1) Introduction and overview

Tux:MP is much more than just a game. It is a game system, which means that it is a general framework that is extendable by end users. This manual covers these extensions. The reason behind this is to allow maps to do cool things. A simple race course would be ultimately boring. How about multiple courses on a single map? Want to have new players qualify on a bunny slope before they can race with everyone else? How about having teams of racers? How about relay races? It’s all possible with a little scripting.

Everything unique to a map is stored in a map file. The map file is what is downloaded from server to client, or loaded from disk during development. The map file is a file that contains other files such as the race script, images, models, sounds, etc. Map files are created and extracted with the make\texttt{map} script. The format of the map file is also documented in the $\texttt{make\_map}$ script itself.

When Tux:MP runs a map, it expects certain files to always exist within the map. The main two are $\texttt{map.py}$, the script that runs the map, and $\texttt{surface}$, which is a compressed file that describes the surface of your race course.

The $\texttt{surface}$ file is created by the surface editor, described in detail in a chapter of its own. The scripting language used by Tux:MP is a dialect of Guido van Rossum's wonderful Python scripting language, available at \url{http://www.python.org}. Tux:MP implements a subset of the python language, which we call "py". An undocumented difference between behaviour in py and Python is considered a bug.

Py generally avoids duplicate ways of doing things found in python. There are almany features of python that are not present in py for security reasons—py severely restricts access to operating system services like files or network. On the other hand, if your favorite text editor knows how to highlight and indent python, it will work fine with py. A reference for the py language can be found in a later chapter.

When a Tux:MP starts a map, it runs the $\texttt{map.py}$ script. The script will do things like place trees and gates on the map, define variables, classes and even create instances of classes. Graphics and physics are suspended until the initial script ends. When the script ends, threads must be in place to handle regular game events.

The threading model of py is different from Python. In py, the built-in \texttt{thread()} function calls a function with arguments. The thread is run at some later time. The order in which individual threads run cannot be guaranteed.

Once a thread is running, it will run until something happens that causes it to stop— it goes to sleep, an uncaught exception, or it tries to read (or perhaps send) a message. Unlike python, a thread is never interrupted by another thread. This makes things much easier.

Threads are easy things to create and have a wide range of uses. At the most trivial level, a thread can be responsible for displaying a message on the screen, sleeping for some interval and then deleting the message and itself. Less trivial threads send and receive messages from the network, handle players coming and going, and even controlling events during a race.

Another important concept is message channels, which allow threads to communicate with one another and the rest of the program. For example, when you hit a key on the keyboard, a message describing the key is generated in the depths of Tux:MP. The message is read by a script (not the map script), which decides what to do with it. This can amount to anything between making the penguin go faster or quitting the
game. Network messages are similarly sent between client and the server.

Messages are important because Tux:MP is built around sending and processing messages. Messages sent by scripts are mingled into a single stream with messages that are not visible to scripts.

For example, suppose a player joins a server. A (nonscript) message is sent to all players about the new player. New players start off invisible, so there is no physics that has to be done for them. At a later point in time, the new player selects the 'Race' menu option. This leads to a message being sent to his map script, which in turn sends a message to the server requesting to join the race.

When the server script gets the race request, it sends a reply back to the player accepting or rejecting the request. The network message causes something appropriate to be displayed on the player’s screen. If the request to race is granted, the new player goes into a list.

When the time comes for the race to start, the server sets the visibility and position of the players locally. This result in nonscript messages being sent to all players that causes the same thing to happen on all clients. A countdown clock can be started by another script message from the server.

When the race is started, the players in the race are "switched on" at the server, which lead to automatic messages to the clients that do the same thing. Physics processing starts and the penguins start sliding downhill. Control inputs are passed to the server in nonscript messages, and the server keeps its own idea of where the players are. The control inputs are also rebroadcast to other players. Since the server and clients are all processing the same messages in the same order, everything hopefully stays consistent.

It is possible to write a script that does not maintain consistent results—this has to be avoided. A client that gets out of synchronization with the server will disconnect itself.

The best way to get a feel for how a race is scripted is not to read the rest of this manual straight through, but rather to look at existing examples and then modifying small parts of them. When you see something you don’t understand, look it up in the manual and failing that ask someone for help.

2) Making a map, short version

Besides running a downloaded map or a single mapfile, Tux:MP can run maps from a development directory. To start developing a new map, create a new directory. The two files that are required to be in this directory are map.py and surface. Use the surface editor (described in another chapter) to create the surface file, or copy one from another map you like. If this is your first map, don’t try to write a new map.py, copy a simple one and modify it slowly. One place to get these is to save a downloaded map and then use the makemap utility to extract it into a directory where you can see and modify individual files.

To try your new map, run Tux:MP with a command line argument that is the path to the directory. This will run your map in single player mode. Run some races by yourself to check out how the terrain feels and tweak your map from there. Add any new textures or sounds that you need into the test directory as you need them.

When you are ready to test in multiplayer mode, you’ll need to create a single map file with the makemap utility, since a server loads the whole mapfile into memory in order to provide it to clients. To start a multiplayer map, run Tux:MP with the --server filename arguments to start the map. At this point, you can connect to the server by running a client in another window, connecting to localhost.

To make your map available publicly, start it on your dedicated server, connect to it and play a game. Then use the Leave option followed by the Connect option to connect to another public map. Your client will tell the public server about the new map, and it will spread the word to other servers and clients.
3) Surface editor

The surface editor is used to produce the map on which the race takes place. The file within a map is always named `surface`. It contains the height of the map for each point of the map, along with the surface type for each point. The surface editor is integrated with the game itself. To edit a surface, run the game program with the `--editor` option. When run in this manner, you’ll see an initial (flat) surface in view mode.

In all of the modes, a cyan rectangle is drawn at height \( z = 0 \) so that you can see the inclination of the surface. The origin of the map at \( x = 0, z = 0 \) has large vertical cyan line, and the map corner at \( x > 0, z = 0 \) has a smaller vertical line to give the viewer the orientation of the map.

The map editor is always in one of three modes– view, edit or drive mode. View mode allows the view of the map to be adjusted. The surface can be drawn as a wire frame, flat colored polygons or textured polygons. Edit mode changes the heights or surface type of the selected part of a map. Drive mode lets you drive around the map with the viewer position at the surface of the map, allowing you to see what the map will look like to someone on it.

Editing always starts out in view mode. In view mode, the mouse and mouse buttons do the following things:

- Holding left mouse down rotates the viewer around the map.
- Holding right mouse down moves the viewer around the map.
- Middle mouse selects a rectangular area on the map for editing.

View mode has the following key commands:

- `w` Toggles between drawing wireframes and filled polygons.
- `p` Toggles into driving mode (see below).
- `e` Toggles into edit mode (see below).
- `t` Toggles textures, which must be loaded to work.
- `-` Decreases the detail level.
- `+` Increases the detail level.
- `=` Increases the detail level (plus key without the shift)
- `v` auto-view– Switch the view to directly above the map, looking straight down.
- `PAGEUP` Zooms view out from the map.
- `PAGDN` Zooms view closer to the map.

Driving mode is like view mode except the view is constrained to follow the surface of the map. A pair of concentric square reticules are drawn in the middle of the screen. If the mouse pointer is in the middle of the inner reticule, the view remains constant. Moving the mouse above the inner reticule causes the view to move forward along the map. Moving the mouse below the inner reticule causes the view to move backwards. Moving the mouse left and right cause the view to pan left and right. Moving the mouse outside of the larger reticule causes the view to change at a faster rate than in the inner one.

The same keys are available in drive mode as are available in view mode, with the addition of the up and down arrows, which move the viewpoint up and down with respect to the surface of the map.

Eventually, you’ll want to actually change part of the surface. In view mode, select an area using the center mouse button and hit `e`. If an area has not been selected with the middle mouse button, the whole map will be selected. The portion of the map not selected will vanish and only the selected part will be drawn. If you select a large area, you will lose fine control over modifications. To make fine modifications, select a smaller area.

Unlike other modes, edit mode displays several status indicators. The top left of the screen shows the current position of the mouse in \( x \) and \( y \) map coordinates. This can be useful for finding the coordinates to place a tree or gate when writing a map script. The top right shows current brush mode, speed and noise settings.

The “brush” is the circular or square region around the mouse pointer. It is where actual changes to the map take place. If the brush is green, then the mouse buttons work the same way as in view mode, except
for the middle mouse button which cannot re-select areas. The space key toggles the brush to blue, which actually changes the surface, depending on the type of brush selected.

With a blue (active) brush, holding the left or right mouse buttons down causes something to happen to the area select by the brush. For brushes that modify surface height, the left mouse button moves the surface up and the right mouse button moves the surface down. For brushes that set the kind of surface, the left mouse button applies the change to all points contained in the brush. You can move the mouse while applying changes. This changes the position where changes are applied.

The keys available in edit mode are:

- `w` Toggle wireframe on and off.
- `b` Toggle between square and circular brush shapes.
- `e` Toggle from edit mode back to view mode.
- `m` Step the brush mode forward.
- `M` Step the brush mode backwards.
- `space` Toggle between a viewing (green) brush to editing (blue) brush.
- `s` Increase the speed parameter.
- `S` Decrease the speed parameter.
- `n` Increase the noise parameter.
- `N` Decrease the noise parameter.
- `v` Auto-view, toggles view to above map looking downward.
- `up` Increase the size of the brush.
- `down` Decrease the size of the brush.
- `pageup` Zooms view out from the map.
- `pagedn` Zooms view closer to the map.

The speed parameter controls how fast height changes are made to the map. The noise parameter controls how much random noise is added to each height modification. A little noise makes terrain look much more realistic.

The brush mode controls the kind of modification made to the area under a brush. The brush modes are:

- **HILL** In hill mode, a round hill is added to the current height map. The amount of height is the most at the center of the brush and tapers off exponentially, leaving the characteristic roundness.
- **PYRAMID** Pyramid mode is similar to hill mode, except that the height being added tapers off linearly from the center. This results in a pyramid shape for square brushes and cones for round brushes.
- **PLATEAU** A plateau brush moves the whole surface up or down uniformly. This mode is great for changing the heights of large areas.
- **SMOOTH** Smoothing mode causes points to be adjusted to the average of their neighbors. This removes previously added noise.
- **LEVEL** Level mode causes the height of the points within the brush to approach the height at the center of the brush. This differs from smoothing in that a smooth area can have a steady slope.
- **NOISE** Noise mode adds random noise to the brush area.
- **SNOW** Mark the area under the brush as ‘snow’.
- **ICE** Mark the area under the brush as ‘ice’.
- **ROCK** Mark the area under the brush as ‘rock’.
- **DIRT** Mark the area under the brush as ‘dirt’.

The last set of editor commands involve using the command console. Hitting the backtick character ‘ will cause a command line menu to drop down on the top left of the screen. While down, the console will read and execute commands until the backtick character is pressed again. The editor mode has commands that are only active there. These commands involve operations that require argument like filenames or other parameters. The commands are:
size
This command takes two integer arguments, the new \textit{x} size and \textit{y} size of the grid. Any existing height information will be truncated. The units of distance are roughly meters (or yards). A game penguin is about one unit tall. The surface files are compressed, but large rough maps take up more space than small smooth ones.

gradient
This command adds a constant gradient to the map. The two floating point arguments are the gradient in the \textit{x} and \textit{y} directions. The new heights are added onto the existing map and the whole map is shifted so that the lowest point on the map is always at \( z = 0 \).

load
This command takes a filename argument. It loads the surface into memory for editing, discarding the old one.

save
This command takes a filename argument. It saves the existing map to the filename, ready to be loaded into a game.

skybox
This command takes a filename argument. The skybox command loads a set of six PNG images which are used to form the sky. The filename is a prefix. The string \_\textit{ft.png} is appended to form the name of the front image, \_\textit{rt.png} for the right image, \_\textit{bk.png} for the back image, \_\textit{lt.png} for the left image, \_\textit{up.png} for the up image and \_\textit{dn.png} for the down image. This is a useful mode for testing skyboxes. If no skybox is present, the background will be a flat black.

3) Py reference

Early versions of Tux:MP had a map file that was parsed to load surfaces, objects and specify various aspects of game play, but it felt more and more wrong as development progressed. Eventually, we decided to replace it all with a real scripting language, to allow map designers to specify menus and even game play itself. This extended Tux:MP from a \textit{game} to a \textit{game system}, something easily extensible by end users.

Since our favorite scripting language is python, we looked at it but discarded it for a couple of reasons. The first involved security. We didn’t want malicious map programs doing nasty things to user’s systems. Earlier versions of python included various forms of a restricted execution environment, but this was removed by the designers of Python because the language is fundamentally impossible to secure. The second reason was that including the stock Python interpreter involved a lot of overhead, particular the modules.

The end result is that we’ve implemented a subset of Python that we call \textit{`py`}. Implementing a subset of Python means that we can eliminate the insecure parts of python—access to files and operating system services. At the same time, we’ve implemented extensions that facilitate game programming. Py is loosely based on the Python 2.x series, not 3.

If you run Tux:MP with an argument of \textit{--py filename}, your py program will be parsed and executed. This is useful for testing small programs or doing tasks like modifying databases offline. Programs that manipulate game objects won’t result in anything being displayed.

Conventions:

\textbf{typewriter font} Used for code and names that you’ll use in a real program.

\textit{italic font} Used to indicate a word that should be replaced with some sort of argument.

[ ] Square brackets indicate that the enclosed item is optional.

* An asterisk indicates that the preceding item can be repeated zero or more times.

| Items separated by a vertical bar indicate a choice between the items.

Major differences:

Py does not have lambda functions.

Py does not have decorators
Py does not have list comprehensions.
Py vectors look like, but are not tuples.
Py does not have modules.
Py does not have iterators.
Py does not have generators.
Object methods replace common library methods.
Py does not support `eval`, `exec`, or `execfile`

Names:

In py, names consist of letters or an underscore followed by letters, digits and underscores. Names can
be at most 100 characters long. Reserved words in py are:

```
and  break  class  continue
def  del    elif  else
except finally for  global
if    in     is     not
or    pass   print  raise
return try    while
```

There is no privacy name-mangling of class methods.

Statements:

Py programs are almost indistinguishable from python programs. A program consists of a series of
statements that are executed one after another, except where modified by flow control statements. Some
statements enclose blocks of other statements. A block of statements is distinguished by a new level of space
indentation. The end of a block is distinguished by the resumption of a previous level of indentation. A
deeper nesting of statements involves indentation greater that previously seen. Although it is not required,
nesting by increments of four spaces is a usual nesting convention.

Although most statements occupy a single line, ending a line with a `\` indicates that the current statement
is continued on the next line. If a literal list, vector or dictionary is being constructed, newlines are ignored
inside the literal. Multiple simple statements separated by a semicolon may be placed on a single line.

Comments begin with a `#` character. Anything following is ignored to the end of the line.

The `print` statement:

```
print [ expression ] [, expression]* [,]
```

The print statement causes the expressions to be printed. The output usually shows up on standard
output, or sometimes the game console. Each expression will be separated by a single space. If the `print`
statement ends without a comma, a newline is printed at the end. Otherwise, a single space is generated
and execution continues. The output will likely be buffered until a newline is printed by a future `print`
statement. Because there are no file objects in py, the extended print statement is not supported.

The `pass` statement:

```
pass
```

The `pass` statement does nothing. It is useful where a statement is required but no action is desired.
For example:

```
while sub():  
    pass # Loop forever
```

The `if` statement:

```
if expression:  
    suite  
[ elif expression:  
    suite ]*
```
[ else:
    suite ]

If the expression is “true”, the first suite is executed. Otherwise any elif expressions are evaluated in the order that they appear. For the first “true” expression, the corresponding suite is executed. If none of the expressions are “true”, then the optional else suite is executed if present. “False” values are considered to be the False object, the None object, numbers with a zero integer part, zero length strings, zero length lists and dictionaries with no keys.

The while statement:
    while expression:
        suite
[ else:
    suite ]

The first suite is executed repeatedly, while the expression is “true”. The test is done before execution. The optional else suite is executed if the while exits due to expression being “false” as opposed to a break statement. Like the if statement, “false” values are considered to be the False object, the None object, numbers with a zero integer part, zero length strings, zero length lists and dictionaries with no keys.

The for statement:
    for variable in sequence:
        suite
[ else:
    suite ]

The for loop assigns successive values of the sequence to the variable and executes suite. The sequence can be a list or a string. The else suite is executed unless the loop exits via a break statement.

The break statement:
    break

The break statement causes current innermost loop to cease, with execution continuing after the enclosing loop.

The continue statement:
    continue

The continue statement causes execution of the current innermost loop to cease and resume on the next iteration.

The del statement:
    del variable | map[item] | list[a:b] | object.attribute [, del-item ]*

The del statement deletes things. In the first form, del deletes the named variable. The second form deletes the item from a mapping, which can be a key/value from a dictionary or an indexed item from a list. The third form deletes slices from lists. The last form deletes attributes from class instances. Multiple items may be specified in a single statement, separated by commas.

The def statement:
    def name ( [ arg-name ] [, arg-name ]* ):
        suite

The def statement associates a function body with a name. If inside a class definition, the function becomes a method of the class. Otherwise, the function is associated with a global name. Once defined, a function can be called by name. Python’s default values for arguments are not supported. Static scopes (def’s within def’s) are not supported.
The `return` statement:

```
return [ expression [, expression]* ]
```

The `return` statement causes execution to cease within the innermost enclosing function. If a single expression is present, it is returned as the value of the function to the caller. If multiple expressions are present, a list of expressions is returned. If no expressions are present, the `None` object is returned.

The `class` statement

```
class name [ ( super-name [, super-name ]* ) ] :
  class-members *
  class-methods *
```

The `class` statement associates a class with a name. Assignments inside of a class become class members that are visible to all instances of a class. `def` statements inside a class create methods of a class, also visible to all instances. Names inside of the optional parenthesis are the names of super-classes from which this class should inherit members and methods. Classes are searched depth-first, left to right. These amount to Python’s “old style classes”.

Classes are quite useful in race programming, but inheritance should be used sparingly.

The `global` statement:

```
global name [, name ]*
```

The `global` statement causes a list of names to refer to global names instead of local names. This is only useful within function bodies.

The `try` statement:

```
The `try` statement comes in two forms. The first is:

```
try:
  suite
  [ except [ exception [, data-var ] ] :
    suite ]*
  [ else:
    suite ]
```

The `try` statement is used to catch exceptions. The first `suite` is executed. If any exceptions are generated, execution is interrupted and control is transferred to the innermost enclosing `try`, unwinding any nested subroutines to get to the `try`. The generated exception is matched against the exceptions listed in the order listed. Each `exception` may also be a list of comma-separated exceptions.

If an exception matches, extra data from the exception is assigned to the name `data-var` if present. Then the `suite` is executed. The `else suite` is executed if no exceptions are generated in the `try` block.

The second form of the `try` statement is:

```
try:
  suite
finally:
  suite
```

In this form, the first `suite` is executed. If an exception is generated, the second `suite` is executed, then the original exception is re-raised. If no exception is generated, the second `suite` is executed after the first `suite` finishes. If the first suite contains a control transfer statement like `break`, `continue` or `return`, the `finally suite` is executed “on the way out”.

The `raise` statement:

```
raise [ exception-name [, expression ] ]
```
The `raise` statement generates exceptions. If the `exception-name` is not present, the current exception is re-raised. Otherwise, an `exception-name` exception is generated. If the optional `expression` is present, it becomes the data associated with the exception.

Built-in Types: Numbers:

Numbers in py are fixed point numbers. These are different from 'floating point' numbers in that the number of digits before and after the whole and fractional parts are always the same. The whole part of a number ranges from zero to 2147483647 (31 bits). The fractional part has about eight decimal digits. Numbers have a sign and can be either positive or negative. Minus zero is a number that is distinct from plus zero, although minus zero equals plus zero if things come to that. Examples of legal numbers are:

```
1  -3  5.12  +123.456  .001
```

If a number is used in a context that requires an integer, then the fractional part is not used (the number is truncated towards zero). Numbers have no methods.

Strings:

A string is an immutable sequence of bytes. Literal strings begin with a single or double quote and are ended by the same quote that started the string. The backslash character `\` allows special characters to be encoded in a string literal. The characters that follow the backslash can be:

- `\` Backslash character
- `'` Single quote
- `"` Double quote
- `\a` Bell
- `\b` Backspace
- `\f` Form Feed
- `\n` Newline
- `\r` Carriage Return
- `\t` Tab
- `\v` Vertical Tab
- `\ooo` Character specified by octal digits `'ooo'`
- `\xhh` Character specified by hexadecimal digits `'hh'`
- `\newline` Continues the current string on the next line

Strings that are single quoted may contain unquoted double quotes. Similarly, double quoted string may contain unquoted single quotes. Strings that begin with triple quotes can be spread over multiple lines—they can contain raw newlines. Adjacent strings constants in a source file are concatenated. Examples of strings are:

```python
'Example string'  'she said "Hello".'
```

Strings can be indexed and sliced. If `s` is a string, then `s[0]` is a string that consists of the first character of the string. `s[0:1]` is a string that consists of the first two characters and so on. Negative indices are relative to the end of the string, so that `s[-1]` is the last character of the string and `s[-2:-1]` is a string that consists of the last two characters in the string and so on. Strings can be concatenated using the `+` operator. Strings that are multiplied by numbers are repeated. So `5*'a'` and `'a'*5` evaluate to `aaaaa`.

Strings can be formatted with the binary `%` operator. This operator takes a string on the left with a list on the right, and returns a string with formatting code in the string replaced with string representations with values of the list. If the string contains a single format item, then only the value being substituted can be used.

Characters in the format string are copied until a `%` character is seen, which causes the next format value to be converted into the place. Following the `%` character is an optional decimal width, which if present can be followed by an optional period and a decimal precision, followed by a character code that determines the kind of the data to be converted. The character codes are:
A literal percent sign, no format values are consumed
s Format the value as a string
x Format the value as a lowercase hexadecimal number
X Format the value as an uppercase hexadecimal number
o Format the value as an octal number
i Format the value as a decimal number
d Format the value as a decimal number
f Format the value as a floating point number
e Format the value as an exponential floating point number

If no width is specified, the smallest width possible is used when converting the value. If a negative width is specified, the value is left-justified within the field. If a leading zero is specified for a width, leading zeros are used for numeric conversions instead of spaces. If a width is not enough to convert a value, more characters are used without error.

If a precision is specified for a number, that many decimal places are used. If no precision is specified, the minimum number of digits after the decimal point is used.

String methods:
Methods with optional start and end parameters give the start and end position of a particular operation. If not specified, start is zero, and end is len(s). Whitespace characters are considered to be spaces, tabs, newlines, carriage returns, form feeds and vertical tabs.

s.capitalize()
Returns the string s with the first character capitalized if it is a letter.

s.center(width)
Returns a string of width width, with s centered, using space characters.

s.count(sub [, start [, end ] ] )
Returns the number of times that the nonoverlapping string sub appears in s.

s.endswith(sub [, start [, end ] ] )
Returns True if s ends with the string sub, False otherwise.

s.find(sub [, start [, end ] ] )
Returns the index at which the first occurrence of sub appears in s. Returns −1 if not found.

s.index(sub [, start [, end ] ] )
Returns the index at which the first occurrence of sub appears in s. Raises ValueError if not found.

s.isalnum()
Returns True if all of the characters in s are either alphabetic or numeric.

s.isalpha()
Returns True if all of the characters in s are alphabetic.

s.isdigit()
Returns True if all of the characters in s are numeric.

s.islower()
Returns True if all of the characters in s are lowercase alphabetic.

s.isspace()
Returns True if all of the characters in s are whitespace characters.

s.isupper()
Returns True if all of the characters in s are uppercase alphabetic.

`s.join(list)`
Returns a string consisting of all of the elements of list joined with s between elements.

`s.ljust(width)`
Returns a string of width width, with s left-justified.

`s.lower()`
Convert the uppercase letters of s to lowercase.

`s.lstrip()`
Strip whitespace characters from the left of s until the first non-whitespace character.

`s.replace(old, new [, maxrep ] )`
Replace up to maxrep nonoverlapping instances of old with new. If not specified, replaces all instances.

`s.rfind(sub [, start [, end ] ] )`
Returns the index at which the first occurrence of sub appears in s, searching from the right. Returns −1 if not found.

`s.rindex(sub [, start [, end ] ] )`
Returns the index at which the first occurrence of sub appears in s, searching from the right. Raises ValueError if not found.

`s.rjust(width)`
Returns a string of width width, with s right-justified.

`s.rstrip()`
Strip whitespace characters from the right of s until the first non-whitespace character.

`s.split([sep [, maxsplit ] ] )`
Returns a list of words in s separated by the characters in sep. If sep is not given, whitespaces are used as the separators.

`s.startswith(sub [, start [, end ] ] )`
Returns True if s begins with the string sub, False otherwise.

`s.strip(str)`
Returns a string with the characters in str removed from the beginning and end of s. If str is not given, whitespace characters are used.

`s.swapcase()`
Returns s with the lowercase letters converted to uppercase and uppercase letters converted to lowercase.

`s.title()`
Returns s with the first letter of each word uppercased, and the rest of the letters of each word lowercased.

`s.translate(table [, delchars ] )`
Returns the string s with characters in delchars deleted if present. The table string must be a string that is 256 characters long and determines how the remaining characters are translated.

`s.upper()`
Convert the lowercase letters of $s$ to uppercase.

$s$.zfill($width$)

Returns a string of width $width$, padding '0' characters on the left.

Lists:

Lists are sequences of any other py objects. Lists values are enclosed in square brackets and separated by commas. Examples:

- $[\ ]$ The empty list
- $[1, 2, 'a']$ List of three items.

Like strings, lists can also be sliced. If $L$ is a list, then $L[0]$ evaluates to the first element of $L$. Likewise, $L[0:2]$ is a list consisting of the first three elements of $L$. Negative slice values have the length of the list automatically added to them so that $L[-1]$ evaluates to the last element of the list. The in and not in operators can test for the presence of an element in a list.

Unlike strings, lists are mutable, and can be altered by putting a slice specification on the left side of an assignment statement. The statement $L[0] = a$ sets the first element of $L$ to $a$.

List Methods:

$L$.append($x$)

Appends $x$ to $L$ in place and returns None.

$L$.count()

Returns the number of elements in the list.

$L$.choice()

Returns a random element of $L$. Does not remove it. This is a py extension not found in Python.

$L$.extend($x$)

Returns None and appends all items in the sequence $x$ to list $L$.

$L$.index($value [, start [, end ] ]$)

Returns the position of the first occurrence of $value$ in the list. Begins the search at index start if supplied or zero if not. The last index searched is end if supplied or the last element of the list if not. Raises IndexError if not found.

$L$.pop($n$)

Returns element $n$ of the list, also removing it from the list. The first element is zero.

$L$.remove($x$)

Returns None and removes the first occurrence of $x$ in the list. Raises IndexError if not found.

$L$.reverse()

Returns None, and reverses the elements of $L$.

$L$.shuffle()

Randomly shuffles the list $L$ in place. Returns None. This is a py extension.

$L$.sort([ $cmp$ ])

Returns None and sorts list $L$ in place. If the function $cmp$ is supplied, it is called with two arguments to determine the order of pairs of elements in $L$. It must return zero if equal, less than zero if the first element is less or greater than zero if the second element is greater than the first. Uses regular py comparison if not supplied.
Dictionaries:

Dictionaries are objects that contain key-value pairs. Key types can only be numbers, strings, booleans or None. The values can be any py object. If d is a dictionary, d[1] = 'abc' sets the key 1 to the string 'abc'. The expression d[1] evaluates to the value of the key 1. Retrieving a key that is not in the dictionary causes a KeyError to be thrown. Membership of a key in a dictionary is tested with the in or not in operators. The has_key() method is not present in py.

Dictionary methods:

d.clear()

Returns None, clears all items from dictionary d.

d.get( key [, default ] )

Retrieves the value associated with key in dictionary d. If the key is not present, default is returned if it is given, otherwise returns None.

d.items()

Returns a list of key/value pairs in d, where each pair is a two-element list.

d.keys()

Return a list of all the keys in the dictionary in no particular order.

d.popitem()

Return an arbitrary two element key/value list from the dictionary, removing the key and value from the dictionary.

d.values()

Return a list of values in the dictionary in no particular order.

Vectors:

Vectors are a set of three numbers. Vector values are three numbers enclosed by parenthesis. They are like python tuples, but vectors are mutable and can only contain exactly three numbers. Vectors are used to specify points in space or a red-green-blue color triple. In the game, vectors are used for positioning players, objects the viewing camera, or defining the colors of floating text. Examples of vectors are:

(1, 2, 3) (-2, 2.4, 5.2) (255, 255, 128)

Vectors can be added with each other to produce another vector. The sum consists of another vector with components that are the sum of corresponding components. Vectors can also be subtracted from one another. A vector can also be added, subtracted, multiplied and divided by regular numbers. For example, 1 + (1,2,3) is (2,3,4).

Unlike python tuples, vectors are mutable and can be changed in place. If v is a vector, v[0] may be used to extract and set the first element.

Booleans:

The only two boolean values are True and False. These are produced by relational operators like >, <, etc. They have no methods.

Expressions:

Expressions consist of variables or constants that can be operated on by operators. If A and B are constants or variables, the operators are (from highest precedence to lowest):

( ... ), [ ... ], { ... }

Creation of vectors, lists and dictionary constants. Dictionary constants consist of a list of key-value elements inside of curly braces. The key expression comes first and is separated by a colon from the value expression. An example of a dictionary constant is: { 1:2, 3:4 }
Function call, indexing, slicing and method access of a class instance. The argument list in a function call must either be a list of comma-separated expressions, or a * followed by an expression that results in a list that will be passed to the function as the arguments. Keyword arguments and Python’s ** notation are not supported. The indexing and slice expressions, i and j must be integers.

Unary plus, unary negation, bitwise complement, exponentation

Multiplication, division, floor division, modulus (or string formatting). Floor division is like regular division except that the fractional part is truncated.

Addition (or concatenation), Subtraction

Bitwise left shift, Bitwise right shift.

Bitwise AND.

Bitwise exclusive-or.

Bitwise OR.

These are comparison operators. The traditional comparison operators work by comparing numbers as numbers and strings as strings. Lists and dictionaries are compared element by element until a difference is found. Different kinds of objects are compared in a consistent but undefined manner. The is and is not operators compare two object to see if they are the same object or not. The in and not in operators test for membership of the left object in a string, dictionary or list on the right.

Comparison operators may be strung together like the other operators. The expression A < B < C is as valid as 1 + 2 + 3. In the case of comparison operators, the B expression is only evaluated once, and evaluation of a multiple comparison stops if one of the comparisions is False, making the whole expression False without having to evaluate other terms.

Logical negation. The result is True if A is considered not to be True by the same test as described in the if statement.

Logical AND. B is only evaluated if A is True.
Logical OR. \( B \) is only evaluated if \( A \) is False.

Variable Scoping

Variable names that appear outside of a function have global scope. Variable names that are assigned within a function have the scope of that function. Otherwise the variable in the global scope is used. If a name is only referenced within a function, it is considered to be a global variable.

If a function needs to set a global variable, it must appear in a `global` statement. Functions cannot access variables in other function scopes unless they are passed as function parameters.

For example:

```python
a = 1  # a is global
b = 2  # b is global

def f(x):
    c = 1  # c is set, so it is local
    a = 1  # a is set, so it is local, masking the global a
    print b  # prints the global b
    return

def g(y):
    global a
    a = 1  # sets the global a
    print a  # prints the global a
    return
```

Variables that are members of a class instance follow the same rules, with the first variable having to be global in order to be set. For example, the built-in variable `world`, is an instance of the `world` class.

```python
# world is a global variable
world.ice_friction = 0.5

def f():
    print world.ice_friction
    return

def g():
    global world
    world.ice_friction = 0.25  # 'world' must be global to set its member
    return
```

Built-in names

The below list of names are built-in. They exist when a program is started.

```python
broadcast( message [, exclude ])

The broadcast() function sends the message string to all players. If exclude is present, it must be an integer player number. The message is not sent to an excluded player. Messages are received on channel 2 on the clients.

chr( x )

The chr() function returns a single character string given an integer ASCII character code. For example, chr(64) returns the string '@'.

decode( x )
```

15
The `decode()` function takes a string generated by the `encode()` function and returns the original object passed to `encode()`. Useful for converting an object or list of objects to a string suitable for passing as a message.

```python
encode( x )
```

The `encode()` function takes a py object and converts it into a string suitable for passing as a message. The format is inspired by Bram Cohen’s bittorrent b-encoding. Objects that can be encoded are lists, dictionaries, numbers, booleans, strings, vectors and `None`. Lists and dictionaries can contain other encodable objects.

```python
fixed( s )
```

The `fixed()` function is analogous to the (unsupported) Python `float` function. It converts a string to a fixed point number. A `ValueError` is thrown if something is wrong with the syntax of the number.

```python
hash( [ text ])
```

The `hash()` function returns a new instance of a hash object. An optional string argument is hashed if present. Hash objects have a member function named `update(s)` which allows additional string material to be hashed. They also have a `digest()` member function that returns a 64-digit hexadecimal value of the digest. The underlying hash function is Dan Bernstein’s SALSA20 hash function. This function is useful for generating cryptographic signatures.

```python
int( x )
```

This function returns an integer interpretation of `x`. If `x` is a string, the usual conversion is applied. If `x` is a number, then the integer part of `x` is returned. A `ValueError` is thrown if something goes wrong.

```python
isinstance( obj, classname )
```

Returns `True` if the `obj` object is an instance or subclass of `classname`, which must be the name of a class. Returns `False` otherwise.

```python
len( v )
```

The `len()` intrinsic function returns the lengths of things. If `v` is a string or list, it returns the length of those objects. If `v` is a dictionary, it returns the number of key/value pairs in the dictionary. If `v` is a vector, the Pythagorean length, \( \sqrt{v_x^2 + v_y^2 + v_z^2} \), is returned.

```python
max( list )
```

The `max()` function returns the largest item in a list, using regular py comparison.

```python
min( list )
```

The `min()` function returns the smallest item in a list, using regular py comparison.

```python
ord( s )
```

The `ord()` intrinsic function returns the ASCII character code of the character in a string of length one. For example, `ord('0')` returns 64.

```python
png( filename )
```

Given a string `filename`, loads a PNG image from the mapfile and returns a PNG object. The images is displayed as a floating object that is displayed above the regular scene.

```python
random( x )
```

The `random()` function returns a pseudo-random number. If `x` is less than one, a number uniformly distributed on the interval \( 0 \leq r < 1 \) is returned. If `x` > 1, a random integer from zero to \( x - 1 \) is returned.

```python
random_seed( x )
```
The `random_seed()` function seeds the pseudo-random number generator used by the `random()` function. The parameter $x$ is a number. The same seed will cause the same sequence of numbers to be generated by `random()`. If you use `random()` in your map, you should always call `random_seed()` because the state of a client’s generator can be different because of running a previous map.

```
range( start, stop )
```

Returns a list of integers from `start` to `stop − 1`.

```
recv( channel [, timeout ] )
```

Tries to receive a message on `channel`. Returns the string message or `None` on timeout. If the `timeout` parameter is present, the thread sleeps up to `timeout` game ticks before returning `None`. Without the `timeout` parameter, the thread does not sleep. `channel` numbers are 0 to 9, see below for a list of reserved channel numbers.

```
send( channel, message [, timeout ] )
```

The `send()` function sends the `message` string to `channel`. If no other thread is receiving on the channel, the thread sending the message is put to sleep for up to `timeout` game ticks. When another thread calls `recv()` on that channel, the sending thread will become unblocked and resume execution. If a message is not sent, an `IOError` is raised.

```
sendmsg( message )
```

On a client system, sends the `message` string across the network to the server. Clients cannot send to other clients. There is no corresponding `recvmsg()` on the server. Messages are `recv()`’d on channel 2.

```
sendmsg( player, message )
```

On a server system, sends the `message` string to `player`, where `player` is an integer player number. Like the client `sendmsg`, the message is `recv()`’d on channel 2.

```
sign( fontname, message )
```

The `sign()` intrinsic returns a new sign object. The `fontname` must be the filename of a TrueType font in the map file. The `message` string is the text of the message to draw. Signs are discussed at length in the section on floating objects below.

```
sleep( ticks )
```

The `sleep()` intrinsic puts the calling thread to sleep for `ticks` game ticks before resuming execution. If `ticks` is less than one, the thread will not sleep at all.

```
text( message )
```

The `text()` intrinsic returns a new text object which draws the `message` on the screen in a small internal bitmap font. Text objects are described below in the section on floating objects.

```
thread( sub, args )
```

The `thread()` intrinsic creates a new thread of execution by calling `sub` and applying `args`, which is a list of arguments, to it. The thread continues executing until it returns from `sub` or raises an unhandled exception. The `thread()` intrinsic returns a new thread object. Thread objects have a single member attribute named `running`. This member is `True` if the thread is running, `False` otherwise. A thread may be terminated by setting `running` to something that py considers false.

The order in which threads execute is not defined. You must not write a map program that depends on this— it will fail at random times.

```
world
```

The `world` variable is actually the single instance of a “world” class. It contains many things that control the game and has its own section, below.
Floating Objects

Floating objects, or “floaters” are graphical objects that appear to float in front of the game scene. They provide the menus and informational displays about the state of the game. Floaters are PNG images, renderings of letters in TrueType fonts and text messages in a simple bitmap font. When dealing with floating objects, the bottom left of the screen is at $x = 0, y = 0$, with $x > 0$ to the right and $y > 0$ upwards. The position values are in pixels. Colors are red-green-blue triples ranging from 0 to 255. An alpha value of 0 is completely transparent, an alpha of 255 is opaque.

All floating objects have a priority which determines which floaters are drawn first. Lower priority values are drawn first, and are obscured by floaters with higher priorities. Two floaters with the same priority will be drawn in an undefined order.

PNG objects are created with the `png()` built-in function. PNG objects have the following methods/members:

- $x$  
  This integer member holds the $x$ position of the PNG graphic.

- $y$  
  This integer member holds the $y$ position of the PNG graphic.

- `visible`  
  This boolean member controls if the PNG graphic is displayed or not.

- `width`  
  This read-only member gives the height of the PNG graphic in pixels.

- `height`  
  This read-only member gives the width of the PNG graphic in pixels.

- `alpha`  
  This integer member controls the alpha value of the PNG graphic.

- `priority`  
  This integer member controls the display priority value of the PNG graphic.

Sign objects

Sign objects are created by the `sign()` built-in. Sign objects have the following members:

- $x$  
  This integer member holds the $x$ position of the sign.

- $y$  
  This integer member holds the $y$ position of the sign.

- `priority`  
  This integer member holds the priority of the sign.

- `color`  
  This vector member holds the color of the sign.

- `alpha`  
  This integer member holds the alpha value of the sign.

- `scale`  
  This integer member controls roughly how many pixels high the sign is scaled to be.
width
This read-only integer member indicates roughly the width of the sign in pixels that the sign is. If the scale is changed, so does the width.

visible
This boolean member controls whether or not the sign is visible.

Text objects
Text object display messages in a simple bitmap font, and are created by the text() built-in. Text objects have the following members:
x
This integer member holds the x position of the text display.

y
This integer member holds the y position of the text display.
priority
This integer member holds the display priority of the text display.
color
This vector member holds the color of the text display.
bg
This vector member holds the background color of the text display. Default is black.
alpha
This integer member holds the alpha value of the text display. An alpha value of about 128 is about optimal between an invisible text message and an annoyingly visible one.

width
This read-only integer member holds the width of the text display in pixels.

visible
This boolean member controls whether or not the display is visible.

The Time Object
The time variable is a single instance of the time class. This instance is very similar to the Python time module. There are two representations of time. The first is the number of seconds since a particular epoch. Our epoch is midnight of January 1, 2000 GMT. This is different from the usual unix epoch of 1970 because we only have 31 bits in our integers. The second representation of time is a list of nine integers. The elements of the list represent broken down parts of a time:

<table>
<thead>
<tr>
<th>Year</th>
<th>The year in the range 2000+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month</td>
<td>Month number in the range 1-12</td>
</tr>
<tr>
<td>Day</td>
<td>Day of the month in the range 1-31</td>
</tr>
<tr>
<td>Hour</td>
<td>Hour of the day in the range 0-24</td>
</tr>
<tr>
<td>Minute</td>
<td>Minutes after the hour, 0-59</td>
</tr>
<tr>
<td>Second</td>
<td>Seconds after the minute, 0-59</td>
</tr>
<tr>
<td>Weekday</td>
<td>Day of week, where Sunday is 0, in 0-6</td>
</tr>
<tr>
<td>Day of Year</td>
<td>Day of the year, 0 - 365</td>
</tr>
<tr>
<td>DST</td>
<td>Daylight savings time, 1 in effect, 0 not in effect, -1 unknown</td>
</tr>
</tbody>
</table>

The day of the week field follows the C library convention, which differs from Python. When writing a game program, it is best to compute times in GMT time, send the GMT times to clients, which display them in local times, since players are likely in widely different time zones. The methods of time are:
time()
This function returns the number of seconds since the epoch. Our epoch is January 1st, 2000 at midnight, GMT.

gmtime([sec])
Given an integer time in sec, return a list of the broken down time for the GMT time zone. If sec is not given, uses the current time.

localtime([sec])
Given an integer time in sec, return a list of the broken down time for the local time zone. If sec is not given, uses the current time.

mktime(t)
Given a broken-down time list t representing a local time, return the number of seconds since the epoch.

strftime(format, t)
Return a string with the broken down time t formatted according to the format string. Regular character in format are copied. If a % character is found, the next character is examined to determine what to replace it with. The format descriptors are:

- %
- La
- %A
- %B
- %c
- %C
- %d
- %D
- %e
- %G
- %g
- %h
- %H
- %I
- %j
- %k
- %l
- %m
- %M
- %n
- %p
- %r
- %R
- %s
- %t
- %T
- %u
- %w
- %x
- %X
- %y
- %Y

20
The World Object

The `world` variable is a single instance of the world class. This class has many members and methods that control the rest of the game. The members of `world` are:

**screen.w**
This is a read-only integer that is the current width of the screen in pixels. This is not changable within a script. When a screen is resized, the value likely changes.

**screen.h**
Like `screen.w`, but this is the height of the screen in pixels.

**server**
Read-only boolean that is `True` if running on the server, or `False` if running on a client.

**ice_texture(filename)**
This is a function that takes the name of a PNG image within the map file. The image is used as the texture for ice surfaces.

**rock_texture(filename)**
This is a function that takes the name of a PNG image within the map file. The image is used as the texture for rock surfaces.

**snow_texture(filename)**
This is a function that takes the name of a PNG image within the map file. The image is used as the texture for snow surfaces.

**dirt_texture(filename)**
This is a function that takes the name of a PNG image within the map file. The image is used as the texture for dirt surfaces.

**sky(filename)**
This is a function that takes a string argument. The string is used as a prefix for loading the six PNG textures that form the skybox. The string `_ft.png` is appended for the front image, `_rt.png` for the right image, `_lt.png` for the left image, `_up.png` for the up image and `_dn.png` for the down image.

**tree(pos, height)**
This function returns a new tree object. As long as the object exists, a tree is displayed on the map. The `pos` argument is a vector that gives the position of the tree. The `height` is the height of the tree.

Tree objects have a single member function `texture()`. It takes the texture object that is to be used on the trunk of the tree. A common error is to call the `tree` function without assigning the object to anything. In this case, the object (and the tree in the game) will be deleted. The usual practice is to put the tree object into a global list of trees in the game.

**gate(pos, angle, width, height, radius, restitution)**
This function returns a new gate object. The `pos` argument is a vector that is the position to place one of the gate poles at. All other arguments are numbers. The `angle` is the angle in degrees that the second pole is at relative to the x-axis, the `width` is the distance between poles, the `height` is the height of the poles, and `radius` is the radius of the poles and `restitution` is the coefficient of restitution between penguin and the gate poles– zero is inelastic, one is perfectly elastic.

Gate objects have two attribute functions. The first is `texture()` which takes a texture object as the texture to apply to the gate poles. The second is `traversal()`, which takes no arguments and returns a list of player numbers that have traversed the gate during the most recent game tick. Must be called in every tick during a race.
texture(filename)

This function loads a PNG image from the mapfile and returns a texture object. The texture object can be applied to gates and tree objects.

camera(vector)

The camera() function takes a vector, and makes that the new location of the view camera.

camera_dir(vector)

The camera_dir() function takes a vector which defines the direction the camera is looking at.

camera_up(vector)

The camera_up() function takes a vector and uses it as the “up” direction when orienting the camera.

chase_player

The chase_player member indicates which player number is currently being chased. If the view is not in chase mode, it is None. Setting this member changes the chased player or free view mode.

chase_next()

This function causes the chase view to switch to the next chasable player in the sequence.

chase_prev()

This function causes the chase view to switch to the next chaseable player in the sequence.

chase_forward

This boolean indicates whether chase view is from behind or ahead of the current chased player.

course

This member is a list of strings that are the courses on the map. When the player selects the 'Race' item, one of these strings is passed to the map on channel one, indicating which race the player wants to participate in.

connected

This read-only boolean member indicates whether the client is connected to a server or not. Returns True if so, False if not.

me

For a client system, returns the player object associated with the local player.

lag

For a client system, this read-only integer attribute gives the number of ticks that a client system is lagged behind the rest of the game.

max_lag

For a server system, this read-write integer attribute is the maximum lag to tolerate before kicking a player.

player(n)

Given an integer player number n, return the player object associated with that player.

message_player

For a server system, this read-only integer attribute is the player number of the most recent network message. On a client, messages are always from the server.

model(filename, scale, offset [, flag])
This function loads a penguin model from a 3DS model file in map file. The filename argument gives the filename. The penguin should be centered at the origin, with the head in the +Z direction. The +Y is the dorsal side of the penguin, −Y ventral. When loading the points of the model, all points are multiplied by the scale and the offset vector is added (in that order). This allows fiddling with the penguin scaling and positioning outside of the 3DS editor. If the optional flag is present and true, the “debug penguin” is drawn—this is a dozen points on the surface of the penguin as well as an arrow that gives the direction of the head.

GL materials defined in the model file will be used to color the penguin surface, with small variations made for different players. Most 3DS extensions are ignored.

snow_fricction
This numeric attribute allows giving or setting the friction associated with snow surfaces. Must be in the range 0 < f < 1.

ice_fricction
This numeric attribute allows giving or setting the friction associated with ice surfaces. Must be in the range 0 < f < 1.

rock_fricction
This numeric attribute allows giving or setting the friction associated with rock surfaces. Must be in the range 0 < f < 1.

dirt_fricction
This numeric attribute allows giving or setting the friction associated with dirt surfaces. Must be in the range 0 < f < 1.

ground_restitution
This numeric attribute allows giving or setting the coefficient of restitution between the penguin and ground. Zero gives a completely inelastic collision, one is a completely elastic collision.

penguin_restitution
This numeric attribute allows giving or setting the coefficient of restitution for penguin-penguin collisions. Zero gives a completely inelastic collision, one is a completely elastic collision.

control_factor
This numeric attribute sets how the penguin responds to ground forces. Zero causes the penguin to remain completely stable. A value of 1.0 makes the penguin almost uncontrollable. This basically limits how fast penguins can go before they tumble out of control.

map_db
For the server, this attribute is a database instance that corresponds to the map database. The map database is a file named map.db that the server uses to store permanent data. See the section on database objects for accessing the database.

sound( filename )
This function loads a sound from filename and returns a sound object. The object has a single method, play( [ loop ] ). When play is called, the sound is played. The optional loop parameter indicates how many times to loop the sound. The default is zero, which plays the sound once. A loop of one plays the sound twice and so on. A loop of −1 loops forever.

Player Objects
Player objects represent players within the game. Even when players leave and join in a networked game, the objects remain the same. Player objects have the following attributes:

number
This read-only integer member is the player number of the player object. It cannot be changed.

**position**
This read-only vector attribute gives the player’s current position.

**velocity**
This read-only vector attribute gives the player’s current velocity.

**set_position(pos, angle)**
The function sets the position of a player. It can only be called in single-player mode or by the server in multiplayer mode. The `pos` parameter is a vector position, the `angle` is an angle in degrees measured from the +X axis. The height part of `pos` is ignored. The penguin is placed resting on the surface. In multiplayer mode, the new position is sent to all clients automatically.

**kick()**
This member function causes the player to leave the game immediately. Useful for dealing with griefers.

**address**
This read-only string attribute is the IP address of a connected player. Useful for keeping track of griefers!

**visible**
If `True`, the penguin is drawn and can be hit by other visible players. This boolean attribute is settable only in single player mode or on the server in multiplayer mode. Changing the value on the server automatically transmits the change to clients.

**physics**
If `True`, the penguin is subject to the laws of physics— it will slide downhill. If the penguin is visible and `physics` is `False`, the penguin is an immovable object when hit by other penguins. This boolean attribute is settable only in single player mode or on the server in multiplayer mode. Changing the value on the server automatically transmits the change to clients.

**control**
If `True`, the penguin is subject to control inputs from its user. Otherwise, control inputs are thrown away, and the penguin will gradually slide to a halt. This boolean attribute is settable only in single player mode or on the server in multiplayer mode. Changing the value on the server automatically transmits the change to clients.

Database objects
Database objects provide a way of persistent storage across invocations of a server. Database objects associate string keys with string values. Multiple columns can be simulated by using the `encode()` intrinsic on a list of items. Multiple tables can be simulated by prefixing keys with a common value.

For example, supposed you wanted to keep track of griefers by IP address. If you have an IP address, you could create a key using:

```
key = 'griefer:' + address
```

where the value of the address could be an expiration time. A single lookup determines if the IP has been banned, and a periodic sweep removes bans that have expired.

All database methods can throw an `IOError`. Database objects have the following attributes:

**get(key)**
This function returns a string value associated with the string `key`. Throws a `KeyError` if the key is not in the database.
put(key, value)
    Associates the string value with the string key. A previous association is overwritten.

delete(key)
    Deletes the key from the database. If the key does not exist, no error is thrown.

has_key(key)
    Returns True if key is present in the database, False if not.

next_key(key)
    Returns the next key in the database following the given key. Returns the null string if no keys remain.
    In order to search for all keys that start with a common prefix, start the search for the prefix itself.
    When you reach a next-key that does not start with the prefix, you’ve seen all of the keys with that
    prefix.

Scripting Tutorial

Now that the components of map scripting have been documented ad nauseum, it’s time to describe how
to put the pieces together. Let’s start out with a script for a single player map. We’ll start with initializing
various world parameters:

# Map program for the Bunny Slope

world.snow.texture('Snow.png')
world.rock.texture('Rock.png')
world.ice.texture('Glacier.png')
world.sky('sky')

world.courses = [ 'Bunny Slope' ]

world.snow.friction = 5.0
world.rock.friction = 10.0
world.ground.restitution = 0.1

finish_gate = world.gate((250, 500, 0), 0, 20.0, 10.0)

The first group of lines handle load textures for the various surface types along with the skybox. The
middle line makes the racable course names available to the control script. The last group of lines initializes
friction values for each of the surface types.

Let’s make a countdown timer, a subroutine for putting it in the right place, and call it for the first
time. We assume that we have a TrueType font named lcd.ttf in the map file/directory.

timer = sign('lcd.ttf', '')
timer.scale = 60

def position_timer():
    timer.x = world.screen.w - 300
    timer.y = world.screen.h - 100
    return

position_timer()

The next section of code is a function named race that handles a race from beginning to end. It runs
in its own thread.

def race():
    global world, timer
world.me.visible = True
world.me.physics = False
world.me.control = False

world.me.set.position((250, 2950, 0), 180)
world.chase.player = world.me.number
timer.color = (255, 0, 0)

The first section of the race function initializes the player’s penguin to be visible, without any user control input and without processing physics for it. It also sets the view to be chasing the player and sets the color of the timer to red.

\[
n = 5 \times 30 \quad \# \text{5 seconds} \times 30 \text{ ticks/second}
\]
while \(n > 0\):
    if (n % 30) == 0:
        t = n // 30
        if t == 2:
            timer.color = (255, 255, 0)
            pass
        timer.string = '0:%02d ' % t
        pass
    if n == 5:
        self.sign.color = (0, 255, 0)
        pass
    n = n - 5
    sleep(5)
    pass

world.me.physics = True
world.me.control = True

The next section handles the countdown timer. There are 30 game ticks per one second, and the \(n\) variable counts ticks until the start of the race. We count down by five ticks at a time because we want to change the color of the timer to green before the timer changes from a countdown to stopwatch. Two seconds before start, the timer is changed to yellow.

\[
n = 0
\]
while \(n < 2 \times 60 \times 30\): \# Two minutes
    if world.me.number in finish_gate.traversal():
        break
    t = 0.30 * tick \# Tenths of seconds
    timer.string = '%d:%02d.%d' % [ t // 600, (t // 10) % 60, t % 10 ]
    n = n + 1
    sleep(1)
    pass
else:
    timer.string = 'TIMEOUT'
The next section handles the race itself. First, the player is freed and control inputs are enabled. The \( n \) variable is used for the number of ticks in the race so far. The loop continues until we reach a timeout or the player traversed the \texttt{finish\_gate}. The timer is also updated every tick.

```python
world.me.control = False
race_thread = None
return
```

The last section cleans up. The control is removed from the penguin, which will slide to a halt. The variable holding the thread object is cleared to \texttt{None}, because the thread terminates when the \texttt{return} is executed. The last part of the program is a function which processes system messages. It looks like:

```python
def run_cmd(cmd, arg):
    global world

    if cmd == 'race':
        if race_thread is not None:
            race_thread.running = False
            pass

        self.race_thread = thread(race, [])

    elif cmd == 'leave':
        if self.race_thread is None:
            return

        race_thread.running = False
        race_thread = None
        world.me.physics = False
        world.me.visible = False
        pass

    elif cmd == 'resize':
        position_timer()
        pass

    return
```

This functions processes messages received from the internal menu system. The 'race' command kills any existing race and starts a new race thread. The 'leave' command just kills an existing race thread and the 'resize' command repositions the timer in response to a window resizing.

```python
def command(self):
    while True:
        cmd = recv(3)
        m = cmd.split(' ', 1)
        if len(m) == 1:
            run_cmd(m[0], None)

        else:
            run_cmd(m[0], m[1])

        pass
    pass # Never returns
```

The last function, \texttt{command()}, reads channel 3, and calls \texttt{run\_cmd} with what it gets. It runs forever.
The last bit of code starts the rest of the game in motion:

```python
race_thread = None
thread(command, [])
```

That’s all there is to it for the simplest map.