handles all of the installation of auxiliary data onto the basic data types.

def install(name, class, installs_on)
The name parameter is the name the auxiliary data will be known as for whatever it is installed on. The class parameter is the class that embodies this auxiliary data. It should contain an __init__ method that takes a storage set as an optional argument, a copy method, a copyTo method, a store method, and a read method. The installs_on parameter is a comma-separated list of things that the auxiliary data will install onto. Possible values in this list include "room", "character", and "object".