TinyStory

1.1.3

TinyStory is small indentation-based Pick Thine Individual Escapades engine I wrote because why not.

Syntax

The most important thing to note is that the nesting of text and options is achieved with indentation. For that, you must either use tabs or 4 spaces, but mustn't mix the two.

Options

```
* I am a clickable option
    And this is the text that will be displayed if you click it
```

Note: If there are no subsequent options or jumps, the previous options will be displayed again.

Setting Variables

```
= variable value
```

Conditions

Check if a variable has a certain value. Not putting a value will default to `undefined`. Conditions can be nested.

```
: variable true
    * I appear only if the variable is true
: variable
    * I appear if the variable hasn't been set or is undefined
: v1 true
    : v2 true
        I appear if both v1 and v2 are true.
```

Labels

Mark parts to jump or link to. Following blocks are nested through indentation as well.

```
[Label]
    * I'm part of the label's block!
```

Jumps

```
* Option text
    [Label]
        We're going to jump here in a second
* Here we go
    > Label
```

Note: Putting `> End` will stop the game.

Links

Links work pretty much the same away, except they don't erase the displayed text, so you can merge texts.

```
* Some option
    [Label]
        Things are happening here!
* And now for a link...
    As we can see:
    < Label
```

Output:

> As we can see: Things are happening here!

Functions

You can define and call functions by first adding functions to the TinyStory.functions object in the index.html:

```
<script type="text/javascript">
    window.onload = function()
    {
        TinyStory.functions.load = function() { TinyStory.loadGame(); }
        TinyStory.functions.save = function() { TinyStory.saveGame(); }
        TinyStory.load();
    }
</script>
```

And then executing them from within the story:

```
* Load
    ! load
    > Start
* Save
    ! save
```

Customization

Title

This will change the HTML title and add the title to the top of page as well:

```
# title A Cool Title
```

Font

You can easily add a Google Font by adding:

```
# font Google Font Name
```

Audio

You can play an audiofile by adding:

```
# audio path/to/file.ogg
```

The file should be provided in both OGG and MP3 format. You must link to the OGG file though. It will loop automatically.

Background

To set a background image, you can simply add:

```
# bg path/to/file.png [options]
```

You can also set a few basic options for background size, repeat, and position. These include:

```
[cover|contain] [no-repeat|repeat|repeat-x|repeat-y|space|round] [center|top|bottom|left|right]
```

These can be combined in any order. For example:

```
# bg path/to/file.png center cover no-repeat
```

CSS

For more involved styling, costumizing the style.css file is of course the best approach. But if you want to make a few small changes and keep them all contained in your story file, you can set CSS properties by adding:

```
# css [selector] [property] [value]
```

By default, the following selectors are available: * html * body * .text * .option * #wrapper * #title * #credit * #error

For example:

```
# css #title margin-top 50px
```

Filename

If you would like to change the name of the story file, tell TinyStory what it is in the index.html by setting

```
TinyStory.filename = "yourname.txt";
```

prior to load().

Functions

You can register functions to call from the story (see Syntax > Functions) by adding them to the TinyStory.functions object in the index.html:

```
TinyStory.functions.yourFunctionName = function() {
    // Your function
};
TinyStory.functions.anotherFunctionName = function() {
    // Another function
};
```

Fade Speed

Change the fade speed by setting

```
TinyStory.fadeSpeed = 500;
```

in the index.html (value in ms).

Restart Pause

Change the timeout duration during a restart by setting

```
TinyStory.restartPause = 500;
```

in the index.html (value in ms).

Autosave

You can enable autosave by setting

```
TinyStory.autosave = true;
```

in the index.html. It will save whenever a new branch is loaded.

Autoload

You can enable autoload by setting

```
TinyStory.autoload = true;
```

Autoload will load the last save upon starting the game.

Auto-Return

Auto-return is the automatic loading of the last node when no further actions or links are present within a node. You can disable this by setting

```
TinyStory.autoReturn = false;
```

Custom Save/Load/Restart

If you want to create custom save, load, and restart functions, you can use:

```
TinyStory.saveGame();
TinyStory.loadGame();
TinyStory.restart();
```

Example

Finally, here's a quick example to make all of that up there a little clearer:

```
# title Example

What would you like to do?

* I want to set a variable.
    Alrighty, let's do it!
    * Set it to true
        Done.
        = var true
    * Set it to 5
        Okay.
        = var 5
        // Note that if no further options or jumps exist, the previous options are displayed again
    * Now I'd like to do something else.
        > I want to do something else.
    * Okay, that's enough.
        As you wish.
        > End
* I want to do something else.
    : var undefined
        I see you didn't set the variable.
        > End
```

Check out the `sample.txt` for a more complete example.

Tips and Tricks

Modular Scripting

Whether you have nodes that you want multiple branches to pass through, or simply want to divide up your code to make it more manageable, your first instinct might be to break it down into several labeled blocks:

```
[block1]
    // Nested choices eventually linking to block2 and block3

[block2]
    // Nested choices eventually linking to block3

[block3]
    // Ending
```

However, TinyStory executes all same-level instructions it can reach, it doesn't "enter" the label to stop execution of the rest. They are merely jumping markers and otherwise work no different than three consecutive lines of text, meaning all three blocks will be executed at once. To prevent that, you can simply set a condition that will stop execution on the first pass:

```
[block1]
    // Nested choices eventually linking to block2 and block3

: started true
    [block2]
        // Nested choices eventually linking to block3

    [block3]
        // Ending
```

Now TinyStory checks for a variable named `started`, which we haven't set, skipping the nested blocks until `block1` links there. Since jumps and links go directly to the labels and don't check prior conditions, we won't have to ever set that variable either. We simply create a condition that is always false, allowing us to jump between labels as and when we need to.

Execution Order

Let's say we want to create a simple choice loop:

```
[choice]
    * Wait
        You wait and wait, but nothing happens.
        < choice
    * Leave
        > End
```

In this example, the player has the option to wait for as often as they like, before eventually choosing to leave. To spice this up though, we can add some conditions that track whether they have waited before:

```
[choice]
    * Wait
        : waited // with no value given, the condition checks whether waited is undefined
            You wait and wait, but nothing happens.
            = waited true
            < choice
        : waited true
            You've been waiting for a while, but still nothing.
            < choice
    * Leave
        > End
```

On first glance, this seems like a simple enough solution: check whether the variable `waited` has been set before, displaying the first message if it hasn't and the second if it has. However, just as with labels, TinyStory doesn't consider conditions blocks that it will enter and stop execution of the next. All same-level instructions under the block `* Wait` will be executed sequentially, because of which `waited` will be set to `true` before the second condition is tested, executing them both.

To prevent this, the order of execution is important to consider:

```
[choice]
    * Wait
        : waited true
            You've been waiting for a while, but still nothing.
            < choice
        : waited // with no value given, the condition checks whether waited is undefined
            You wait and wait, but nothing happens.
            = waited true
            < choice
    * Leave
        > End
```

Now the first conditional will be skipped before the variable is set in the second, allowing different messages to be displayed on first and consecutive visits.