Mysterious Messenger Wiki

Release v3.3.0

Shawna P

Sep 25, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Welcome to Mysterious Messenger, a messenger game created in Ren'Py. This guide was created to help users understand how the program works together, and how to take advantage of the various functions.

This guide is best used by navigating to the sections with the information you need as you need it rather than reading it from start-to-finish. Most sections have a line at the top that says "Example files to look at"; these will usually show the features described in the guide in action in the program itself. As much as possible, the code has been annotated to help you understand what's happening in the program. Additionally, you'll find plenty of code excerpts in this guide itself. They look like this:

```
label day_1_chatroom_1():
    play music mystic_chat
    enter chatroom u

    u "Congratulations! You've created your first chatroom."

    menu:
        "That's amazing!":
            u "Isn't it? I'm glad you think so."
        "This is a lot of work.":
            u "I'm sure it will get easier with practice!"
            u "There are plenty of wiki pages to help you out, too."

    u "I'm leaving now. Good luck!"
    exit chatroom u
    return
```

Most of these code excerpts are taken directly from the program itself and can be copied in to use for your own test purposes.

Some sections will also have a "quick-start" guide at the beginning; they look like this:

Note: Example files to look at:

- tutorial_5_coffee.rpy
- tutorial_6_meeting.rpy

A brief overview of the steps required (more detail below):

1. Create a new `.rpy` file (Optional, but recommended)
2. Create a new label for the chatroom.
3. (Optional) Set the background with `scene morning` where `morning` is replaced by whatever time of day/background you want to use.

4. Add background music with `play music mystic_chat` where `mystic_chat` is replaced by the desired music variable.
5. Write the chatroom dialogue.
    1. You may want to use either `Script Generator.xlsx` or the `msg` CDS.
6. End the label with `return`.

If you just need a quick reminder of what to do to get some code up and running, you can refer to this. If you need more detail, however, the rest of that section will explain further.

If you haven't already, you should also play through the Tutorial Day in the program at least once to get a feel for what the program is capable of. Then, if you're new to Ren'Py and/or to programming, you should take a look at the *Beginner's Guide*, which will take you through the steps to make the first chatroom of a new route playable.

CHAPTER

TWO

FULL CHANGELOG

2.1 3.3.0

It should be noted that this new release supports Ren'Py 8.0+, which uses Python 3. It is suggested you move to Ren'Py 8+ with new games, as Python 3 offers greater flexibility.

2.1.1 Major New Features

Feature	Description
Automatic chatroom backgrounds	The game will automatically set up chatroom backgrounds based on the time of day the chatroom occurs at. You can override this automatic background by providing your own e.g. `scene morning`.
Adaptable screen size	The game can adapt to screen sizes taller than the original 9:16 ratio e.g. 9:19

2.1.2 Minor New Features

Feature	Description
Manually award hourglasses	Convenience function added for more control over when the player is awarded an hourglass.
Chat vs Gallery CGs	Show a different image in chatrooms than in the gallery when sending CGs.
Phone call expiry dict	A convenience dictionary to easily customize how long phone calls should be availlable for.
Email testing hub	An email testing hub from the Developer settings in-game allows you to quickly invite guests, send replies, and test the party.
VN Screen shake	You can use `show shake` during Story Mode (VN mode) to shake the screen. Takes into account the player's accessibility preferences.
Splash Screen	The splash screen has been relocated and updated to make it easy to add your own image and "Tap to Start" before entering the main menu.
Award Hearts	You can provide a number to `break heart` or `award heart` to award/break multiple hearts at once.

2.1.3 Fixes

Fix	Description
Doc improvements	Added doc pages on *deleting messages* and *adding greetings*.
Profile picture callback fix	Profile picture callbacks still go off even if you use a text message or email popup notification to leave the profile screen.
History fix	Fixed an issue when rewatching chatrooms in the history after buying back that same chatroom during the story.
MC Profile Picture bug	Fixed a bug with the `add_mc_pfp` function to properly unlock profile pictures as seen.
Animated Backgrounds	Animated backgrounds move more smoothly and don't have any stuttering lines when looping.
Guest Comment Interpolation	Guest comments in the guestbook properly interpolate variables like `[name]` or pronouns like `[they]`. There is also a dissolve transition in/out of viewing the comment.
Guest Hourglasses	Fixed an issue where players weren't awarded an hourglass upon first viewing a guest's information in the guestbook.
Branch VNs	Fixed an issue with branch VNs that would cause a script error dialogue for trigger times.
Background shake	Improved how background shake is set up. Backgrounds without an animated version still shake when animated backgrounds are turned on.
Timeline display	The timeline correctly displays the "5th DAY" with the suffix when opening a particular day's timeline list.
Chat Creator	Fixed an issue where the chat creator didn't correctly export other characters' special bubbles.
Hack tear screen update	The hacking "tear" effect is updated for Ren'Py 8 and functions better with large numbers of sections.
Text message preview	The text message preview can handle any number of interpolated variables, as needed.
CG Gallery fixes	Fix CG gallery thumbnails with automatic cropping.

2.1.4 QoL Improvements

Feature	Description
Remember save page	The game remembers which save page you last used.
Main Menu Screen Actions	Common actions for buttons on the main menu and home screen have been turned into functions for easier UI modification.
Check for updates in thread	The updater launches a separate thread to check for updates so the UI is not paused/frozen while waiting for the results of the check.
`gamestate`	Whether the program is in a chatroom, phone call, VN/Story Mode, or text message is tracked via a consistent *gamestate* variable.
Available call indicator	A new dev option will add a "NEW" sign under a contact's icon when there is a call available for them and add an `Online` indicator on their profile screen.
get_terms	Convenience function for using gendered terms to refer to the player.
Developer Release Options	Added a section to `variables_editable.rpy` which can be customized for sharing a program distribution with others.
Email newlines	You can turn off the parser that turns email single newlines into a space (`email_newline_to_space`).
Gallery update	The gallery has been redone to cause fewer headaches overall and requires one definition instead of two.
`big` text tag	There is now a `{big}` text tag which can be used similarly to how `big` is used for the msg CDS.
Popup updates	Popups have been updated to take advantage of screen tags. An infinite number of popups of any type can be on screen at any time (so, no more restrictions on how many text message popups, heart icons, or stackable notifications etc you can see at once).

2.1.5 Incompatible

- While not strictly incompatible, the gallery system has been updated to use constants rather than both a default and a persistent album variable. The game will attempt to transfer over any unlocked photos from the old gallery version into the new one. It is recommended that you update your gallery definitions to use `define` and the new `GalleryImage` class rather than `Album`. You can find more information *in the docs on the gallery.*

2.2 3.2.0

2.2.1 Major New Features

Feature	Description
Chatroom Creator	A visual creator to write chatrooms and export them to code or watch them as played in-game.
Update to 7.4+	Mysterious Messenger takes advantage of 7.4+ update such as matrixcolor. You will now require Ren'Py 7.4+ to run Mysterious Messenger.

2.2.2 Minor New Features

Feature	Description
Phone hang up callback	A special function is called when the player hangs up in the middle of a call, which you can use to alter the game state.
Manually send text messages	Manually send text messages during chatrooms, phone calls, Story Mode, or whenever you like.
Vanderwood expressions	Vanderwood has new expressions thanks to Rom
Choose profile picture	(Windows only) Choose a profile picture from your computer to use in-game.

2.2.3 Fixes

Fix	Description
Starter story fix	Intro chats can properly have incoming calls and don't boot you out to the main menu.
Updater fix	Updater properly remembers ignored releases and stops checking for updates when it reaches your current version.
Skip Intro/Save & Exit	The chatlog no longer clears when finishing the starter story before exiting to the home screen.
SFX Audio Captions	Sound effects are correctly looked up in the sfx dictionary, not the music dictionary.
`hide_albums` fix	`hide_albums` function works correctly again.
Save file names	Save file names are more reliably saved using json files instead of string separators.
Fix Persistent button	Fix persistent now works with user-defined albums
Animated background screen shake	Animated backgrounds now get black bars offscreen to the left and right to improve the appearance of screenshake while animated backgrounds are active.
Tear screen fix	The tear screen feature has been updated for Ren'Py 8.0

2.2.4 QoL Improvements

Feature	Description
msg CDS underline	The message CDS takes optional `underline` or `under` arguments to underline text.
SFX in replay	Sound effects are included in chatroom replays.
Profile picture scaling	Profile pictures now crop and size themselves to fit the desired dimensions rather than squash/stretching into a square

2.3 3.1.0

2.3.1 Major New Features

Feature	Description
Missed calls callback	Phone calls can play out differently depending on whether the player is calling back a missed incoming call or not. Also supported in the History.
Phone call-only characters	Can create a PhoneCharacter specifically for phone calls who won't show up in the player's contacts.
Input prompts	Can get typed input from players anywhere in the game.
Profile Updates	Players can choose their gender separately from their pronouns, and enter a chatroom username separately from their regular name.

2.3.2 Minor New Features

Feature	Description
reset_participants	New function allows you to easily reset the participants list in the middle of a chatroom.
Link wait pause button	Dev toggle allows you to toggle the footer while waiting for the player to click a link to be the pause button.
Reversable hack effect	The scrolling hacked effects can be reversed to scroll in the other direction.
LinkJump	New action for links to jump to somewhere else in a chatroom (including menus).
IfChatStopped	New action for links that changes depending on whether the chat is stopped or not.
New backgrounds	Three new backgrounds have been added: `snowy_day`, `rainy_day`, and `morning_snow`

2.3.3 Fixes

Fix	Description
Pause button fix	Pausing chatrooms and real-time text messages has been overhauled to cause fewer issues with skipped messages.
Vanderwooo	Fixed Vanderwood's name getting cut off in the phone contacts.
Erroneous msg CDS errors	Skip Intro and Jump to End now work properly with the msg CDS during chatrooms.
Skip Intro fix	Skipping the introduction no longer causes real-time text message to prompt you about leaving even though the conversation is over.
Participated fix	Fixes a bug so the MC's profile picture shows up in the list of participants after playing a chatroom.
Unlocked Profile Pics Fix	Default profile pictures for characters now only unlock after you've seen them as part of the game.

2.3.4 QoL Improvements

Feature	Description
Reset Albums	You can tell the program to forget all seen images and unlocked album entries without losing other persistent variable progress.

2.3.5 Other

Feature	Description
Animated bg relocation	Animated background images have been relocated into their own folder inside of the **Phone UI** image folder for better organization.

2.4 3.0.1

You can find the docs for v3.0.1 here: https://mysterious-messenger.readthedocs.io/en/v3.0.1/

2.4.1 Major New Features

Feature	Description
Chatroom Links	Characters can now post links in chatrooms. Clicking a link can cause a variety of actions to occur, such as jumping to a Story Mode section or showing a CG.
Vanderwood	Vanderwood has been added as a fully-fledged character with his own profile, chat bubbles, heart icon, and more.
Additional Assets	Story Mode has new characters. Chatrooms have two new timeline backgrounds, a new secure chat background, a cracked glass overlay, and a new secure lock opening animation. There is a new red static hack effect.

2.4.2 Minor New Features

Feature	Description
Character definitions condensed	Players now only need to define a ChatCharacter for a new character, and their vn_char and phone_char fields will be automatically defined.
Auto-defined Story Mode window background	Users can now supply a character's definition with the `window_color` field to automatically colour the dialogue background in Story Mode.

2.4.3 Fixes

Fix	Description
Phone call replay	Replaying a phone call always begins with the phone audio playing automatically.
Bonus profile pictures	There is better error checking for bonus profile pictures. Characters without a greeting image can still use their profile picture next to the number of hearts earned with them.
Saves track the next day	Save files now track the name of the next day when loading in real-time.

2.4.4 QoL Improvements

Feature	Description
Tutorial Day Introduction	If you hold Ctrl (Skip) during the introductory phone call on Tutorial Day, you will automatically go directly to the home screen, skipping the introduction.
exclude_suffix	Can now exclude the automatic "Day" suffix appended to RouteDay objects.

2.4.5 Other

Feature	Description
Guest grade update	The images used for your guest grade after all party guests have arrived have been updated.

2.5 3.0.0

2.5.1 Major New Features

Feature	Description
StoryMode	Story Mode sections can exist separately from chatrooms and take a trigger time and a title. This also replaces the previous *VNMode* class.
StoryCall	Phone calls can appear in the timeline and be mandatory for story progression. If a story call expires, the character will leave a voicemail. Story calls can exist on their own with a trigger time or can be attached to Story Mode sections or chatrooms. Only one story call per character is allowed, but multiple story calls with different callers can exist on the same timeline item.
msg CDS	A new way of writing chatroom and text message dialogue. Automatically adds text tags and parses emojis as images. Can also add new speech bubbles and fonts to work with this CDS.
text backlog CDS	Allows messages to be added as "backlog" to a character's text message conversation history. Can manually set the date/time the message appears, relative to the current date/time.
compose text CDS	Replaces calls like *call compose_text(r)* and *call compose_text_end("menu_a2")*. Makes it easier to set up and write text messages, and even schedule when they should be delivered.
Choice paraphrasing	Can turn *paraphrase_choices* on or off on a global, per-menu, or per-choice basis to dictate whether or not the program should automatically make the main character say the dialogue in the selected choice.
Bonus profile pictures	Characters have bonus profile pictures that are unlocked based on user-defined conditions or when seen in-game. NPC profile pictures cost 5 heart points with that character to unlock, and player profile pictures cost 1 hourglass. These values are modifiable.
Profile picture callback	When the player changes their profile picture, a special function is called that allows the characters to comment on their profile picture choice (e.g. by phoning them or sending a text message)
scene and *show* extention	*scene* and *show* have been extended to allow chatroom backgrounds to be set with *scene morning* instead of *call chat_begin('morning')*. Most other effects, such as banners, can be shown via *show* commands like *show banner heart* or *show hack effect*.
timed menu CDS	Timed menus are now typed like regular menus. The dialogue before the first choice will dictate how long the timer is. Choices are shown on-screen while the characters continue messaging. A preferences option turns timed menus into regular menus, and a slider allows for timed menu "bullet time" so that the chat speed can be fast but timed menus slow down to give players time to read.
Email system improvements	Email chains can be of an arbitrary length with any number of replies. Failing an email can still continue an email chain. A string parser makes using triple-quoted strings to type emails much more readable. Players can choose to reply to an email later after clicking Reply.
call chat_begin / *jump phone_end*-style calls removed	The program handles the setup and cleanup of all timeline items. All labels can simply begin without further ado and all labels can end with *return*.

2.5.2 Minor New Features

Feature	Description
add_to_album function	Allows you to append a photo to an album after the game has begun.
hide_albums function	If set, allows you to hide albums from the Album screen unless they have an unlocked CG in them already. Takes a list or a single Album.
_branch for parties	The Party can exist on its own with a trigger time, and may have a *_branch* label which will execute when the player enters the party.
Guestbook Hourglasses	The player is awarded 1 hourglass (HG) the first time they view a guest in the guestbook.
invite guest CDS	Replaces *call invite(guest)*.
award heart and *break heart* CDS	Replaces *call heart_icon(s)* and *call heart_break(s)*, respectively. *break heart* can also be written as *heart break*.
enter chatroom and *exit chatroom* CDS	New way of writing *call enter(s)* and *call exit(s)* respectively.
Custom speech bubbles	Can use provided functions to set custom speech bubbles or do things like let Unknown use Ray's speech bubbles.
was_expired variable	This variable is set before the game executes an *after_* label. Allows you to change dialogue depending on whether or not the associated item was participated in or if it expired.
-b and *-thumb* for album images	You can now define thumbnails and *-b* variants on thumbnails for gallery images. These variants will be used for bonus profile pictures.
History screen customization	Can now add backgrounds to routes in the history screen. A rectangular image will automatically have the corners cropped to fit the button. Can also add prologues through a special variable in *variables_editable.rpy*.
.webp and *.ogg* conversion	Most images have been converted to *.webp* to reduce file size. Some audio files have been converted to *.ogg* for similar reasons.
call answer removed	*call answer* is no longer required before a chatroom or text message menu.
custom_route_select_screen	Can easily switch between a custom route select screen and the default one. Default route select improved to have additional graphics.

2.5.3 Fixes

Fix	Description
History cleanup	RouteDays without content don't show up in the History. Single-day routes are centered in the screen.
buyahead	Buying 24 hours in advance while playing in real-time will continue to unlock items up to the 24 hour mark after proceeding through a plot branch in the middle of the purchased 24 hours.
Text message preview	Text message previews can properly handle text tags such as special fonts or sizes.
Sign screen	Sign screen doesn't show hourglasses/heart points if the player was just replaying it or the item was expired.
Pronoun fixes	Capitalized variants of pronoun variables are automatically defined.
Observing improvements	Replaying Story Mode/phone calls during a playthrough now only presents you with the choices you made on that particular playthrough.
Screen shake	Screen shake now works with animated backgrounds. It can be shown via *show shake*.
Profile picture flexibility	Profile pictures can now use solid colours or cropped/transformed images.
Messenger Error screen	The program will try to catch operational errors and award the player an hourglass if something fails but it is able to recover.
Viewing CGs	Viewing CGs full-screen while on the messenger or viewing a text message now simply halts the game state while it is being viewed instead of manually pausing the chat.
Home screen grid fix	The home screen now properly calculates how many pictures it will be showing so the grid is not over- or underfull.

2.5.4 QoL Improvements

Feature	Description
play music and *play sound*	New CDSs replace *call play_music()* and *call play_sfx()*. Can be typed and used the same way as the regular Ren'Py function, but is now compatible with audio captions.
Custom ending screens	Can now define your own ending image and pass the whole string to the *ending* variable upon ending a route.
create_incoming_call	Easy way of creating a new incoming call; intended primarily to be used in combination with profile picture callbacks.
Keyboard shortcuts	Space bar selects most chatroom footer options, like play/pause/answer/save & exit and the Sign button. Left/right arrows decrease/increase the chat speed.
TimelineItem class	All timeline items (chatrooms and VN mode) have been switched to inherit from a base TimelineItem class. The new classes allow for more flexibility such as standalone story calls and story mode sections, and the individual items in a chain can all have a separate *after_* label.
String to filename conversion	Guest names and album names can be converted to a filename which the program can search for to find images associated with that name. *all_albums* definitions can be modified to reflect this.
persistent.testing_mode improvements	Activating *Testing Mode* from the Developer options will now show a button to instantly end a chatroom. It also removes a lot of the confirm-style message and allows you to right-click an item on the timeline to instantly mark it as played for testing purposes.
persistent.unlock_all_option	An option, Unlock All Story, in the Developer options will make all timeline items immediately available. Plot branches can be proceeded through without playing all prior items.
Save slots	Save slots are now paged rather than one long list.
add_choices for spaceship thoughts	Can now append a new choice to a list of spaceship thoughts.
main_character variable	A character defined as the main character will say non-paraphrased dialogue automatically and be moved to the right side of the messenger.
persistent.pv	*pv* (for chatroom speed) is now persistent and will carry over across playthroughs and on the history screens.

2.5.5 Other

Feature	Description
Folder restructure	Engine code has been moved to the *01_mysme_engine* folder. Modifiable code, or code that is expected to be added to by the user, is directly inside the *game/* folder.
Casual and Jaehee Route	A stub implementation of Casual Story and Jaehee Route is implemented as an example. Allows for easy testing of real-time mechanics and demonstrates a complete route definition with branches for multiple endings.
show_empty_menus	Can choose whether timed menus with no choices (e.g. all choices have conditions which evaluate to False) should show their dialogue or be skipped altogether.
Script error screen	The program will try to detect when you have made an error in your script and notify you or direct you to an appropriate wiki page rather than crashing.
missing_label and *missing_image* callbacks	The program will try to correct and/or recover from missing labels or missing images without disrupting the program.
use_timed_menus	*persistent.autoanswer_timed_menus* has been renamed to *persistent.use_timed_menus*. The preference option is titled Timed Menus.

2.5.6 Renamed Variables

Old Name	New Name
post_chat_actions	reset_story_vars
persistent.completed_chatrooms	persistent.completed_story
plot_branch_end()	execute_plot_branch
num_future_chatrooms	num_future_timeline_items
chatroom_hg	collected_hg
chatroom_hp	collected_hp
chat_archive	story_archive
.chatroom_label	.item_label
most_recent_chat	most_recent_item
current_chatroom	current_timeline_item
screen chatroom_item	timeline_item_display
screen chatroom_item_history	timeline_item_history
screen chatroom_timeline	timeline
next_chatroom	check_and_unlock_story
next_chat_time	next_story_time
chat_24_available	make_24h_available
screen chat_select	day_select
screen day_select	day_display

CHAPTER

THREE

CREDITS AND ACKNOWLEDGEMENTS

I'd like to thank the following people for their contributions to this project:

- CheruAngelic for giving me permission to use her art in this program
- Manami for letting me use their art edits
- mvngx.gif for contributing many of the assets used in the program
- Rom for contributing additional art for Vanderwood
- The people of the Lemma Soft forums and Ren'Py Discord for many tutorials, answers, and resources
- RenpyTom for assistance in fixing some of the errors and animation issues I ran into
- The lovely residents of the Mysterious Messenger Discord server for your help testing and your excellent suggestions
- And you, for your support and enthusiasm in testing out the program!

CHAPTER

FOUR

GETTING STARTED

Welcome to Mysterious Messenger! There are a lot of features to try out and different ways to use the program. This page will walk you through the steps to set up Mysterious Messenger and begin using it, as well as explaining some of the available features and what you can create with this program.

4.1 Setting Up

- If you want to run this code, you will need to download the Ren'Py engine: https://www.renpy.org/
- As of 2022-09-15 (the time of this update) the version of Ren'Py used is 8.0.3. Mysterious Messenger is also intended to be compatible with 7.5+, though it's recommended you use v8.0+ to take advantage of Python 3's capabilities.
- Download or clone the most recent release into your Ren'Py Projects Directory and unzip it into its own folder. If you don't know what it is, you can change it from the Ren'Py launcher via `preferences -> Projects Directory`.
  - The current stable version is 3.3.0
- Refresh your Projects list and you should see `mysterious-messenger` listed
- Save the Script Generator spreadsheet somewhere you can find it later
- Join the Mysterious Messenger Discord for update announcements and help with the program.
- **The images and sound files used in this project are not included in the repository. Please contact me directly if you would like to request the assets for personal use.**

4.2 Getting the Program Running

If you've set up everything properly, from the Ren'Py Launcher you should be able to select the project you created from the column on the left and hit 'Launch Project'. If you don't have the images/sound files, you will likely run into several "not found" errors until you've created your own replacements; otherwise, you should be able to go ahead and type in your desired name for the protagonist and check out the "Tutorial Day", which will walk you through some of the available features in the program.

4.3 Useful Built-in Features

Besides just modifying the code, the program has some extra features built into the Settings screen specific to this program. Many are useful when creating new content.

4.3.1 Accessibility Options

On the **Preferences** screen there are several toggles under the header Accessibility Options. These are explained below:

Hacking Effect Turns on/off the various flashing and glitchy "hacked" animations in the program.

Screen Shake Turns screen shake on/off.

Chatroom Banners Turns on/off animation for banners during chatrooms.

Timed Menus If turned off, Timed Menus will function like regular menus, with the chat stopping to show the player an Answer button before proceeding.

Also see the **Timed Menu Speed** slider, explained below.

Animated Icons Turns heart icon and hourglass animations into small text notifications. Note that you can also choose not to award hourglasses in the chatroom at all by unchecking **Receive Hourglasses in Chatrooms** in the Developer settings.

Dialogue Outlines Adds outlines to fonts during Story Mode, phone calls, and chatrooms to make them more readable.

Under the header **Other Settings** are a few additional sliders and options:

Timed Menu Speed If Timed Menus are turned on, this slider will adjust the amount of time the player is given to answer a timed menu. Moving this slider to the left will make messages post slowly, giving the player time to read them and decide on an answer before the timer runs out, and moving the slider to the right will cause messages to post very quickly, resulting in a shorter timer.

By default, this is set to the equivalent of chat SPEED 5. The value of this slider is not affected by how fast your chatroom speed is – you could have the chatroom regularly operate at SPEED 9 but timed menus slow the chat speed down to approximately SPEED 4 until the player either chooses an answer or the timer runs out.

This has no effect if timed menu are turned off.

VN Window Opacity Adjusts the opacity of the dialogue window during Story Mode (VN) sections of the game. If the slider is all the way to the left, the dialogue window will be completely transparent. Putting the slider all the way to the right will make the dialogue window completely opaque. This can be used in combination with **Dialogue Outlines** to make text easier to read.

Background Contrast Adjusts the opacity of the starry night background used in most menu screens. Dragging the slider all the way to the right makes the background completely black.

Finally, under the **Sound** tab in the Settings is an option for audio captions.

Audio Captions If checked, the program will display a notification describing background music and sound effects briefly when the sound is first played.

4.3.2 Dialogue Settings

There are several options which affect dialogue display on the **Preferences** tab:

Text Speed Affects how fast the text displays during Story Mode and phone calls. By default, the slider is all the way to the right, which means dialogue shows up instantly. Moving the slider farther to the right decreases the number of characters per second (CPS) that are shown, so dialogue will show up character-by-character instead of all at once.

Auto-Forward Time This program includes an "Auto" feature for Story Mode as well as phone calls. Setting this slider farther to the right results in a shorter delay between showing new lines of dialogue, and setting it farther to the left gives you more time to read dialogue before the program moves on to the next line.

Skip Unseen Text By default, this option is checked. If unchecked, the program will stop skipping/stop Max Speed when it comes across text you've never seen before. The program remembers which text you have seen across playthroughs.

Skip After Choices By default, this option is checked. If unchecked, the program will stop Max Speed/skipping after you make a choice, and you will need to press the Skip/Max Speed button again to resume skipping after a choice.

Skip Transitions If checked, this causes the program to not show transitions when skipping.

Indicate Past Choices If checked, the program will display a small checkmark in the corner of choices you've picked during past playthroughs.

4.3.3 Other Settings

Modified UI If checked, the program will change some of the UI elements in the game to be more consistent with the turquoise and black colour scheme found elsewhere in the game. It also includes some subtle animation for the choice screens. Does not affect gameplay in any way.

Animated Backgrounds If checked, the chatroom background will be animated. This often includes features such as gently drifting clouds or twinkling stars. The animations take 2-3 minutes to play out fully and then loop for the rest of the chatroom.

4.3.4 Developer Settings

Finally, there are several settings which are helpful when testing a route. They can be found both on the main menu and on the in-game home screen under the button labelled **Developer**.

Testing Mode When Testing Mode is turned on, you will have access to several useful features and conveniences:

- Shows a button to instantly end a chatroom and mark it as played
- Allows you to right-click any timeline item to instantly mark it as played
- Removes several confirm-style messages (such as showing you how many missed calls and unread messages you have when loading a game, or asking for confirmation when proceeding through a plot branch)
- Unlocks all available profile pictures for both the player and the characters
- Delivers any outstanding email replies after proceeding through a single timeline item
- Allows you to replay timeline items (such as chatrooms) multiple times and choose different options (otherwise, re-entering a played chatroom would just show a replay of the original conversation)
  - Any content in the `after_` label of a timeline item is also delivered each time you play the chatroom (rather than only the first time)

Unlock all story When checked, this option will make all timeline items available to play instantly, rather than having to play each item sequentially from beginning to end. This may result in failed plot branch checks, as available timeline items are not marked as played unless you've actually gone through them, but you are allowed to proceed through a plot branch at any time.

To get around this, you can include a check for *if persistent.unlock_all_story* in your plot branch label to branch the story differently when this option is checked.

Real-Time Mode By default, this is unchecked. Timeline items will appear sequentially one after the other after the previous item is played. However, if Real-Time Mode is checked off, then chatrooms and other content will appear at the scheduled time in real-time.

Hacked Effect This turns on/off the messenger "hacked" effects. This particular variable only affects the save file it is activated from, so it is not available to toggle from the main menu (only the in-game developer menu). It causes the timeline items to appear "glitchy" and changes the music on the main menu screen. Some of these glitch effects will not appear for players with the **Hacking Effect** option turned off.

Receive Hourglasses in Chatrooms Unchecking this option will stop awarding the player hourglasses during chatrooms. Currently hourglasses are awarded on a pseudo-random basis when a character posts a special speech bubble from a subset of special bubbles. This option is useful if the chatroom is intended to be seen as a video and not played through, for example.

Use custom route select screen Checking this option will cause the program to use the screen titled `custom_route_select_screen` instead of `route_select_screen` when the player chooses a route at the start of the game. See *Customizing the Route Select Screen* for more.

Prefer local documentation Checking this will cause the script error popup links to open the locally-saved html documentation that comes with the repository, where possible. If a file cannot be found, it will be opened in a web browser instead. This can be useful if you are working offline.

Use pause footer for links This will use the pause footer at the bottom of the screen while waiting for the player to click a link in the chatroom. By default, the game shows a customizable message that tells the player to click the link. See *Stopping the Chat* for more information.

Fix persistent This is an option primarily intended for users updating Mysterious Messenger from older (<2.0) versions to fix issues with saved persistent values. If Ren'Py is complaining about compatibility issues with persistent variables, you can try using this option to fix it.

Documentation Clicking this will open the documentation home page for Mysterious Messenger. By default, this opens the documentation in a web browser. However, you can check off **Prefer local documentation** (above) to open the html files that come with the repository instead. Both the online and offline versions contain the same information.

Reset Albums This will cause the program to forget all persistent variables associated with albums. This includes all the persistent albums as defined in the `all_albums` variable, and will also clear the program's memory of images the player has been shown in-game (typically this will only affect CG images shown during Story Mode). Use this if you want to reset the persistent albums to their original, defaulted value.

This option is only available on the main menu (not in-game).

Chatroom Creator This will open a sub-menu to create a new chat or load an existing one. You can use the chatroom creator to visually put together chatrooms and then watch them play out in-game or export them as code to put into the program.

4.3.5 Updates

By default, Mysterious Messenger will check for updates to the program once a day. You can customize this in several ways by clicking the update icon in the bottom-right corner of the main menu.

Check for updates (once per day) Unchecking this will stop the program from automatically checking for updates. You can still manually check for them with the **Check for updates** button.

Check for prereleases Checking this will include prereleases when the program checks for updates, and will inform the player if a recent prerelease is available. Unchecking this will cause updates to only search for complete releases.

Ignored releases Whenever the program finds a release you haven't updated to yet, you'll get the option to ignore it. Ignoring a release means that you won't receive any more popups informing you of this particular update version. If you've ignored any versions, they will appear listed under the **Check for prereleases** option. You can uncheck any of these releases to stop ignoring them.

Check for updates Clicking this button will cause the program to manually check for any updates that fit the conditions you've specified above. You will receive a popup if it finds a new version you can update to.

Note: Mysterious Messenger requires an internet connection to check for updates. If it cannot connect to the internet and automatic updates are turned on, it will simply silently fail to fetch them (with no adverse effects to the rest of the program).

If Mysterious Messenger finds an update for you, you will see information such as the version number, publish date, whether or not it's a prerelease, and a download link for the new code files. You can also choose to ignore this release. Your current program version will be displayed in the bottom-left corner of the popup.

4.3.6 Uncategorized

This section is for tips or program features that don't have a particular category, but may be useful to players.

- The space bar will activate most of the main buttons in a chatroom. It will switch between play/pause, click the Answer button when it is available, activate the Save & Close button, and click "Sign" on the signature screen.
- The left/right keyboard keys will decrease/increase the chat speed during a chatroom.
- You can swipe photos in a CG album to view the next or previous image without returning to the selection screen. It will automatically skip locked photos.
- You can see the heart points you've earned with each character on your character's profile screen
- If you are on Windows, there is a file called `file_picker.exe` which is included in the assets. This will allow the game to display a file picker and let the player choose an image from their computer to use as a profile picture. It does not currently work for other platforms.

4.4 Program Features

Finally, below is an overview of the various features available within Mysterious Messenger:

- Fully-featured chatrooms
  - Banners, emojis, screen shake
  - "Hacked" effects (e.g. screen tear)
  - Special fonts and bubbles
  - Animated chatroom backgrounds (optional; activated in settings)
  - Heart icons
  - Optional timed menus
  - Messages with clickable links
  - Create-a-chatroom feature to visually put together chatrooms and watch them play out in-game or export them as code to use in the program
- Phone calls
  - **Calls can become available after each timeline item for the player to call the characters**
    - The dialogue can change depending on whether the player is calling back a missed phone call or not
  - Characters can call the player
  - Story Calls, which are a mandatory part of the story on the timeline
  - Voicemail for when a character doesn't pick up
  - Special callbacks that trigger when the player hangs up in the middle of a call
- Text messages
  - Characters can send text messages after any timeline item
  - Schedule text messages to deliver at a particular time, or a time relative to other timeline items
  - Send emojis and CGs through text
  - Have text message conversations play out in real-time like a one-on-one chatroom
  - Create text message backlog before a route begins
- Story Mode (VN)
  - Includes all the main characters' outfits and expressions
  - Jump to Story Mode in the middle of a chatroom for flexible storytelling opportunities
- Emails
  - Automatically unlock guests in the guestbook when you first invite them and when they attend the party
  - Flexible email chains of any length
  - Allows for "recovery" emails where the player made the wrong choice but can continue to email the guest to improve their chances of attending
  - Successfully invited guests are showcased before the party
  - Receive an hourglass for viewing a character in the guestbook for the first time

- Routes
  - Set up plot branches during the story or when the player enters the party
  - Mix and match chatrooms, standalone story mode, and story calls
  - Include different Good/Bad/Normal ends (or your own kind of end!)
  - Previously played story is automatically unlocked in the History screen
- Other features
  - Unlock CG images for a character's album
  - Profile picture callbacks to have the characters react to the player's profile picture choices
  - Spaceship thoughts and occasional prizes from the Honey Buddha chip bag
  - Customizable pronouns (she/her, he/him, they/them) and gender (male/female/nonbinary)
  - Select a chatroom username separate from your regular name
  - Custom ringtones
  - Character greetings on the main menu
  - Real-time and sequential mode
  - Get input from the player for more detailed information
  - Support for multiple screen sizes
  - Custom splash screen

CHAPTER

FIVE

BEGINNER'S GUIDE

If you want to create your own route and some of the technicalities are going over your head, this guide will take you through setting up a new route from start to finish. More specific pages will be referenced throughout. If you already know a bit about the program, you may want to start with *Setting up a Route* instead.

5.1 Getting Started

If you're absolutely new to Ren'Py, I have a tutorial on my website you can follow to get set up with the engine and a code editor: Getting Started With Ren'Py

You may also consider looking through the Ren'Py Quickstart

There are instructions in the `#information` channel of the Mysterious Messenger Discord server on how to download the program and assets to get it up and running.

Once you've downloaded the program and assets (or created your own replacement assets), the first thing you should do is hit `Launch` from the Ren'Py launcher and play through the Tutorial Day to get a feel for what the program is capable of.

> **Warning:** If you tried to launch the game before obtaining or creating the assets, be sure to use **Delete Persistent** in the Ren'Py launcher under the **Actions** header.

5.1.1 Opening the code in a code editor

Next, you need to create a new `.rpy` file. This is where you will write the code that will tell Mysterious Messenger how you want your route to be set up. If you have a program to edit code in, such as VS Code or Atom, you should open that program. Otherwise, you can either download an editor online or tell Ren'Py to download it for you.

Note: If you don't yet have a code editor, I recommend VS Code, which you can install directly from the Ren'Py launcher. The instructions to do so can be found on my website here: Getting Started With Ren'Py

---

`Return` to the main screen of the Ren'Py launcher. Next, in your file explorer, open the `mysterious-messenger/game` folder. To keep things organized, you should create a new folder here. Call it `my_new_route`.

Now you can open your code editing program and use `File -> Open Folder...` to navigate to the `mysterious-messenger/game/my_new_route` folder. It's blank for now, but you will add `.rpy` files to it soon.

5.1.2 Creating a new .rpy file

Inside your code editor, you can use `File -> New File`. Name this file `my_route.rpy`. Don't forget you need to include `.rpy` at the end so the editor and program understand this file contains code for Ren'Py.

5.2 Defining the route

Inside your new `.rpy` file, copy and paste the following code:

```
default my_route_good_end = ["Good End",
    RouteDay('1st'),
    RouteDay('2nd'),
    RouteDay('3rd'),
    RouteDay('4th'),
    RouteDay('5th'),
    RouteDay('6th'),
    RouteDay('7th'),
    RouteDay('8th'),
    RouteDay('9th'),
    RouteDay('10th'),
    RouteDay('Final')]
```

This sets up a variable which is going to contain the information the program needs to understand how to display the route to the player. In particular, this defines the "Good End" of a route. In the history screen, when the player reaches the end of this route the final timeline item will show up under the title "Good End". This guide only covers how to add one ending, but if you want to learn more you can refer to *Plot Branches*.

You can learn more about variables in this series on my website: A Quick Primer on Variables in Ren'Py

5.2.1 Ensuring the route shows up in History

The last variable you defined is just for the "Good End" path of a route. You may want multiple endings or paths, perhaps with different characters or to reflect how the player's choices affected the narrative. This next variable tells the program that the various endings are part of the same "route". For now, you only have one ending, "Good End", but you can add more later.

Beneath the code from earlier, copy and paste the following code:

```
default my_new_route = Route(
    default_branch=my_route_good_end,
    branch_list=None,
    route_history_title="My New"
)
```

This tells the program which endings are associated with this route. `route_history_title="My New"` means that in the History screen, this route will be displayed as "My New Route". The default/longest path (and in this case, the only path) in this route is `my_route_good_end`, and there are no other branches so `branch_list=None` lets the program know there aren't any other branches. How to add additional branches will be covered later.

5.3 Accessing the New Route

Now you need to tell the program to begin your route when the player hits the "Original Story" button from the main menu. To do this, you need to modify the `custom_route_select_screen`, which is found in the file `screens_custom_route_select.rpy`. Open that file in your editor.

The default code for this screen looks like:

```
screen custom_route_select_screen():
    vbox:
        style_prefix 'route_select' # Remove this if you want your own styles
        button:
            ysize 210 # Set the height of the button
            # The image that goes on the left of the button
            add 'Menu Screens/Main Menu/route_select_tutorial.webp':
                align (0.08, 0.5)
            action Start()
            # The box with text on the right side of the button
            frame:
                text "Tutorial Day"
```

The simplest way to modify this is to change the `action Start()` property for the existing button. Normally this tells the program to begin the game at the `start` label. Modify `action Start()` to say `action Start("my_route_introduction")`, and change the line `text "Tutorial Day"` to say `text "My New Route"`. The whole screen will now look like:

```
screen custom_route_select_screen():
    vbox:
        style_prefix 'route_select' # Remove this if you want your own styles
        button:
            ysize 210 # Set the height of the button
            # The image that goes on the left of the button
            add 'Menu Screens/Main Menu/route_select_tutorial.webp':
                align (0.08, 0.5)
            action Start("my_route_introduction")
            # The box with text on the right side of the button
            frame:
                text "My New Route"
```

You can also change the image 'Menu Screens/Main Menu/route_select_tutorial.webp' to something else, though it's perfectly fine to leave it as-is for now.

Now you can test out this code in-game. Make sure you've saved your file, then open up the Ren'Py launcher again and hit **Launch** to start the game. To see your new route select screen, on the main menu, click the **Developer** button in the bottom right corner, then check off the option titled "Use custom route select screen". Now when you start a new game and click Original Story, your custom route select screen will be displayed, though you haven't made an introduction yet so clicking on this button will cause an error.

5.4 Creating the Introduction

Now that you've got a button to start your new route, you need to write an actual introduction for it. You can close the program again while you're editing the code and return to your code editor. To keep things organized, create a new `.rpy` file like you did last time and put it inside your `my_new_route` folder. Call this file `my_route_intro.rpy`.

Now you need to create a label that the program will jump to when the player hits the button on the route select screen from earlier. The action on that button was `action Start("my_route_introduction")`, so this label must be called `my_route_introduction`:

```
label my_route_introduction():
    $ new_route_setup(route=my_new_route, participants=None)
    $ paraphrase_choices = False

    scene night
    play music mystic_chat
    enter chatroom u
    u "This is an example chatroom! You can customize it more later."
    exit chatroom u
    return
```

This is where the program will jump to when the player selects this route to play. Then there are a few lines of setup, which will be explained below.

`new_route_setup` is a special function that sets up the introduction of a route and lets the program know which route it should set up in the timeline. It has two fields:

route The Route variable defined earlier containing the main path and any branching paths for this route.

participants By default, this is `None`, which means that no one starts in the introductory chatroom. If you would like to have characters start in the chatroom, then you can add them here as a list e.g. `participants=[ja, ju, s, y, z]` adds the characters ja, ju, s, y, and z to the chatroom initially before the player arrives.

Next is the line `$ paraphrase_choices = False`. This tells the program that the main character should say the exact dialogue found on the choice buttons after the player has selected that choice. If you want to type out most choice dialogue yourself, then this should be `$ paraphrase_choices = True` instead. Note that you can toggle choice paraphrasing on and off on a per-menu or per-choice basis; see *Paraphrased Choices* for more information. The rest of this tutorial will assume you have `$ paraphrase_choices = False`.

`scene night` tells the program to set up the `night` background for the chatroom. For more information on backgrounds, see *Adding Chatroom Backgrounds*.

`play music mystic_chat` tells the program to play the "mystic_chat" music in the background. This loops automatically. The usual Ren'Py method of playing music has been overwritten to support audio captions, though it supports all the usual features such as fadeout and looping as found in the Ren'Py documentation on audio. For more information, see *Adding New Audio*.

`enter chatroom u` and `exit chatroom u` display messages like "Unknown has entered the chatroom." and "Unknown has left the chatroom." respectively. For more information on these sorts of functions, see *Advanced Chatroom Features*.

Finally, all labels including this introductory chatroom should end with `return`. In this case, it will show the Save & Exit button to the player and return them to the game's home screen.

All of these features and more are covered in *Chatrooms*, including how to write dialogue. You're welcome to expand on this chatroom later with more dialogue.

5.5 Defining Timeline Items for your Route

Now that you've created an introduction for your route, you need to actually put some content in it. Currently the player can select your route and play the introduction (feel free to test it out!) but there won't be any chatrooms or Story Mode sections on their timeline screen. You'll also see an error message informing you that you have no content to your route. It's time to fix that now.

Return to your file `my_route.rpy`. It should contain the code from earlier:

```
default my_route_good_end = ["Good End",
    RouteDay('1st'),
    RouteDay('2nd'),
    RouteDay('3rd'),
    RouteDay('4th'),
    RouteDay('5th'),
    RouteDay('6th'),
    RouteDay('7th'),
    RouteDay('8th'),
    RouteDay('9th'),
    RouteDay('10th'),
    RouteDay('Final')]
```

To create your first chatroom, modify the above code so it now looks like the following:

```
default my_route_good_end = ["Good End",
    RouteDay('1st',
        [ChatRoom("Welcome!", "day_1_chatroom_1", '00:01')
        ]),
    RouteDay('2nd'),
    RouteDay('3rd'),
    RouteDay('4th'),
    RouteDay('5th'),
    RouteDay('6th'),
    RouteDay('7th'),
    RouteDay('8th'),
    RouteDay('9th'),
    RouteDay('10th'),
    RouteDay('Final')]
```

The main thing that has changed is the code after `RouteDay('1st',`, which now has something called a `ChatRoom` object. This object holds information that tells the program information like the title of the chatroom ("Welcome!"), the label where the program can find the chatroom ("day_1_chatroom_1"), and the time the chatroom should appear at ("00:01" aka 1 minute past midnight). A typical RouteDay is made up of a list of these items, along with StoryMode and StoryCall objects (for more, see *Adding Timeline Items*).

Next, you need to define your new chatroom, similar to what you did for the introduction.

5.5.1 Creating the First Chatroom

There are two main ways of writing chatroom dialogue. The most flexible way is to write the script for the chatroom yourself in an `.rpy` file, described below. If you're more of a visual person, though, and some of this is going over your head, you might also try the **Chatroom Creator**, which you can access by launching the program and clicking on the **Developer** button on the main menu. See *Chatroom Creator* for more detailed information on using this. While you will have fewer options to customize the chatroom using the creator as opposed to creating it yourself in code (for example, the chatroom creator cannot make choice menus), you can export your created chatroom as code and modify it further in the generated `.rpy` file to customize it to your needs. However, below we'll explore how you can create chatrooms just from code.

Like before, you should create a new `.rpy` file in order to keep things organized. Call this one `day_1_chatroom_1.rpy`. It doesn't have to be named the same as the chatroom label, but noting the day and chatroom number of this chatroom will help you keep things organized.

Now you need to define the body of the chatroom. First, make a label with the name you wrote earlier for the `ChatRoom` object, namely `day_1_chatroom_1`:

```
label day_1_chatroom_1():
    scene earlyMorn
    play music mystic_chat
    enter chatroom u

    u "Congratulations! You've created your first chatroom."

    menu:
        "That's amazing!":
            u "Isn't it? I'm glad you think so."
        "This is a lot of work.":
            u "I'm sure it will get easier with practice!"
            u "There are plenty of wiki pages to help you out, too."

    u "I'm leaving now. Good luck!"
    exit chatroom u
    return
```

This defines a very basic chatroom with the character "Unknown" (`u`). In this particular chatroom, the player is allowed to make a choice, as defined under the `menu:` code. For more on writing chatrooms and creating choices, see *Advanced Chatroom Features*. There are many more things you can do besides just chatrooms as well, such as having characters send text messages or call the player. For more on those, see the corresponding sections in the documentation.

5.5.2 Creating an Expired Chatroom

There's one last thing you should do before finishing your first chatroom. If the player is playing in real-time, or chooses to exit a chatroom with the back button before they've finished viewing it, the chatroom may "expire". Typically this means the player will then see a version of the chatroom where they don't get to participate in the conversation.

The program automatically looks for this "expired chatroom" using the name of the original chatroom + the suffix `_expired`. So you should define your expired chatroom label beneath the regular chatroom like so:

```
label day_1_chatroom_1_expired():
    scene earlyMorn
    play music mystic_chat
    enter chatroom u
```

(continues on next page)

(continued from previous page)

```
    u "Oh... it appears [name] is not here."
    u "Well, I'll come back later. Bye!"

    exit chatroom u
    return
```

This is the chatroom the player will see if it is expired. It can be as similar or different from the original non-expired chatroom as you like. You may also note the use of `[name]` in the dialogue; this will be replaced with the player's name as it was entered in the profile screen. There are also variables to handle the player's pronouns, as players are free to choose one of she/her, he/him, or they/them pronouns at any time. See *Pronoun Integration* for more.

Note: In this program, there are also standalone Story Mode and Story Calls. Story Mode sections do not expire, but Story Calls *will* expire like chatrooms in real-time or if the player hangs up in the middle of a conversation, and should have an `_expired` label as well.

5.6 Playing Your Route

To play your new route, close the program if open and re-launch it after saving all your open `.rpy` files. Select "Settings" from the main menu and then navigate to the "Others" tab and select "Start Over". Then ensure you have the option titled "Use custom route select screen" checked off in the **Developer** settings button in the bottom right corner of the main menu. Now you can press "Original Story" and click on your new route to play it!

5.7 Tips For Testing and Debugging

When writing a new route, you might run into some error screens from time to time. There are a few steps you can take to help identify the problem:

1) Take a look at the error message. It goes back in what's called "reverse stack order", so the lines at the top are typically less related to the problem than the ones lower down. In particular, you should look at the line just before it says "Full Traceback".

2) Usually you will see an error that reads something like `NameError: name "something" is not defined`. The first part is the type of error - in this case, a `NameError`. There are other kinds of errors, like `ScriptError`, `AttributeError`, and more. The next part tells you a bit about the error itself. Sometimes this will be immediately obvious - for example, `ScriptError: could not find label 'myroute_day3_3'` means that the program can't find the label `myroute_day3_3`, so either you've made a typo somewhere, failed to save the file with the label on it in the right place, or haven't created the label yet. Other times the error is a bit more obtuse.

Tip: One of the most common sources of errors is missing commas or incorrect parentheses. If you see an error that looks like `TypeError: string indices must be integers`, be on the lookout for missing commas or parentheses! Unfortunately the error messages for these sorts of mistakes tend to seem unrelated to the missing commas/parentheses, but make sure you take a good look around the lines of code near where the error is detected to rule it out.

---

3) If the error message isn't particularly clear, the first thing you should usually try is just to quit the game and re-open it. If you've been making lots of script changes with the game open, particularly

if the game is set to Reload or Auto-Reload, occasionally Ren'Py can't figure out how to update the script while the game is still open and it crashes. In particular, if you see an error message that looks like `Exception: Couldn't find a place to stop rolling back. Perhaps the script changed in an incompatible way?`, this is a very common cause of that error. Closing and re-launching the game from scratch will often fix this.

4) Another common source of errors is old save games. If you aren't seeing changes you've made, try starting a new game from the settings rather than loading an old save or hitting "Original Story" (which automatically loads a save). You can do this by going to the settings -> Other -> Start Over.

5) And of course, be sure you're saving any changes you make to `.rpy` files in your project! Ren'Py needs to either reload or re-launch a game to see the changes made in any code files.

If you're still having trouble, feel free to send a message in the Mysterious Messenger Discord to report a bug!

5.8 Next Steps

If you'd just like to play around with some of the features available in chatrooms, you can check out the Chatroom Creator from the Developer options on the main menu. There is an option to export created chatrooms to code which you can then add into your route like the code used earlier. If you'd like to take advantage of the full set of features Mysterious Messenger offers, this may help you see how to write the code for particular chatroom features.

Since many things in the game build off of chatrooms, you should look at the documentation on *Chatrooms*. You should also look at the code for some of the chatrooms and other features included on Tutorial Day, such as `tutorial_5_coffee.rpy` and `tutorial_6_meeting.rpy`, as they have many notes included explaining the various features you will see when you play through those chatrooms.

Once you're comfortable writing and modifying chatrooms, you can look into adding *Text Messages*, then *Phone Calls*, and finally *Emails*. You should also look at *Setting up a Route* for more information on setting up a full route with plot branches and a party. Good luck!

CHAPTER

SIX

GIT SETUP

If you want to easily stay up-to-date with the latest version of Mysterious Messenger without disturbing your existing code, setting up a fork of the Mysterious Messenger repository will allow you to easily merge updates onto your existing code.

What you need:

- A GitHub account (https://github.com/)
- Git and Bash (http://opensourcerer.diy.org/challenge/3)
  - A more in-depth guide on installing Git Bash for Windows: https://www.stanleyulili.com/git/how-to-install-git-bash-on-windows/

6.1 Instructions

- First, go to https://github.com/shawna-p/mysterious-messenger and sign into GitHub.
- In the top-right corner of the page, click **Fork**.
- Open Git Bash (Windows) or Terminal (Mac)
- Change the current directory to the one where you want the cloned project to be. You can type `ls` (lowercase `LS`) to see a list of directories available to you, and `cd Documents` to change to the Documents folder, for example.
- Type `git clone https://github.com/YOUR-USERNAME/mysterious-messenger` where `YOUR-USERNAME` is your GitHub username where you forked the repository. Press Enter.

You should now have a local copy of the mysterious-messenger repository. Next, you'll configure it to get the most recent update.

- Navigate in Git Bash/Terminal to the new directory where you cloned the repository. If you still have Git Bash/Terminal open, you should be able to type `cd mysterious-messenger` to open it.
- Type `git remote -v` and press Enter. You should see

```
$ git remote -v
> origin https://github.com/YOUR-USERNAME/mysterious-messenger.git (fetch)
> origin https://github.com/YOUR-USERNAME/mysterious-messenger.git (push)
```

- Type `git remote add upstream https://github.com/shawna-p/mysterious-messenger.git` and press Enter.
- Type `git remote -v` and press Enter. It should now look like:

```
$ git remote -v
> origin https://github.com/YOUR-USERNAME/mysterious-messenger.git (fetch)
> origin https://github.com/YOUR-USERNAME/mysterious-messenger.git (push)
> upstream https://github.com/shawna-p/mysterious-messenger.git (fetch)
> upstream https://github.com/shawna-p/mysterious-messenger.git (push)
```

Now you should make sure you get the most recent released version, which is currently v3.3.0

- Type `git checkout v3.3.0` and press Enter
- Next, type `git merge tags/v3.3.0` and press Enter.

You should now have v3.3.0 of Mysterious Messenger. In the future, you can use `git checkout v3.3.0` and `git merge tags/v3.3.0` with future version numbers to update to the latest version of Mysterious Messenger.

6.2 Further resources

- How to fork a repo https://docs.github.com/en/github/getting-started-with-github/fork-a-repo
- Configuring a remote for a fork https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/configuring-a-remote-for-a-fork

CHAPTER

SEVEN

CHATROOMS

7.1 Creating a Chatroom

Tip: The following section is for creating chatrooms using Ren'Py scripting and additions added for Mysterious Messenger. If you're interested in the visual chatroom creator, see *Chatroom Creator*.

Note: Example files to look at:

- tutorial_5_coffee.rpy
- tutorial_6_meeting.rpy

A brief overview of the steps required (more detail below):

1. Create a new `.rpy` file (Optional, but recommended)
2. Create a new label for the chatroom.
3. (Optional) Set the background with `scene morning` where `morning` is replaced by whatever time of day/background you want to use.
4. Add background music with `play music mystic_chat` where `mystic_chat` is replaced by the desired music variable.
5. Write the chatroom dialogue.
    1. You may want to use either `Script Generator.xlsx` or the `msg` CDS.
6. End the label with `return`.

The first thing you should do when creating a new chatroom is create a new `.rpy` file and name it something descriptive so it's easy to find it again. For example, you might name it something like `day_1_chatroom_3.rpy` or `day_1_3.rpy`. It's a good idea to put all the files related to a particular route inside a folder for organization.

In your newly created `.rpy` file, begin by making a label for the chatroom:

```
label day_1_1:
```

Don't forget the colon after the label name. Your label name also can't have any spaces in it or begin with a number.

Next, it's time to set up the chatroom background. Note that everything under the label name **should be indented at least one level to the right**. You can look at the example files mentioned above if you're not sure what this means.

New to v3.3.0 is the "autobackground" feature, where Mysterious Messenger will automatically set the background for a chatroom based on the time of day it should trigger at. If you don't set a background, the automatic one will be

used instead. If you want the program to set up the background for you, you can skip over this next section. Otherwise, you can manually set up the background like so:

```
label day_1_1:
    scene morning
```

While chatrooms also use Ren'Py's `scene` statement to show backgrounds, there is a limited number of built-in backgrounds to use. New backgrounds must be defined as described in *Adding Chatroom Backgrounds*. Your background options are:

- morning
- noon
- evening
- night
- earlyMorn
- hack
- redhack
- redcrack
- secure
- rainy_day
- snowy_day
- morning_snow

The background names **are** case-sensitive, so you need to get the capitalization correct. The program will automatically clear the chat history when beginning a new chatroom so new messages begin appearing at the bottom.

Tip: If you're using Mysterious Messenger's autobackground feature to set up the background for you, but you need to change the background manually during a chatroom (e.g. to show the hacked background), if you want to change it back according to the time-of-day rules, you can use the special line:

```
scene autobackground
```

When this line is encountered, the program will automatically switch to whatever background should be used based on the time-of-day this chatroom triggers at.

Note: The autobackground feature is based off of the **trigger time** of the chatroom where it is used, **not** what real-life time the player plays the chatroom at. So, for example, if the player buys back a chatroom that had expired, and that chatroom's trigger time is 8:00am, whether it is 8:00am now or 10:00pm, autobackground will show the `morning` background to the player for that chatroom, since it was set to trigger at 8:00am and that's within the range of the "morning" background time.

Now that the background is set up, you probably want some music. Music is played with the line:

```
play music mystic_chat
```

where `mystic_chat` can be replaced by the name of whatever music you want. There are several files predefined in `variables_music_sound.rpy`. If you want to define your own music, you need to add it to the

`music_dictionary` as well so that it is compatible with audio captions. See *Adding New Audio* for more information.

Next, you'll write the dialogue for your chatroom. See *Writing Chatroom Dialogue*. Finally, to end the chatroom, end the label with `return`:

```
label day_1_1:
    scene earlyMorn
    play music mint_eye
    # Dialogue will go here
    return
```

To learn how to make this chatroom appear in your game, check out *Setting up a Route*.

7.2 Writing Chatroom Dialogue

There are two primary ways of writing dialogue for your chatroom. The first is to use the `msg` CDS, and the second is to use a special spreadsheet to help generate the dialogue for you.

7.2.1 Using the msg CDS

The `msg` CDS helps add effects to your dialogue, such as special speech bubbles or alternative fonts, without sacrificing the readability of the dialogue. Dialogue written with the `msg` CDS looks like the following:

```
msg r "Usually, the program would delay sending text messages and phone calls" sser1
msg r "until after both this chatroom and the Story Mode were played." sser1
msg r "But since v3.0, things are more flexible!" flower_m
msg r "If you declare a route using the new format,"
msg r "you can have text messages and phone calls and the like after any story item,"
msg r "even a chatroom that has a Story Mode attached" glow
```

(Dialogue taken from `tutorial_3_text_message.rpy`)

The `msg` CDS requires a speaker (in the example, `r`) and some dialogue (surrounded by `""`). There are several optional clauses that can be added after the dialogue to affect how it displays.

Fonts

The `msg` CDS has several built-in fonts that can be applied to the text. These are:

Short form	Description
sser1	Sans serif font 1 (Nanum Gothic)
sser2	Sans serif font 2 (Seoul Namsan)
ser1	Serif font 1 (Nanum Myeongjo)
ser2	Serif font 2 (Seoul Hangang)
curly	Cursive font (Sandoll Misaeng)
blocky	Blocky font (BM-HANNA)

You can also use your own fonts with the `msg` CDS. See *Custom Fonts and Bubbles* for more on custom fonts and special bubbles.

In order to use one of the built-in fonts, just include the name of the desired font after the dialogue e.g.

```
msg u "This is example dialogue in the cursive font." curly
msg u "This is in the sans serif 1 font." sser1
```

Note that you can only have one font at a time; including more than one will simply use the last font given. The default font is `sser1`, and dialogue will show up using `sser1` unless you give it a different font.

Emphasis

You can emphasize text in several ways. To use the **bold** version of a font, use the argument `bold` after the dialogue e.g.

```
msg u "This text is bolded." bold
```

Some fonts have **extra-bold** variants as well. These are defined in `variables_editable.rpy` in the variable `bold_xbold_fonts_list`. To make a font extra bold, add the `xbold` clause after the dialogue e.g.

```
msg ja "This text is in the sans serif 2 font, extra bold." sser2 xbold
```

Finally, you can increase the size of the text inside a speech bubble with the `big` argument e.g.

```
msg ju "This text is shown at a bigger size." big
```

You can also combine `big` and `bold` or `xbold` for additional emphasis.

```
msg s "VERY IMPORTANT MESSAGE!!" sser2 big xbold
```

To apply these styles only to parts of a word, you can use the `{b}` text tag for bolded text:

```
msg y "I can't believe you {b}left{/b} me..."
```

And the custom `{big}` text tag for larger text:

```
msg z "I can't believe you think it's {big}not a big deal!{/big}"
```

Underline

To underline all the text in a bubble, you can use either `underline` or `under` as part of the msg CDS:

```
msg u "This text is underlined!" curly underline
msg u "This is also underlined." under
```

If you just want to underline particular words, the built-in `{u}` tag will handle that:

```
msg u "This is {u}underlined{/u} for emphasis." glow
```

Special Bubbles

You can also use special speech bubbles as the background of dialogue. There are several bubbles built in to the program:

Base bubble	Sizes	Additional Notes
cloud	s, m, l	
round	s, m, l	Only available for `ja`, `ju`, `s`, `v`, and `y`. Using a `round` style for `z` or `r` will result in using the `flower` bubble.
round2	s, m, l	Only available for `s`
sigh	s, m, l	Not available for `sa`.
spike	s, m, l	small (s) size only available for `ja`, `ju`, `s`, `y`, and `z`.
square	s, m, l	Not available for `s`.
square2	s, m, l	Only available for `r`.
flower	s, m, l	Only available for `r` and `z`. Previously called `round`.
glow	N/A	Available for all pre-defined characters excluding `m` i.e. `u` and `ri` also have this bubble.
glow2	N/A	Only available for `sa`. An extra variant on the `glow` bubble.

Unless otherwise mentioned, `u`, `ri`, and `m` have no special bubble variants of their own.

For bubbles which have sizes, you must include which size bubble you would like (`s` for small, `m` for medium, and `l` for large) after the name of the bubble e.g. `square_m` or `sigh_l`.

To use a special bubble, add the name of the bubble after dialogue e.g.

```
msg r "This is a bubble with the flower background." flower_m
msg r "And this is a bubble with the glowing background." glow
msg r "These can be combined with other fonts and effects, too!" glow curly big
```

You can also have the characters use each other's special speech bubbles by prefacing the bubble name with their file_id e.g.

```
msg s "I'm using Jumin's cat bubble!" bubble_ju_cloud_l
y "This message uses Zen's flower bubble~" (bounce=True, specBubble="z_flower_m")
```

Finally, you can use your own bubbles with the `msg` CDS. See *Custom Fonts and Bubbles* for more on custom special bubbles and fonts.

Images

The `msg` CDS will automatically detect if dialogue includes a recognized emoji and mark the dialogue as an image accordingly. If you want a character to post an image in the chatroom (such as a CG), then you can use the `img` argument:

```
msg s "cg s_1" img
msg s "I just posted a CG!"
```

You need to follow the rules outlined in *Defining a CG* and *Showing a CG in a Chatroom or Text Message* in order for the program to find the correct image and display it during a chatroom. For CGs, the program will also automatically unlock the image in the gallery.

For emojis, it's sufficient to write:

```
msg s "{image=seven_wow}"
```

where `seven_wow` is the name of the emoji image to be shown. The program will automatically recognize it as an emoji.

Modifying Message Speed

Finally, you can also adjust the speed at which a message is posted. For example, if you want a character to post a bunch of messages in quick succession, you can use the `pv` clause to use a multiplier on the speed at which a message is posted e.g.

```
msg s "These" pv 0.1
msg s "messages" pv 0.1
msg s "are" pv 0.1
msg s "posted" pv 0.1
msg s "quickly!!!" pv 0.1
```

If you have `paraphrase_choices` turned off, you will generally want to add `pv 0` after a message posted by the main character after a menu e.g.

```
menu:
    "Emojis and Images":
        msg m "I want to learn how to use emojis and images." pv 0
        msg u "Emojis and images, huh?" ser1
```

7.2.2 Using the Chatroom Spreadsheet

The second way of writing chatroom dialogue is similar to writing regular Ren'Py script, but passes special keyword arguments in brackets after the dialogue. A spreadsheet, `Script Generator.xlsx`, is included with the program to make this style of writing easier.

The first tab in the spreadsheet is called **Chatroom Instructions** and explains how the **CHATROOM TEMPLATE** tab is used. Where possible, the spreadsheet will try to check off the appropriate boxes depending on what fonts, emphasis, or bubbles you want to use. It will also notify you if you've typed a character's name incorrectly.

The tab **tutorial_6_meeting** has examples directly from the corresponding `.rpy` file of how dialogue for that chatroom was written using the spreadsheet.

In general, you should create a copy of the **CHATROOM TEMPLATE** tab and fill it out with your desired dialogue. Don't forget that messages such as `707 has entered the chatroom` are handled differently – see *Advanced Chatroom Features* for more.

If you've filled out the spreadsheet correctly, you should be able to copy-paste the dialogue from the "What should be filled into the program" column into your script file.

7.2.3 Comparison of Chatroom Dialogue

Both methods of writing dialogue can be mixed and matched freely in-game. For example, if you don't want to add any additional fonts, emphasis, or special bubbles, it can be easier to type out dialogue using just the character variable and their dialogue. Here is a comparison of what dialogue looks like for each method when various fonts, emphasis, and special bubbles are added:

```
# msg CDS
msg u "Extra bold sans serif 2 font" sser2 xbold
# Spreadsheet dialogue
u "{=sser2xb}Extra bold sans serif 2 font{/=sser2xb}"

# msg CDS
msg u "Glowing bubble with blocky font" blocky glow
# Spreadsheet dialogue
u "{=blocky}Glowing bubble with blocky font{/=blocky}" (bounce=True)

# msg CDS
msg z "Bold large curly font with flower bubble" curly bold big flower_m
# Spreadsheet dialogue
z "{=curly}{size=+10}{b}Bold large curly font with flower bubble{/b}{/size}{/=curly}"␣
↪(bounce=True, specBubble="flower_m")

# msg CDS
msg s "{image=seven_wow}"
# Spreadsheet dialogue
s "{image=seven_wow}" (img=True)

# msg CDS
msg m "Dialogue with a speed modifier." pv 0
# Spreadsheet dialogue
m "Dialogue with a speed modifier." (pauseVal=0)

# msg CDS
msg s "cg s_1" img
# Spreadsheet dialogue
s "cg s_1" (img=True)
```

Attention: If you have script from v2.x or earlier, you don't need to modify it to work with the `msg` CDS. You can mix and match script writing styles within the same chatroom.

7.3 Advanced Chatroom Features

Now that you've made a label for your chatroom and filled it with some dialogue, you may want to add additional polish to your chatroom, like allowing the player to make a choice or adding the special `707 has entered the chatroom`-style messages.

7.3.1 Entering and Exiting the Chatroom

To get the message `Character has entered the chatroom`, use the `enter chatroom` CDS:

```
enter chatroom s
```

where `s` is the variable of the character who is entering the chatroom. See *Creating Characters* for a list of the characters currently programmed into the game.

Tip: You can also use `call enter(s)` to have a character enter the chatroom.

To get the message `Character has left the chatroom`, use the `exit chatroom` CDS:

```
exit chatroom s
```

where `s` is the variable of the character who is exiting the chatroom.

Tip: You can also use `call exit(s)` to have a character leave the chatroom.

7.3.2 Updating the Characters' Profiles

Updating the Profile Picture

To update a character's profile picture, you just need to write

```
$ ja.prof_pic = "Profile Pics/Jaehee/jae-2.webp"
```

where `ja` is the ChatCharacter variable of the character whose profile picture you'd like to update. Profile pictures should be 110x110 pixels at minimum, and can optionally have a second, larger version supplied that is up to 314x314 pixels and will be used on the character's profile screen. The program looks for the larger version under the same name with the suffix `-b` e.g. `Profile Pics/Jaehee/jae-2-b.webp`.

You can update a profile picture anywhere – during a chatroom, phone call, text message, etc. To have the profile picture show up in the History screen as well as during regular gameplay, it is recommended that you change the profile picture at the beginning of a label and its associated expired version, if applicable e.g.

```
label day_1_chatroom_1():
    $ r.prof_pic = "Profile Pics/Ray/ray-1.webp"
    play music mint_eye
    enter chatroom r
    msg r "Good morning!" curly glow
```

Updating a character's profile picture will automatically change their portrait on the homepage to have an "updated" background.

Note: There are also bonus profile pictures, which the player can update themselves. For more on that, see *Bonus Profile Pictures*.

Updating a Character's Status

To update a character's status, use

```
$ ju.status = "I wonder what Elizabeth the 3rd is up to..."
```

where `ju` is the ChatCharacter variable of the character whose status you would like to update. Updating a character's status will automatically give their portrait on the homepage an "updated" background to indicate there is new information on their profile page.

You can update a character's status wherever you like inside a label; in the middle of phone calls, at the end of a chatroom, in the `after_` label, wherever.

Updating the Character's Cover Photo

To update a character's cover photo, use

```
$ s.cover_pic = "Cover Photos/rfa_cover.webp"
```

where `s` is the ChatCharacter variable of the character whose cover photo you would like to update. Cover photos should be 750 x 672 pixels in size.

Updating a character's cover photo will automatically give their portrait on the homepage an "updated" background to indicate their is new information on their profile page.

You can update a character's cover photo wherever you like inside a label; during Story Mode, in the middle of an expired chatroom, in a text message, wherever.

7.3.3 Clearing the Chat History

If you would like to clear the chat history so that all previous messages are erased, you can use the `clear chat` CDS:

```
y "Anyway, I should go now."
exit chatroom y
clear chat
scene hack
show hack effect
enter chatroom u
u "Hi, [name]."
```

Note: The chat history is automatically cleared before beginning a new chatroom.

7.3.4 Clearing Chatroom Participants

If you would like to clear all current chatroom participants from the chatroom header, you can use the extra argument `participants` for the `clear chat` CDS:

```
label my_chatroom:
    scene morning
    play music narcissistic_jazz
    enter chatroom z
    z "Has the chatroom been acting up for you too, [name]?"
    msg z "I've been having trouble with it all morning." sigh_s
    clear chat participants
    scene hack
    show hack effect
    enter chatroom u
    u "Ah, excellent."
    u "I've gained access to the messenger."
```

This will both clear the chat history and remove any existing participants from the list at the top of the messenger screen.

This isn't typically used outside of using story mode sections in the middle of chatrooms (see *Including a Story Mode During a Chatroom*) or linking together "separate" chatrooms as part of a scene for something like an After End.

7.3.5 Resetting Chatroom Participants

Similar to the above section on clearing chat participants, there is also a special function provided to reset chat participants even without jumping to Story Mode and back, for example, as part of a montage of chatroom sequences. The function is called `reset_participants` and takes one parameter, a list of ChatCharacters to add back to the participants list.

```
u "This is the end of the first chatroom."
$ reset_participants([z, s, m])
s "Hey [name], Zen, you haven't seen anything suspicious lately, right?"
```

Note: You will need to add `m` to the participant list yourself if you would like the player to appear in the participants list after the reset, regardless of whether the chatroom was expired or not.

7.3.6 Providing Choices

During a chatroom, you may want to allow the player to make a choice. This can be accomplished with Ren'Py's built-in menu system:

```
msg s "What kind of food do you eat?"
menu:
    "I like soup.":
        msg s "You like soup?"
    "I eat a lot of junk food.":
        msg s "lolol same."
msg s "But you should have a balanced diet, unlike me~" glow
```

Note that for this menu, it is assumed that `paraphrase_choices` is turned on for this route. This means that the main character will automatically say the exact dialogue on the chosen choice. If `paraphrase_choices` is turned off, the menu might look something like this:

```
msg s "What kind of food do you eat?"
menu:
    "I like soup.":
        msg m "I like soup." pv 0
        msg s "You like soup?"
    "I eat a lot of junk food.":
        msg m "I eat a lot of junk food." pv 0
        msg s "lolol same."
msg s "But you should have a balanced diet, unlike me~" glow
```

You can also turn `paraphrased` on or off on a per-menu or per-choice basis. For more on paraphrasing, see *Paraphrased Choices*.

You can add as many choices as you want to the menu, though only 5 options will fit on the screen at once. All code indented after a choice will only be run if the player picks that choice. So, only a player who chose `"I like soup"` will see the line `"You like soup?"`. Anything indented at the same level as the menu will be run regardless of the choice made, so the player will see the line `"But you should have a balanced diet, unlike me~"` regardless of whether they said they like soup or eat junk food.

Warning: You can use the `TAB` key to indent your code an additional level to the right, but make sure your code editor is using spaces to indent code. Otherwise, you will get errors complaining about "tab" characters in your code.

You'll also notice in the "paraphrased" version of the menu, the main character (`m`) has the clauses `pv 0` after their dialogue. This tells the program to not wait at all before posting the MC's message. Usually the program will pause before posting a message to simulate "typing time", but you want the player's choice to appear right away after a choice, so you should include `pv 0` if you're using the `msg` CDS, or `(pauseVal=0)` if you're using the spreadsheet style.

7.3.7 Showing a Chatroom Banner

There are four special banners included in the program. The available banners are:

- lightning
- heart
- annoy
- well

You can show a banner with the code:

```
show banner lightning
```

Note that while the order of `banner` and `lightning` don't matter, the name is case-sensitive. So, you could also use `show lightning banner` but `show Lightning banner` would not work.

Note: These images are not displayed for players who have toggled the **Chatroom Banners** setting off.

7.3.8 Showing a Heart Icon

Awarding a heart point

To show a heart icon to the player and award them "heart points" associated with a particular character, use

```
award heart s
```

where `s` is the variable of the character whose heart icon you'd like to show. The player will receive one heart point with that character when this code is run. See *Creating Characters* for a list of the characters built-in to the program and how to add your own character.

Note: Ray (`r`) and Saeran (`sa`) share heart points. So, if you award a heart point for Saeran via `award heart sa`, Ray will also receive 1 heart point.

By default, each `award heart` statement gives one heart point to the specified character. You can award multiple at the same time, however, by providing a number e.g. `award heart u 10` would give Unknown 10 heart points.

There is also a second optional argument to `award heart`:

```
award heart ju bad
```

`bad` tells the program that this heart is a "bad" heart point, and should count towards a bad ending. In-game, a `bad` heart appears the same as a normal heart point. You can use this method to count the number of choices a player makes that would lead towards a bad ending, and then check whether the player made more "good" or "bad" ending choices when they reach a plot branch. See *Plot Branches* for more information on this.

Removing a heart point

If the player makes a choice a character doesn't like, you can cause them to lose a point with a character by showing a "heart break" icon.

```
break heart ju
```

where `ju` is the variable of the character whose heart break you'd like to show. This will **always** subtract points from the character's "good" heart points and never the "bad" points. Note that losing heart points in this way does not subtract from the heart point totals the player uses to unlock additional profile pictures for a character (See *Bonus Profile Pictures*).

Tip: Both `break heart ju` and `heart break ju` will show the heart break animation for the character `ju`. You can't switch the word order for `award heart` though!

Like with awarding heart points, you can take away multiple heart points by adding a number at the end e.g. `break heart s 4` would remove 4 heart points from 707.

7.3.9 Awarding an Hourglass

Hourglasses are randomly awarded to players when certain bubbles (specified in `hourglass_bubbles` in `variables_editable.rpy`) are used in chatroom dialogue so long as `Receive Hourglasses in Chatrooms` is active from the Developer options. However, you can also manually award the player hourglasses with a special function, `award_hourglass`. The function takes two fields:

random False by default. Set this to True if you would like there to be a random chance of getting an hourglass even when this function is triggered (e.g. you have a specific message you'd like to potentially award an hourglass, but it shouldn't be guaranteed).

force False by default. Set this to True if you'd like to force the player to receive an hourglass in all circumstances. Note that this isn't the same as having `random=False`; `force` means that even if the player is viewing an expired chatroom or watching a replay from the History, they will receive an hourglass, *including if* you have `Receive Hourglasses in Chatrooms` turned off from the developer settings.

This option is largely intended for non-playable gameplay such that you might record a video of. If you'd simply like to guarantee the player receives an hourglass in the event that it is possible to receive them based on how they're viewing the current story item, then you should leave both `random` and `force` as False.

In the game, you might use it as follows:

```
msg z "I hope I'll see you soon, [name]~" flower_m curly
$ award_hourglass(random=True)
msg z "I should go get ready."
```

This will randomly award the player an hourglass when they see the line "I hope I'll see you soon, [name]~" (that is, on some playthroughs they will receive an hourglass, and on others they will not). They won't see this hourglass if they're viewing this chat from the History or on a rewatch.

7.3.10 Showing the Hacked Scroll Effect

To show the scrolling "hack" effect during a chatroom, use the line

```
show hack effect
```

or alternatively,

```
show redhack effect
```

for the red version of the scrolling hack image. You can reverse the direction of the scrolling with `reverse` e.g.

```
show hack effect reverse
exit chatroom u
show redhack effect reverse
```

There is also a "red static" effect which can be shown with:

```
show red static effect
```

This can also be reversed by adding the *reverse* tag:

```
show red static effect reverse
```

Note: These effects are not displayed for players who have toggled the **Hacking Effects** setting off.

7.3.11 Showing the Secure Chatroom Animation

To show the "secure chatroom" animation, use the line

```
show secure_anim
```

Note: If you're using this effect, you may also want to change the background of the chatroom box on the timeline screen. You can do this with the `box_bg` parameter when defining a ChatRoom for the route e.g.

```
ChatRoom('Hacking', 'hack_example', '20:41', box_bg='secure')
```

7.3.12 Shaking the Screen

To shake the screen, use the line

```
show shake
```

Screen shake is compatible both with regular backgrounds and with the animated backgrounds.

Note: This effect is not displayed for players who have toggled the **Screen Shake** setting off.

7.3.13 Showing the Cracked Overlay

There is also a "cracked" overlay you can layer on top of any chatroom background (including animated backgrounds). It gives the appearance of cracked glass on top of the background. After showing your background, show the crack effect like:

```
scene redhack
show screen_crack
```

7.3.14 Sending Links

Characters may also send links in the chatroom. You can provide an action for a link message, which will be executed when the player clicks on it. Some convenience functions are provided to make some of these actions simpler.

The simplest way to show a link looks like:

```
va "Click Link" (link_title="Password")
```

There are several fields you can provide to a link message to customize it. You must provide at least one of these fields in order for a message to be considered a link.

link_title By default, links do not have titles. If provided, the title will be shown in a smaller size above the link's text inside square brackets (so, `link_title="Password"` appears in-game like `[Password]`.

e.g. "Address"

link_img By default, this is an image of a house. It is 81x81 pixels but can be slightly larger or smaller. You can also provide a general displayable such as a Transform or an AlphaMask.

If you don't want an image at all, this should be `Null()`, so `link_img=Null()` will prevent the link message from having an image.

e.g. "Bubble/link_house_btn.webp"

link_action By default, a link button will not have any action at all and will not be interactable. See *Link Actions* for more action examples.

e.g. ShowCG('common_1')

link_text Optional. If this isn't provided, the link text takes the character's dialogue (so in the line `va "Click me" (link_action=ShowCG('common_1'))`, the `link_text` is "Click me". If the dialogue is empty, it will be the phrase "Click Link".

e.g. "Click Link"

Note: You can use the `msg` CDS for links as well, though it can't take any additional special parameters such as font, bounce/glow, image, or specBubble. A msg version of a link post might look like:

```
msg va "Click Link" (link_title="Password", link_img=Null(), link_action=JumpVN(
↪'unlock_door'))
```

Link Actions

Link messages can take any action you want. By default, only a `ShowCG` action (which shows a CG image) can be clicked more than once. All other actions (including multiple actions in a list) cause the link button to be insensitive after the action has executed once.

Some special link actions include:

ShowCG Takes one argument, the name of the CG image to show. This follows the same naming rules as *Defining a CG* and *Showing a CG in a Chatroom or Text Message*, so if you have:

```
image cg common_4 = "CGs/common_album/cg-4.webp"
default common_album = [
    GalleryImage("cg common_1"),
    GalleryImage("cg common_2"),
    GalleryImage("cg common_3"),
    GalleryImage("cg common_4")
]
```

and you want to show "cg common_4", then you should use the action `ShowCG("common_4")`.

This will also take care of automatically unlocking the CG in the player's album.

JumpVN This action will take the player to a Story Mode (VN) section before returning to the chat. It works the same way as `call vn_during_chat` as explained in *Including a Story Mode During a Chatroom*. The first argument should be the name of the label the program should jump to for the story mode.

It also takes all the same arguments, including `clearchat_on_return`, `new_bg`, `reset_participants`, and `end_after_vn`.

e.g. `JumpVN("my_story_label", end_after_vn=True)`

Note: Unless you pass the argument `end_after_vn=True` to JumpVN, when the Story Mode ends the game will return to the chatroom exactly where the chat left off before the player clicked the link which took them to Story Mode.

This means if you have a chatroom like:

```
u "Click Link" (link_title="Password", link_action=JumpVN("prologue_unlock_door"))
u "That's the password for the door."
# Player clicks the link before this next message appears
u "Try to use it, okay?"
```

If the player clicks the link where indicated, when they return to the chatroom, Unknown will post the message "Try to use it, okay?" and the rest of the chatroom will proceed as normal. It's the equivalent of:

```
u "That's the password for the door."
call vn_during_chat('prologue_unlock_door')
u "Try to use it, okay?"
```

However, with the link, the player **does not** have to click the link to proceed; the chat will simply continue even if they do nothing. If you want to stop the chat to ensure the player clicks the link, see *Stopping the Chat*.

LinkJump(label=None, is_menu=False) Similar to `JumpVN`, this action acts as a jump between labels. Unlike `JumpVN`, however, the jump is intended to be within the same chatroom, rather than jumping to a Story Mode (VN) section.

`LinkJump` takes two optional parameters. The first, `label`, is the name of a label or menu to jump to when the link is pressed. If left blank, this acts like `ContinueChat` if the chat is stopped, or does nothing if the chat is not stopped.

`is_menu` can be set to True if you would like this LinkJump to jump to a menu without showing the answer button at the bottom of the screen. For example:

```
u "Click Link" (link_title="Address", link_action=LinkJump('getaddress', is_
↪menu=True))
stop chat
menu getaddress (paraphrased=True):
    "(Go to the sent address).":
        pass
```

When the player presses the link, they will immediately see the option to (Go to the sent address) as though they had pressed the answer button on a menu.

ContinueChat This action will cause the chat to continue if it was previously stopped.

IfChatStopped(true_action, false_action=None) This is a special action that takes two parameters. The first is the action (or list of actions) to perform if the chat is stopped when this button is clicked. It can also be `None` to do nothing if the chat is stopped.

The second parameter is the action to perform if the chat is not stopped (see the section below for more on stopping the chat). This parameter can also be `None` or omitted to do nothing if the chat is not stopped.

e.g.

```
link_action=IfChatStopped([ShowCG('common_2'), ContinueChat()], ShowCG('common_3
↪'))
```

This causes the CG "common_2" to be shown to the player if the chat is paused before continuing the chat. Otherwise, if the chat is not paused, clicking the link will show the "common_3" CG instead.

This action is useful if you only want the player to be able to proceed by clicking a link once the chat is stopped e.g.

```
    link_action=IfChatStopped(JumpVN('drive_scene_2'))
```

If used as the link action, the player can only click on the link when the chat is stopped.

Stopping the Chat

If you want to stop the chat to wait for the player to click on a link, you can do so with `stop chat` e.g.

```
a "Click Link" (link_title="Password",
    link_action=JumpVN('unlock_door_password'))
stop chat
```

This will prevent any further messages from being posted and displays the text "Click the link to proceed" at the bottom of the screen. If you would like to change the text, at the bottom, you can also provide a string after `stop chat` like:

```
a "Click Link" (link_title="Password",
    link_action=JumpVN('unlock_door_password'))
stop chat "Click the link to continue"
```

There is also an optional toggle in the **Developer Settings** called **Use pause footer for links** which will show the pause button at the bottom of the screen while waiting for the player to press a link. Note that if you do choose to enable this, it is less clear to the player what they should do to proceed with the story when the chats stops to wait for them to click a link.

Warning: If you use the `JumpVN` action, the chat will automatically resume after it returns from the VN label. However, if your action is doing something like viewing a CG or setting a variable, you **must** include the action `ContinueChat()` as the final action in your link_action parameter:

```
s "Add to Contacts" (link_title="Contact Info", link_img=Null(),
    link_action=[CConfirm("Phone number added to contact list."), ContinueChat()])
```

Note: `CConfirm` is a special action which shows a confirmation prompt to the user. In this case, it displays the given message and requires the player to press Confirm to dismiss it.

7.4 Custom Fonts and Bubbles

Mysterious Messenger supports user-defined fonts and special bubbles through customizable variables and functions. These work with the `msg` CDS as well as the text tag approach used for the spreadsheet dialogue.

7.4.1 Custom Fonts

To add your own font to the game, first you need to add it to the `all_fonts_list` found in `variables_editable.rpy`:

```
define all_fonts_list = ['sser1', 'sser2', 'ser1', 'ser2', 'curly', 'blocky']
```

In general you should **not** replace this list; just add your new font to the end. For this example, a font called "cursive" will be added.

```
define all_fonts_list = ['sser1', 'sser2', 'ser1', 'ser2',
    'curly', 'blocky', 'cursive']
```

Now you need to add it to the `font_dict`:

```
define font_dict = { 'curly' : gui.curly_font, 'ser1' : gui.serif_1,
        'ser1b' : gui.serif_1b, 'ser1xb' : gui.serif_1xb,
        'ser2' : gui.serif_2, 'ser2b' : gui.serif_2b,
        'ser2xb' : gui.serif_2xb, 'sser1' : gui.sans_serif_1,
        'sser1b' : gui.sans_serif_1b, 'sser1xb' : gui.sans_serif_1xb,
        'sser2' : gui.sans_serif_2, 'sser2b' : gui.sans_serif_2b,
        'sser2xb' : gui.sans_serif_2xb, 'blocky' : gui.blocky_font,
        'cursive' : "fonts/cursivefont.ttf"
    }
```

"fonts/cursivefont.ttf" should be the path to the `.ttf` or `.otf` file of your desired font.

If your font has bold and/or extra bold variants, you will also add it to the `bold_xbold_fonts_list` variable:

```
define bold_xbold_fonts_list = ['sser1', 'sser2', 'ser1', 'ser2', 'cursive']
```

And you will need to specify how the bold and extra bold variants should be mapped. The bold version of a font is always the name of the font in the `all_fonts_list` + `b`, and the extra bold is the name + `xb`.

```
define font_dict = { 'curly' : gui.curly_font, 'ser1' : gui.serif_1,
        'ser1b' : gui.serif_1b, 'ser1xb' : gui.serif_1xb,
        'ser2' : gui.serif_2, 'ser2b' : gui.serif_2b,
        'ser2xb' : gui.serif_2xb, 'sser1' : gui.sans_serif_1,
        'sser1b' : gui.sans_serif_1b, 'sser1xb' : gui.sans_serif_1xb,
        'sser2' : gui.sans_serif_2, 'sser2b' : gui.sans_serif_2b,
        'sser2xb' : gui.sans_serif_2xb, 'blocky' : gui.blocky_font,
        'cursive' : "fonts/cursivefont.ttf",
        'cursiveb' : "fonts/cursivefont-bold.ttf",
        'cursivexb' : "fonts/cursivefont-xbold.ttf"
    }
```

In this example, the bold is called "cursiveb" and the extra bold "cursivexb". Note that you will need an entry for **both** bold and extra bold, even if you only have one font. In that case, you can set the file path for both entries to the same font.

Finally, to use the font as a text tag, you need to define a style for it. Existing styles can be found in `style_definitions.rpy`:

```
style cursive:
    font "fonts/cursivefont.ttf"

style cursiveb:
    font "fonts/cursivefont-bold.ttf"

style cursivexb:
    font "fonts/cursivefont-xbold.ttf"
```

Now you can use the font while writing chatroom or text message dialogue. If you use the `msg` CDS, you need to preface the name of the font with the tag `font` e.g.

```
msg s "This is a new font!" font cursive
msg s "You can add other effects, too!" big font cursive xbold
```

And text tags work the way they would with the predefined fonts:

```
s "{=cursive}This is a bit more verbose.{/=cursive}"
s "{=cursivexb}Which method you use depends on your preference.{/=cursivexb}"
```

7.4.2 Custom Bubbles

You can also add your own custom bubbles to work with the `msg` CDS and spreadsheet dialogue method, or expand the functionality of existing bubbles.

To add a new bubble, first you must add it to the `all_bubbles_list` in `variables_editable.rpy`. For this example, three bubbles called "spooky_s", "spooky_m", and "spooky_l" will be added (for small, medium, and large variants).

```
define all_bubbles_list = ['cloud_l', 'cloud_m', 'cloud_s', 'round_l',
    'round_m', 'round_s', 'sigh_l', 'sigh_m', 'sigh_s', 'spike_l', 'spike_m',
    'spike_s', 'square_l', 'square_m', 'square_s', 'square2_l', 'square2_m',
    'square2_s', 'round2_l', 'round2_m', 'round2_s', 'flower_l', 'flower_m',
    'flower_s', 'glow2', 'spooky_s', 'spooky_m', 'spooky_l']
```

Next, you need to define a base style for the bubble. Typically the most important property is the padding. First, define a general style:

```
style spooky_s:
    padding (25, 25, 25, 25)

style spooky_m:
    padding (25, 25, 25, 25)

style spooky_l:
    padding (25, 25, 25, 25)
```

This defines a style for the spooky bubbles with 25 pixels of padding on the left, top, right, and bottom of the bubble. These numbers should, of course, be adjusted to suit the actual bubble image you're using.

Next, if any characters have a variant of this bubble that requires different styling, you can define a style for them. For example, if the padding for `s`'s spooky_m bubble is different, you can create an `s_spooky_m` style:

```
style s_spooky_m:
    is spooky_m # This inherits the spooky_m padding
    top_padding 35
```

`is spooky_m` tells the style to inherit from the original `spooky_m` style; this is optional, but can be helpful. In this case, it means that the `s_spooky_m` bubble has a left/right/bottom padding of 25, as in the `spooky_m` style, but the `top_padding` is now 35.

Finally, you need to define an offset for each of your new bubbles. This is how far the bubble should be relative to the top left corner of the character's profile picture (for bubbles posted on the left side of the messenger). All the existing bubble offsets can be found in `gui.rpy`. For the `spooky_` bubbles, three variables are needed:

```
define gui.spooky_s_offset = (140, 38)
define gui.spooky_m_offset = (130, 38)
define gui.spooky_l_offset = (140, 32)
```

Note that these are the name of the bubble + the suffix `_offset`. You can also define specific offsets for characters using their file_id. e.g. if `em`'s spooky_m bubble needs to be a bit further to the right than the rest of the characters', you can define the value `define gui.em_spooky_m_offset = (150, 32)`.

And that's all! To show this special bubble in-game, you must ensure you have the appropriate images defined inside the "Bubbles/Special/" folder.

- The image extension must be one of `.webp`, `.png`, or `.jpg`
- If every character uses the same bubble, it is sufficient to have a "Bubbles/Special/spooky_s.webp" image. Otherwise, you will need an image with the file_id of each character who has a variant e.g. "Bubbles/Special/ja_spooky_s.webp", "Bubbles/Special/ju_spooky_s.webp", etc.

Then you can write dialogue which will show the bubble in-game e.g.

```
ju "This will use the spooky_m bubble." (bounce=True, specBubble="spooky_m")
msg ja "This message will also use the spooky_m bubble" bubble spooky_m
```

Unlike the predefined bubbles, custom bubbles must be prefixed with `bubble` as in `bubble spooky_m` when used in the `msg` CDS.

Tip: You can also add the name of your bubble to the `hourglass_bubbles` variable in `variables_editable.rpy`. This means that when a character uses that bubble in-game, it has a chance of awarding the player an hourglass.

Typically only the large (`l`) version of a bubble is added to this list e.g. `"spooky_l"`

7.4.3 Extending Custom Bubbles

Besides adding your own bubbles, you can also extend the functionality of existing bubbles (or your own custom bubbles) via some special provided functions in `variables_editable.rpy`. There are three such functions. All are passed a `msg` variable, which is a ChatEntry object with the following fields:

who The ChatCharacter object of the sender of the message.

e.g. `s`

what The contents (dialogue) of the message.

e.g. "Have you heard from V lately?"

thetime A MyTime object with information on the time the message was sent at (in real-time)

img True if this message contains either an emoji or a CG; False otherwise.

bounce True if this message should "bounce" in as its animation. This is True if the message uses the "glowing" style of bubble, and also True if a special speech bubble is used (such as "cloud_m"). It is generally False for images and the default speech bubble.

specBubble The special bubble. Typically this is equal to something like "sigh_s" or "spike_l". However, you can pass in particular strings and use this function to take care of what background image it should evaluate to.

Custom Bubble Background Function

Inside `variables_editable.rpy` is a function called `custom_bubble_bg` under the **CUSTOM MESSENGER ITEMS** header. Each chatroom message is passed to this function, which allows you a chance to check for certain conditions and return particular bubble backgrounds.

> **Warning:** You must ensure any new bubbles you allow characters to use (such as a third glowing bubble variant) are added to the `all_bubbles_list` and have styles defined for them as described above, and aren't just defined in the custom bubble function.

You can also return general displayables, such as a Frame(), inside the `custom_bubble_bg` function. For example, you could give Emma (from the new character examples) a second "glowing" bubble variant:

```
def custom_bubble_bg(msg):

    if msg.who.file_id == "em" and msg.specBubble == "glow2":
        return Frame("Bubble/Special/em_glow2.webp", 25, 25)

    return False
```

This returns the "Bubble/Special/em_glow2.webp" image, formatted as a frame with borders 25 pixels wide. The inside of the bubble will expand to be large enough to accommodate the text (this is how the regular speech bubbles and glowing bubble variants are defined and used).

You could then write dialogue to use this bubble like:

```
em "This goes in my glowing bubble." (bounce=True, specBubble="glow2")
msg em "As does this message." bubble glow2
```

Custom Bubble Style Function

Although the program will try to pick out specific styles based on the name of the special speech bubble used, there may be some cases in which you want to apply a particular style to a bubble. The `custom_bubble_style` function in `variables_editable.rpy` will let you return a particular style.

For example, you might want to apply special styling to `em`'s second glowing bubble variant from the last example:

```
def custom_bubble_style(msg):

    if msg.who.file_id == "em" and msg.specBubble == "glow2":
        return "em_glow2_style"

    return False

style em_glow2_style:
    padding (30, 40)
```

This function should return a string that corresponds to the name of a style. For example, if you want a bubble to use the style `style my_special_bubble`, then you need to return the string `"my_special_bubble"`.

Custom Bubble Offset Function

This function allows you to fine-tune the offset of a message, relative to the top-left corner of the message box (typically the top-left corner of the character's profile picture). For example, if you wanted to use special speech bubbles for the character on the right side of the messenger (typically the MC), you would need to adjust these values appropriately. This function is expected to return either False or a tuple of (x, y) integers for the position of this bubble.

An example utilizing all three custom functions that allows the right messenger to use Jumin's large and medium-sized "cat" bubbles might look like the following:

```
def custom_bubble_bg(msg):
    # If the messenger is on the right side (usually the player), flip
    # the background of the special cat bubble
    if msg.who.right_msgr and msg.specBubble == "ju_cloud_l":
        return Transform("Bubble/Special/ju_cloud_l.webp", xzoom=-1)
    elif msg.who.right_msgr and msg.specBubble == "ju_cloud_m":
        return Transform("Bubble/Special/ju_cloud_m.webp", xzoom=-1)
    return False

def custom_bubble_offset(msg):
    # If the messenger is on the right side, adjust the offset for
    # the two cat bubbles
    if msg.who.right_msgr and msg.specBubble == "ju_cloud_l":
        return (650, 10)
    elif msg.who.right_msgr and msg.specBubble == "ju_cloud_m":
        return (640, 39)
    return False

def custom_bubble_style(msg):
    # Ensure the bubbles use their original styling rather than styling
    # for the main character's bubbles
    if msg.who.right_msgr and msg.specBubble == "ju_cloud_m":
        return 'ju_cloud_m'
    if msg.who.right_msgr and msg.specBubble == "ju_cloud_l":
        return 'ju_cloud_l'
    return False
```

7.5 Adding Chatroom Backgrounds

Chatroom backgrounds must be defined in a particular way in order to be compatible with the `scene` and `show` statements to display them.

Existing background options are:

- morning
- noon
- evening
- night
- earlyMorn

- hack
- redhack
- redcrack
- secure
- rainy_day
- snowy_day
- morning_snow

The background names **are** case-sensitive, so you need to get the capitalization correct.

For this example, you'll see how the `rainy_day` background was set up.

To add a new background, inside `variables_editable.rpy` under the header **CUSTOM MESSENGER ITEMS** are several variables. First, you need to define the image that will be used as the background. This should be `bg` + the name of the image as you want to write when displaying it in a chatroom:

```
image bg rainy_day = "center_bg:Phone UI/bg-rainy-day.webp"
```

The `center_bg:` before the file path to the image file itself is optional, but will center the image on the screen if it is not exactly 750x1334 pixels. If your image is exactly 750x1334 pixels, then just `image bg rainy_day = "Phone UI/bg-rainy-day.webp"` would be sufficient.

You can put this image definition wherever you like, though it may make sense to keep it under the **CUSTOM MESSENGER ITEMS** header alongside the background variable definitions.

Next, add the background name to the `all_static_backgrounds` list:

```
define all_static_backgrounds = ['morning', 'noon', 'evening', 'hack',
        'redhack', 'night', 'earlyMorn', 'redcrack', 'secure', 'rainy_day']
```

Note that you do **not** add the "bg" part to the name of your background.

Next, if the characters' names should appear in black during a chatroom (as is typical of the lighter backgrounds, such as "morning" and "noon"), you also need to add it to the `black_text_bgs` list:

```
define black_text_bgs = ['morning', 'noon', 'evening', 'rainy_day']
```

Otherwise, if the background is not in this list, the characters' names will be displayed in white.

7.5.1 Adding an Animated Background

If you would also like to add an animated version of your new background, you need to define a specific screen for it. First, however, you will add the background to the `all_animated_backgrounds` list in `variables_editable.rpy`:

```
define all_animated_backgrounds = ['morning', 'noon', 'evening', 'night',
            'earlyMorn', 'rainy_day']
```

Next, you need to define the screen which will have the animated background. The existing backgrounds are defined in `screens_backgrounds.rpy`.

Because the new background is called "rainy_day", you need to create a new screen called `animated_rainy_day`:

```
screen animated_rainy_day():
    zorder 0
    tag animated_bg

    # add images here
```

You **must** include the lines `zorder 0` (to ensure the animated backgrounds appears behind other screen elements) and `tag animated_bg` (to ensure only one animated background is shown at once, and so that the background is properly cleared at the end of a chatroom).

You can then add animated elements to the screen. For a rainy day background, for example, you might take advantage of the existing `slow_pan` transform for clouds, and perhaps add raindrops which fall from the top of the screen to the bottom. As the exact requirements for a given animated background will vary substantially depending on what sort of background you want to add, this part is largely up to your own skill with screen language. You can see how the existing animated backgrounds are put together in the `screens_backgrounds.rpy` file.

Warning: If you would like to include an animated background, it must have a corresponding "static" version that can be displayed to users who don't have animated backgrounds turned on.

7.6 Timed Menus

Note: Example files to look at:

- tutorial_5_coffee.rpy
- tutorial_6_meeting.rpy

Mysterious Messenger includes a new kind of menu which will display answers at the bottom of the screen for a brief period of time while the characters continue to post messages to the chat. The player can choose an answer at any time before the timer runs out, or refrain from choosing anything and stay silent. The time the player has to choose a reply depends on the length of the dialogue before the menu. Currently, this type of menu is **only** available for chatrooms.

An example timed menu may look like the following:

```
u "Hello, [name]!"
timed menu:
    u "I'm working on a UI for the choices you see at the bottom of the screen."
    u "It's typed almost identically to regular menus this time,"
    u "but with some convenience features I think you'll like."
    u "You should try clicking the options before the timer runs out."
    "This is really neat!":
        msg u "I'm glad you think so!" curly
        award heart u
    "Does it remember choices?":
        msg u "Yes it does!" bounce big curly
        award heart u
        msg u "And you can use regular Ren'Py code in the choices"
        msg u "to do things like award heart points and the like." curly
msg u "You can choose an answer any time while the choices are on-screen,"
msg u "Or just let the timer run out to stay silent, too."
```

Timed menus are written almost identically to regular Ren'Py menus, but you can add as many lines of dialogue before the first choice as you like. The choices included in the menu will be on-screen while the dialogue after the `timed`

`menu:` statement is shown.

The dialogue for the menu can be written using the `msg` CDS or the spreadsheet format. You can also include other regular scripting lines, such as characters entering/exiting chatrooms, inviting guests, or awarding heart points. You can include conditional statements inside the menu dialogue or on choices themselves e.g.

```
ju "[name], do you own a cat?"
menu:
    "No.":
        $ owns_cat = False
    "I do.":
        $ owns_cat = True
ju "I see. Cats are wonderful creatures, aren't they?"
timed menu:
    ju "Elizabeth the 3rd has been a constant source of joy in my life"
    if owns_cat:
        ju "as I'm sure your cat is in yours."
    ju "It's a shame some may never know the joy of owning a cat."
    "Some people are allergic though, like Zen.":
        ju "That is true."
        ju "I believe that can be overcome with appropriate medication."
    "I don't know what I'd do without my cat" if owns_cat:
        ju "May the two of you never have to be parted, then."
    "I think dogs are better, though.":
        ju "Hmm. I don't agree but I will respect your opinion."
```

For this menu, the line "as I'm sure your cat is in yours" only appears to a player who previously answered that they own a cat. Similarly, a player who didn't say they own a cat will not see the choice "I don't know what I'd do without my cat".

Tip: Since the choices for a timed menu are smaller, it's often a good idea to either paraphrase the choices and/or keep the amount of text for each choice caption short. A maximum of three choices can appear on the screen at once for a timed menu.

Warning: While you can use Python statements inside timed menus, such as `$ owns_cat = True`, you **should not change the value of variables that are used in the menu**, such as in conditionals. This will cause undefined behaviour. So, the following is **incorrect**:

```
$ owns_cat = False
timed menu:
    ju "I believe you mentioned you own a cat, [name]?"
    $ owns_cat = True
    ju "They are wonderful companions."
    "Yeah I do own a cat" if owns_cat:
        ju "I see."
    "I don't have a cat" if not owns_cat:
        ju "Oh, I was mistaken."
```

Because `owns_cat` was set to True inside the menu itself, its initial value before the menu (False) is used and it will be impossible for the player to ever see the "Yeah I do own a cat" choice, even if they see all the timed menu dialogue.

You can, however, set variables inside the menu provided they are not used in that same menu. So, the following is acceptable:

```
# These variables are set up outside the menu in the case that the player
# doesn't see the full menu
```

```
$ owns_cat = False
$ answered_jumin = False
timed menu
    ju "Do you own a cat, [name]?"
    ju "Elizabeth the 3rd has been a constant source of joy in my life"
    ju "It's a shame some may never know the joy of owning a cat."
    $ answered_jumin = False
    "I have a cat.":
        $ owns_cat = True
        $ answered_jumin = True
        ju "You're someone of excellent taste, I see."
    "I don't have a cat.":
        $ owns_cat = False
        $ answered_jumin = True
        ju "Ah, that is most unfortunate then."
if not answered_jumin:
    # The player didn't reply to the timed menu
    ju "I apologize if that was an overly personal question."
```

7.6.1 Timed Menu Settings

There are two options in the settings that affect timed menus. Both are found under the Preferences tab. The first is **Timed Menu Speed**, a slider at the top of the screen. Clicking the title will cause the timed menu speed to be set to your current chatroom speed. Moving the slider further to the left will slow timed menus down (giving the player more time to reply), and moving the slider further to the right will speed timed menus up.

This creates a sort of "bullet time" for timed menus, where the regular chat speed may be very fast, but the timed menus will slow down to give the player time to read the messages and possible responses before deciding what to do.

The second option is under **Accessibility Options** and allows the player to toggle timed menus off altogether. If timed menus are turned off, the menu will act like a regular menu. All of the dialogue before the choices will be shown, and then the player will be presented with the "answer" button at the bottom of the screen. They will be able to choose between any of the given answers, or use a special choice that will be shown as "(Say nothing)" and will function as though the timer on the menu ran out without the player responding.

Keep these options in mind when using timed menus in your script, as not all players will want to keep them turned on.

7.6.2 Timed Menu Functionality

There are a few special things to note about timed menus:

- The time the player has to answer the timed menu depends on what the **Timed Menu Speed** slider in the settings is set to. By default this is set to the equivalent of SPEED 5 in the chatrooms.
- The timed menu's timer will expire after all the dialogue inside the menu before the choices has been posted. How long it takes to post all the messages depends on the **Timed Menu Speed** as mentioned above. So, regardless of the player's chat speed or timed menu speed, they will always have enough time to view all the dialogue inside a menu before choosing an answer.
- If a player is on Max Speed, the timed menu will be skipped entirely. The dialogue will be posted, but there will be no choices at the bottom of the screen and the program will act as though the player did not reply.
  - However, if the player has timed menus turned off, the answer button will still appear at the bottom of the screen and allow the player to choose an answer (or choose to remain silent)

- If a player chooses an answer before all the menu dialogue has been posted, they will not be able to see the rest of the menu dialogue.
- If a player replays a chatroom with timed menus from the History screen, it will always act as though the Timed Menus option is turned **off**; that is, the chat will always stop at the end of the timed menu dialogue and show the answer button to allow the player to pick a response they have previously seen in-game.
  - If the player has never chosen any of the menu choices in a past playthrough, the answer button and choices will not appear at all (since the only option would be to remain silent). The menu dialogue will be posted and the chat will simply move on.
- If all the choices in the menu have a conditional dictating when they should be shown (e.g. `"I want to visit Seven" if s.heart_points > 10:`) and none of those conditions evaluate to True (that is, none of the choices should be shown to the player because they don't meet the conditions), then the menu dialogue will be shown only if `show_empty_menus` is True. This variable can be found in `variables_editable.rpy` and is True by default. If set to False, if a menu has no valid choices then the menu dialogue will be skipped altogether and the chat will continue after the menu.

7.7 Chatroom Creator

While scripting chatrooms yourself will allow for the greatest flexibility and allow you to create entire routes inside Mysterious Messenger, the program also offers the **Chatroom Creator** to visually put together chatrooms and watch them play out or export them to code.

To access the Chatroom Creator, head to **Developer** options on the main menu and select the **Chatroom Creator** button. You will then see a sub-menu to either create a new chat or load an existing one.

7.7.1 Creating a New Chat

Upon creating a new chat, you will be prompted to enter a title for the chatroom. This will be recorded on the save screen, and is also used to auto-generate the label and title of the chatroom when exporting it to code. You can change this later in the **Other** tab of the creator.

7.7.2 The Dialogue Tab

There are several tabs in the chatroom creator. The first one contains several features for adding basic chat messages.

You can select the person who sends the message by clicking the **Speaker** dropdown. This is also the person who will enter/exit the chatroom if you click the **Add Enter Msg** or **Add Exit Msg** buttons (e.g. if the speaker is 707, **Add Enter Msg** will add the message **707 has entered the chatroom** to the chat).

Next there are several buttons to toggle **Bold**, *Italics*, Underline, decrease/increase or reset the text size, and an additional dropdown for **Fonts**. You will be able to see these changes reflected in the text inside the text box.

To add a message to the chat, choose your speaker and add any font options, then click the text box to enter your dialogue. You can either hit **Enter** on your keyboard or **Add to Chat** in the bottom right corner to add the dialogue to the chat.

If you make a mistake, the **Undo** button at the bottom of the **Dialogue** tab will allow you to undo up to 5 messages. **Redo** will redo messages you previously undid.

Lastly, **Clear Chat** will remove all messages from the chatlog. Use this only if you want to completely start over.

Undo & Redo

On the **Dialogue** tab beneath the text input box, there are also buttons to **Clear Chat**, **Undo**, and **Redo**. Note that undo & redo only go back **5 changes** in the past - so save often to be safe!

Clear Chat will ask for confirmation before clearing all of the messages from the chatlog to start completely from scratch. Note that while this can be undone, it's a good idea to be really sure you want to clear the chat before clicking this option!

7.7.3 The Effects Tab

Emojis

To add emojis to the chat, use the **Add Emoji** button under the **Effects** tab. This will initially bring up all of the emojis available to the current speaker. You can select one of these emojis and click **Confirm** to add it to the chat, or click the **X** in the top-right corner to cancel.

At the top of the Emojis popup are two dropdowns: **Emojis** and **Speaker**.

- **Emojis** changes the set of emojis shown which can be selected to add to the chat
- **Speaker** changes the character who will send the message

Note that the **Speaker** and **Emojis** do not have to be the same person. If you select a different person's **Emojis** to be shown, the speaker will be updated to that person. However, you can then update the **Speaker** to someone else.

For example, if you wanted 707 to send V's emojis, you would first set **Emojis** to **V** to see V's emojis. Then you would change **Speaker** to **707**, select one of V's emojis, and click **Confirm**.

Special Bubbles

To use special bubbles in the background of a message, use the **Special Bubbles** button under the **Effects** tab.

This will bring up a popup showing the available bubbles to use. At the top of the Special Bubbles popup there are two dropdowns: **Bubbles** and **Speaker**.

- **Bubbles** changes the set of special bubbles that can be selected to add to the chat
- **Speaker** changes the character who will send the message

As with emojis, the **Speaker** and **Bubbles** do not have to be the same person. If you select a different Speaker, the bubbles will be updated to that person. However, you can then update the **Bubbles** to someone else.

For example, if you wanted 707 to send Jumin's cat bubble, you would select **707** as the speaker, then select **Jumin Han** under the **Bubbles** dropdown and select the appropriate bubble.

When you click a bubble, in most cases you will be given a small popup letting you choose the size of the bubble. After selecting this, you can hit **Confirm** and the current dialogue in the dialogue box will be posted to the chat with the selected background.

Screen Shake

To make the screen shake, on the **Effects** tab, you can click **Add Shake**. This will display a message in the chatlog to indicate where the screen shake will occur.

Animations

To add animations such as the scrolling hack effects or secure chatroom animation, under the **Effects** tab click **Add Animations** and select the animation you would like to add.

For the scrolling hack effects, you will also have the option to *reverse* the animation by checking the **Reverse Animation** box before clicking **Confirm**. A message will appear in the chatlog indicating where this animation will play.

7.7.4 The Other Tab

Select Background

To change the chat background, go to the **Other** tab and click **Select Background**. You'll also have the option to include the cracked screen overlay by checking that option at the bottom. Note that this applies to all backgrounds for the duration of this chatroom.

By default this background will appear after the last message added to the chatlog, but you can force it to be set up at the start of the chatroom by checking **Add to the beginning of the chat**. A message will be displayed in the chatlog to indicate where the background will be changed.

Add Music

To add music to the chat, on the **Other** tab select the button **Add Music**. Here you'll see a list of available songs to add to the game.

Select a song and click **Confirm** to add it to the chat. A message will appear indicating when the song will begin playing. You can also force the song to start at the beginning of the chat by checking **Add to the beginning of the chat** above the **Confirm** button.

Add Participants

To select who begins in the chatroom, on the **Other** tab, select the button **Add Participants**. Here you'll see a list of all characters currently in the game. You can check off who begins in the chatroom and hit **Confirm**.

Note that this only dictates whose name appears as in the chatroom at the top of the screen during a chatroom. If you use the enter/exit messages, characters will still appear/disappear from that list accordingly.

Play Chat

Clicking **Play Chat** on the **Other** tab will let you play out the chatroom as if it had been in-game. Any messages like `Play Music: Mint Eye` will play as they should in this format (as opposed to it appearing as text). You can use the toggles in the settings at the top-right of the chat creator to turn on/off things like animated backgrounds and the hacking effects as well, which will affect how the chat looks when it's played out.

Generate Code

Clicking **Generate Code** on the **Other** tab will generate a `.rpy` file inside the `game/generated/` folder with the chatroom formatted into code you can put into the program. There are several features the program supports which the chatroom creator does not, such as providing choice menus or jumping to story mode in the middle of a chatroom. If you'd like to use these features, you can use the chatroom creator to build the base of your chatroom before exporting it to code to add these additional features.

Note that the name of the file and associated labels will take the name of the chatroom, as provided when you first made the chat. If you click **Generate Code** when a file with the same name exists in the `generated` folder, you will be prompted if you would like to **overwrite** this file. Keep this in mind if you've made changes to the code outside of the chatroom creator, as that will not be reflected in the chatroom creator.

You can rename the chatroom using the **Rename Chat** button on the **Other** tab.

The generated code will include a label for your chatroom, an `after_` label, optionally for adding things like text messages etc, and a `ChatRoom` definition for your chatroom that you can use as part of a route definition. Note that you will need to fill in the trigger time yourself; it defaults to `00:00` in the generated code.

Saving & Loading Chats

On the **Other** tab, there are also buttons to **Save** and **Load**. These buttons lead to a different save screen than the main game. You can save chatrooms here and load them later to edit, play, or export to code. They will be saved with the name of the chatroom, which can also be changed on the **Other** tab.

7.7.5 Editing Messages

In the chatroom creator, you can also edit messages by clicking on them in the preview window. Most messages will have the following options:

Remove message Remove this message from the chatlog.

Insert message before Creates an overlay above the selected message. Any new messages will then be added at that particular location in the chatlog. Click the overlay to remove it, and new messages will be added to the bottom of the chat as usual.

Edit text Update the text and/or speaker of the selected message. You can change the font, size, and the bold/italics/underine of the message.

Change bubble Change the bubble used for this message.

Change speaker Change the speaker of this message.

Change profile picture Update the profile picture of the current speaker.

Note that when editing an emoji, you can only choose another emoji to swap it with. If you'd like to add text instead of an emoji, you can use the message dropdown to remove the emoji, and then click on the line below the one where you'd like to add the new message and click **Insert message before**. Then type your new message and add it to the chatlog.

CHAPTER

EIGHT

PHONE CALLS

8.1 Creating a Regular Phone Call

Note: Example files to look at:

- tutorial_5_coffee.rpy
- tutorial_11_story_call.rpy

A brief overview of the steps required (more detail below):

1. Create a label + the correct suffix for the phone call.
    1. `my_label_outgoing_ja` to make an outgoing call to the character `ja` available.
    2. `my_label_incoming_ja` to trigger an incoming call from the character `ja` after the corresponding timeline item has been played.
    3. `my_label_story_call_ja` to create a required Story Call after this timeline item from the character `ja`.
2. Write the phone call dialogue.
3. End the phone call with `return`

Phone calls that are tied to a particular timeline item follow a specific naming convention so the program can find them and make them available to the player. If you have a timeline item (i.e. a chatroom, Story Mode section, or Story Call) titled

```
label day_1_4:
```

then an **incoming** call should be named:

```
label day_1_4_incoming_z:
```

an **outgoing** call should be named:

```
label day_1_4_outgoing_z:
```

and an attached Story Call should be named:

```
label day_1_4_story_call_z:
```

where `z` is the variable of the character whom the player is calling or who is calling the player (see *Creating Characters* for a list of the existing characters).

The following section will cover **incoming** and **outgoing** calls to characters that are optional for continuing with the story. You can find more information on Story Calls at *Story Calls*.

8.1.1 Incoming Calls

Once the player is done playing through a timeline item, the program will automatically make any calls associated with that timeline item (through the naming convention mentioned above) available. If the program finds an **incoming call** label, then when the player finishes the timeline item they will receive an incoming call from the associated character. They have ten seconds to pick the phone call up, or they can reject the call and phone the character back later.

Note: There can only be one incoming phone call after a given timeline item. However, it is possible to have both an incoming call after a chatroom as well as a different incoming call after an attached Story Mode.

If the player rejects the incoming call or lets the timer run out without answering it, it acts like an **outgoing** call and the player can phone the character back. The player can typically play one additional timeline item before the phone call "expires" and they cannot call the character back to receive that conversation.

8.1.2 Outgoing Calls

Once a player finishes playing through an item that has associated outgoing calls (linked through the naming conventions mentioned above), all the associated outgoing calls will be made available. The player must then click on the characters in their Contacts list or use the redial button next to a past phone call on the History tab to call the character.

An outgoing call is typically available immediately after its associated timeline item and for one additional timeline item following when it became available, after which it will "expire" and the player will be unable to view that conversation by calling the character.

Note: There can be up to one outgoing call per character available after each timeline item. If a player misses an incoming call from a character who also has an outgoing call, both conversations will be added to the pool of available calls. The player can then call the character back multiple times to receive both conversations.

Note: For development, there is an option to show an indicator underneath a character's contact image when there is a call available for them. You can find it in the Developer options titled **Available call indicator**.

8.1.3 Missed Calls

If the timer on an incoming phone call runs out, the phone call will be marked as "missed" and moved to the pool of available calls. The player can then call the character back to receive that conversation.

Phone calls can also be missed if real-time mode is turned on from the Developer settings. If an incoming call is scheduled to occur after a chatroom at 10:20 and the player opens the game after a new chatroom became available at 13:30, then they will receive a notification of a missed call at 13:30 (just before the new chatroom). If not too much time has passed since the missed call, they can then phone the character back to receive that conversation.

Additionally, if the player is using the game but has not yet played the 10:20 chatroom by 13:30, then if the game is open at 13:30 they will receive an incoming call which they can either accept or reject. This acts the same way as if the incoming call had been received immediately after playing the chatroom.

You can also change the dialogue of a phone call depending on whether the player picked up when the character first called, or if they are calling the character back. For more on that, see *Phone Callbacks*.

8.1.4 Phone Call Expiry

As mentioned, both incoming and outgoing phone calls will become unavailable after a period of time has passed. In-game, this time is tracked via the availability of timeline items.

By default, a phone call will "expire" or become unavailable one additional timeline item after it first became available. In practice, this means if there are timeline items set up at the following trigger times:

- 10:20 - Chatroom
- 11:40 - Chatroom + Story Call (Yoosung)
- 13:52 - Solo Story Mode
- 16:00 - Chatroom + Story Mode

Say that the player can call Yoosung after the 10:20 AM chatroom. The "one timeline item" default timeout means that the player will be able to call Yoosung and get that particular outgoing call any time between 10:20 (after they've played the chatroom) and 13:51 (before they've played the 13:52 chatroom (sequential mode)/before it's become available (real-time)).

Note: Note also that the fact that there's a story call with Yoosung at 11:40 does not affect the availability of his outgoing phone call. Each outgoing/incoming phone call is treated independently of any other phone calls, so it's even possible for the character to have more than one outgoing phone call "queued" at the same time.

e.g. if there's an incoming call from Yoosung at 10:20 which the player misses, and then an outgoing call will become available after the 11:40 chatroom, then at 13:00 the player can call Yoosung once to receive the missed incoming call, and then call again to receive the outgoing call.

You can adjust the number of timeline items after which a phone call will expire through the `phone_timeout_dict` found in `variables_editable.rpy`. This dictionary takes the name of a phone call label and an integer (whole number) representing the number of timeline items after which this phone call should expire. The lowest valid number for this field is 0, which means that if the player doesn't play the phone call before the timeline item immediately after it becomes available, then it will expire. So, if a phone call with an expiry of 0 becomes available at 10:20 using the above timeline, then the player must call the character before 11:40 to see that call.

```
define phone_timeout_dict = dict(
    # TUTORIAL DAY
    tutorial_chat_incoming_x=2,
    tutorial_chat_outgoing_y=0,
    # Feel free to add more below
)
```

8.2 Creating a Story Call

Mysterious Messenger also includes "story calls", phone calls from the characters which the player must see before they can continue with the story. Story calls can be standalone, appearing on the timeline as their own story with a trigger time, or they can be attached to chatrooms or standalone story mode items as well and played after the main/parent timeline item is complete.

To create a standalone story call, see *Story Calls*. Otherwise, the program will automatically create a story call if a certain naming convention is used. If the label of the parent item (i.e. a chatroom or standalone story mode) is `newyear_4_2`, then you can create a label with the suffix `_story_call_` + the file_id of the character who will be calling the player e.g.

```
label newyear_4_2_story_call_ja:
    # This is a label for a story call from the character ja
```

Unlike story mode labels, there is no "generic" story call; all story calls must come from a particular character who will be calling the player.

Tip: If you want to hide the identity of the caller, you can create a special "Anonymous" ChatCharacter whose file_id you can use for the story call label.

Story calls, like chatrooms, can also expire if the player is playing in real-time or if they hang up in the middle of the call. Expired story calls are treated as though the caller left the player a voicemail. As with chatrooms, the expired version of the story call is found at the name of the original story call label + `_expired`:

```
label newyear_4_2_story_call_ja_expired:
    # The expired version of the phone call
```

A player who misses a story call or hangs up in the middle of it will either be able to play the expired version to continue the story, or can "buy back" the story call to experience the participated version.

Story call dialogue is written the same way as regular phone calls.

8.3 Writing a Phone Call

Phone call dialogue uses the regular character variables and doesn't have any special arguments such as `(img=True)` or special fonts. Just use the character variable and write out dialogue in quotations e.g.

```
label day_1_4_incoming_z:
    z "Hi, [name]! This is phone call dialogue."
    z "You write it like this."
    return
```

Phone call labels end with `return` like all other labels.

You may also find Ren'Py's "monologue mode" helpful, since you won't be switching between expressions or different speakers very often. See `tutorial_5_coffee.rpy` for an example of this.

8.4 Providing Choices

Menus for phone calls are written very similarly to menus in other parts of the game:

```
menu:
    extend ''
    "I love cats.":
        ju "I do as well."
    "I like dogs better.":
        ju "Hmm. I will respectfully disagree with your opinion."
```

The only notable difference is that you should include the line `extend ''` right after the `menu:` statement and before the first choice. This will keep the dialogue said just before the menu on-screen while the choices are visible.

This menu, like others, also takes a `paraphrased` argument for either the menu as a whole or for an individual choice. For more information, see *Paraphrased Choices*.

Note also that if you are writing out the main character's dialogue directly after a choice, the first line **does not** require a `(pauseVal=0)` clause.

8.5 Changing a Character's Voicemail

If the program determines there are no phone calls available for a character when the player phones them, it will automatically play the character's voicemail instead. You can change the character's voicemail whenever you like during a route.

To update a character's voicemail, use:

```
$ ja.voicemail = "voicemail_1"
```

where `ja` is the variable of the character whose voicemail you're changing, and `voicemail_1` is the name of the label where the voicemail can be found. The default voicemail, "voicemail_1", can be found in `01_mysme_engine/phonecall_system.rpy`.

Voicemails are written the same way as regular phone calls, though there are no restrictions on what a voicemail label can be called (but it's recommended you pick a descriptive name for it, probably including the word "voicemail").

You can use a character to say voicemail dialogue, or you can use the "generic" voicemail character, `vmail_phone` to say dialogue e.g.

```
label voicemail_1():
    voice "voice files/voicemail_1.mp3"
    vmail_phone "The person you have called is unavailable right now. Please leave a␣
↪message at the tone or try again."
    return
```

As shown above, you can also add voice files to the voicemail, which will be played during the next line of dialogue.

8.6 Phone Callbacks

8.6.1 Missed Calls

Tip: You can see an example of a missed incoming call callback if you miss Seven's call after the 'Inviting Guests' chatroom at 6:11am and call him back before 9:30am (in real-time) or by letting the call time out/hanging up before answering and then phoning him back (in sequential mode). The code is near the top of `tutorial_2_emails.rpy` in the `tutorial_day_scripts` folder.

Sometimes the player will miss a call from a character, either from playing in real-time or by ignoring the call when it comes up. By default, when the player calls the character back, if that call is available then the player will receive the exact same conversation they would have had with the character if they had picked up the phone when they initially called.

However, you can also alter the dialogue to account for the player calling the character back. The program will look for this alternate dialogue under a special label with the suffix `_incoming_z_callback` where `z` is the file_id of the character whom the player is phoning back. More specifically, if you had the following chatroom and incoming call:

```
label day_7_chat_4():

    scene morning
    s "This is some dialogue for the chatroom!"
    return

label day_7_chat_4_incoming_s():
    s "Hello [name]!"
    s "I'm glad you picked up~"
    # ...
    return
```

Then you can create the "callback" label for `s`'s incoming call at:

```
label day_7_chat_4_incoming_s_callback():
    s "[name]!!! You called me back, haha."
    s "I thought maybe you were busy so I didn't wanna bother you."
    s "But I'm glad you called."
    # ...
    return
```

If this `callback` label is present, the program will jump to it when the player is calling the character back instead of going to the regular `incoming_s` label. In replay, if the player has seen both the callback version of the call as well as the regular version, they will be prompted with a menu that allows them to pick whether they want to replay the regular version or the callback version.

If you'll be reusing dialogue between the two phone calls, it might be useful to create a second label that both calls can jump to. For example:

```
label day_7_chat_4():
    scene morning
    s "This is some dialogue for the chatroom!"
    return

label day_7_chat_4_incoming_s():
```

(continues on next page)

(continued from previous page)

```
    s "Hello [name]!"
    s "I thought maybe you'd be having lunch and wouldn't pick up."
    jump day_7_chat_4_incoming_s_ending

label day_7_chat_4_incoming_s_callback():
    s "[name]!!! You called me back, haha."
    s "I thought maybe you were busy so I didn't wanna bother you."
    s "But I'm glad you called."
    jump day_7_chat_4_incoming_s_ending

label day_7_chat_4_incoming_s_ending():
    s "Are you busy right now?"
    menu:
        extend ''
        "No, not particularly.":
            s "Great! So I had this idea for a cat cafe I wanted to bring up in the␣
↪chatroom."
            s "Don't tell Jaehee though, okay? She might get mad."
        "Yeah, I should probably get something done today.":
            s "Aw... don't be like that..."
            s "But, if you really have to leave, I'll understand."
            s "Just call me back when you're free!"
    s "Oh... looks like I actually have to go already. I'm sorry for ditching you!"
    s "Let's talk again later in the chatroom. Toodles!"
    return
```

The two calls start differently, but both jump to the `day_7_chat_4_incoming_s_ending` label to finish off the phone call. This can help keep your code more organized so you're not repeating dialogue in two separate labels.

8.6.2 Hanging Up

You can also define a callback for when the player hangs up in the middle of a phone call. First, in `variables_editable.rpy` scroll down to the header **PHONE HANG UP CALLBACK** and find the variable `default phone_hangup_callback`. By default, it is set to the example function `hang_up_callback_fn`, but you can change this to whatever you like or even change it in the middle of a route.

`phone_hangup_callback` should be given the name of a function which takes one parameter, the call that the player was in the middle of when they hung up the phone. The call is stored as a PhoneCall object, which has several useful fields you can access for more information on the call such as the caller and the label it's attached to. The most relevant fields are:

caller The ChatCharacter object of the person the player was on the phone with.

phone_label A string with the name of the label where this phone call is found.

call_status A string that is one of "incoming", "outgoing", "missed", or "voicemail". Details the status of the phone call.

voicemail True if this phone call is a character's voicemail message rather than a proper conversation. Typically if you check this variable, it's to ignore it.

You can use as many of these fields to check for details on the phone call as you like, then use that information to narrow down what the program should do. Like the profile picture callback, the function should return the name of a label to jump to. For example, the character might try phoning the player back, or you might lower the character's affection for the player. An example might look like:

```
init python:
    def hang_up_callback(phonecall):

        if phonecall.phone_label == 'day_7_chatroom_10_incoming_s':
            return 'day_7_chatroom_10_s_hangup'
        return

label day_7_chatroom_10_s_hangup():
    compose text s:
        s "[name], is your phone's connection okay?"
        s "The call just dropped T_T"
        label day_7_chatroom_10_s_hangup_menu1
    return
```

This can be as elaborate as you want it to be, including setting conditions that allow the player to pass plot branches or acknowledging that the player hung up at a certain point. A more complex example might be:

```
default knows_about_bomb = False
label day_7_chatroom_10_incoming_s():
    s "[name]... I know this might be hard to hear, but I should tell you now."
    s "I understand if you don't want to be a part of the RFA after this."
    $ knows_about_bomb = True
    s "There is a bomb in the apartment. It was supposed to protect sensitive␣
↪information."
    # ...
    return

# Presuming the same callback function is used as shown above
label day_7_chatroom_10_s_hangup():
    if knows_about_bomb:
        compose text s:
            s "Hey... I'm sure that was a lot of information so I get why you hung up.
↪"
            s "You can call me back whenever you're ready to talk."
        $ create_outgoing_call('day_7_chatroom_10_about_bomb', s)
    else:
        compose text s:
            s "[name], I didn't even get to tell you... you must be really upset."
            s "I think this is only going to make you more upset, but..."
            s "..."
            s "Call me when you get a moment, okay?"
        $ create_outgoing_call('day_7_chatroom_10_re_explain_issue', s)
    return
```

In this way, you can create context-sensitive callbacks so that the characters remember what they talked about with the player and can react to their actions in meaningful ways.

8.7 Phone-Only Characters

Like with Story Mode, sometimes you may have a character who only needs to exist for a phone call but doesn't need all the bells and whistles of a full ChatCharacter definition. There are two ways to define such characters:

8.7.1 Dialogue-Only Phone Characters

For characters that only need to speak in phone calls, but don't need a profile picture or any other information, you can create a regular Character and inherit from the `phone_char` object. This applies in situations such as:

- The player can hear the voice of someone else in the background of a call (e.g. the player called Yoosung, but can hear a professor talking in the background on occasion)
- The player called a regular character, but the person who picked up was not the regular character (e.g. the player phoned Jumin, but his father picked up instead)
- You are defining a specific voicemail voice

These characters don't need their own profile picture, but should have their own Character object to speak dialogue with so that the player can properly use the voice tagging system to mute characters they don't want to listen to.

To define a dialogue-only phone character, you can do:

```
define hana_phone = Character("Hana", kind=phone_character)
```

You can then write dialogue in-game via:

```
hana_phone "This is some dialogue said by Hana."
hana_phone "It uses the other_voice tag."
```

8.7.2 Incoming Call Phone Characters

For characters who can call the player, but don't need a fully-fledged character definition, there's the special `PhoneCharacter` class you can use to define a character for phone calls.

PhoneCharacters only need a name, a profile picture, and a file_id. Like with regular characters, the file_id is used to associate them with phone calls, so you can write incoming call labels that are automatically detected by the program. It also helps the program find the phone call to fill out the History.

A typical PhoneCharacter definition might look as follows:

```
default hana_phone  = PhoneCharacter("Hana", "Profile Pics/Other/hana-1.webp", 'ha')
```

You can put this definition anywhere you like, though it's usually a good idea to keep your character definitions in a central place where you can find them later.

Then, to write a phone call for this character, you can add a label like you would with the regular cast of characters:

```
label day_3_chat_7():
    scene evening
    y "Hi [name], sorry to bother you with this!"
    y "I'm about to write a test so I gave your number to a friend."
    y "She'll call you if anything changes."
    y "Wish me luck!"
    exit chatroom y
```

(continues on next page)

(continued from previous page)

```
label day_3_chat_7_incoming_ha():
    hana_phone "Hi, this is... um, [name], I think? Yoosung gave me your number."
    hana_phone "Nothing's wrong, I just wanted to check if the number worked."
    hana_phone "Haha, this is actually pretty awkward, so I think I'll hang up now."
    hana_phone "Bye."
    return
```

A `PhoneCharacter` acts the same as a `ChatCharacter` in nearly all respects, including the ability to update their profile picture, name, and more, but they won't appear in the player's contact list or on the home screen as a character profile. If you want that functionality, you should define them as a ChatCharacter instead (see *Creating Characters*).

Additionally, if the player misses a call from a `PhoneCharacter`, they will only be able to call the character back if that call is still available. In practice, this means that the player can't call a `PhoneCharacter` whenever they like; only when an outstanding call is available. Outgoing calls to `PhoneCharacter` are technically supported if an outgoing call is available; however, if you wish to allow for outgoing calls, you should define this character using the `ChatCharacter` class instead of `PhoneCharacter`, as phone-only characters do not have a contact image which makes it difficult for players to call them.

CHAPTER

NINE

TEXT MESSAGES

There are two kinds of text messaging styles available in this program. The first, referred to as "regular text messages", delivers text messages in "chunks" which the player can reply to whenever they like. The second style, called "real-time text conversations", plays out similar to a one-on-one chatroom. The player receives an initial text message, and upon opening the text conversation, cannot leave until the conversation is over or the player chooses to end it themselves.

9.1 The compose text CDS

The main way of writing text messages is with a special `compose text` CDS that looks like the following example:

```
compose text s real_time:
    s "Hi, [name]."
    s "I wanted to show you how text messages work."
    s "Send a reply if you got this message okay!"
    label example_reply_1
```

The `compose text` CDS takes several options, which are explained below.

who Required. "who" should be the ChatCharacter variable of the character who is taking part in the conversation. e.g. `s`

real_time Indicates that this text message should play out as a real-time text conversation. If this option is not present, the conversation is treated as a regular text message.

deliver_at Takes one of four arguments:

- `random` to have the message delivered at a random time after the associated timeline item has been played but before the next one is made available. This is only applicable if the game is being played in real-time mode; otherwise, the message is simply delivered after the timeline item is played.
- `next_item` to have the message delivered at a random time between the timeline item after this one and the item following it. e.g. if there are three items at 11:00, 12:00, and 13:00 and this text is in the `after_` label of the 11:00 item, `deliver_at next_item` will deliver this message sometime between 12:00 and 13:00, while `deliver_at random` would deliver the text message sometime between 11:00 and 12:00. This only works when the player is playing in real-time mode; otherwise, the message is simply delivered after its timeline item has been played.
- `00:00` where `00:00` is any time formatted in 24-hour format. For example, `14:23` would cause the message to be delivered at `14:23` in real-time. This only works when the player is playing in real-time mode; otherwise, the message is delivered after the associated timeline item is played.
- `now` to show the player a notification for this text message immediately. **This is intended for use during chatrooms, phone calls, and Story Mode**. The message itself will only be sent after the player completes

the story item. If they hang up the call or back out of the chatroom and cause it to expire, the text message will not appear in their inbox. See *Manually Sending Text Messages* for more information.

The dialogue inside the `compose text` block also takes options besides dialogue:

pause <#> Indicates that the delivery of the item after the pause should wait the given number of seconds before being delivered. This only occurs when the player is playing on real-time mode. `pause` also takes math expressions for the time e.g. `pause 2*60` would cause the following message to be delivered two minutes (120 seconds) after the previous message.

label <labelname> Sets the label the program should jump to in order to continue a conversation or allow the player to reply. `labelname` should not be in quotations. Including a label is optional; excluding it will simply not provide the player with the option to answer the text message.

Text messages can also include conditional statements to vary dialogue based on certain conditions. An example text message with several of these options combined might look like:

```
label after_day_1_4:
    compose text ja real_time deliver_at 10:15:
        ja "Hello, [name]."
        pause 60
        ja "I had a moment of time on my break so I thought I would message you."
        if ja.heart_points > 5:
            ja "I would have liked to call and hear your voice, but time is short."
        label day_1_4_ja_msg

    return

label day_1_4_ja_msg:
    ja "What are you doing right now?"
    menu:
        "Not too much. I'm glad you messaged.":
            ja "I see. I had a question for you, in fact."
        "Perhaps you can call me later?" if ja.heart_points > 5:
            ja "I would like that very much ^^"
            award heart ja
            ja "If you don't mind, I wanted to ask you:"
        "I'm actually kind of busy at the moment.":
            ja "Oh, I understand."
            ja "I'm sorry to have caught you at a bad time."
            ja "Before you go, however, I was hoping to ask:"
    ja "Do you like coffee?"
    menu:
        "I love coffee! I drink it every morning.":
            ja "You do? I think it can be nice to have a morning routine, certainly."
        "Only if I can have it with lots of cream and sugar.":
            ja "Ah, that's understandable."
            ja "Coffee can be rather bitter on its own."
        "I drink it straight black.":
            ja "That's very bold of you. Some coffees are rather bitter."
        "I don't really like coffee.":
            ja "I understand. It's not always to everyone's taste."
    ja "I'm afraid my break is over, but I'm glad I got to talk with you."
    ja "I hope we can speak again soon."

    return
```

In this case, the last message of the `compose text` block will be the one delivered at the requested time (10:15), so the first message (Hello, [name]) is delivered around 10:14 and the final message at 10:15. The first message may be

delivered slightly earlier depending on whether the player has more than 5 heart points with `ja` or not, as the program also calculates roughly how long it would take to type out each message and staggers delivery time accordingly. `pause` allows you to fine-tune this timing.

9.2 Regular Text Messages

Note: Example files to look at:

- tutorial_3_text_message.rpy
- tutorial_3b_VN.rpy
- tutorial_5_coffee.rpy
- tutorial_11_story_call.rpy

A brief overview of the steps required (more detail below):

1. Create a label using the prefix `after_` + the name of the timeline item you want to send the text messages after e.g. `label after_day_1_4`
2. **Write `compose text z:` where `z` is the variable for the character who is sending the message.**
    1. Optionally, you can also include arguments to tell the program when to deliver the message. Acceptable options are `deliver_at 00:00` where `00:00` is the time to deliver the message at, in 24-hour time; `deliver_at random` to deliver the message randomly before the next timeline item, or `deliver_at next_item` to deliver the message sometime between the next timeline item being available and the one after that.
3. **Write dialogue for the conversation, indented at least one level to the right underneath the `compose_text` line. You may**
    1. You can also include the `pause` statement with a number of seconds to pause for, e.g. `pause 2.0*60`. The message delivery will be "staggered", with the message after the pause waiting the given number of seconds before sending.
4. (Optional) Include a label at the end which will be jumped to to continue the conversation e.g. `label menu1`
5. End the `after_` label with a return.
6. (Optional) Create the label the player can jump to to continue the conversation.
    1. (Optional) Allow the player to reply to the text message again by writing `$ s.text_label = "menu2"` where `s` is the variable for the character sending the message and `"menu2"` is the name of the label to jump to to continue the conversation.
7. Finish the reply label with `return`

9.2.1 Sending a Text Message

To have a character text the player after a chatroom, the program looks for a label with a special naming convention. For example, if your chatroom is called

```
label my_chatroom:
```

then you should create a label called

```
label after_my_chatroom:
```

Next, inside the `after_` label, use the special `compose text` CDS:

```
compose text s:
```

where `s` is the variable for the character who is sending the message. You can then write dialogue the same way as you would for a chatroom, including adding CGs, emojis, and changing fonts. You can use either the spreadsheet style or the `msg` CDS (see *Writing Chatroom Dialogue*).

Warning: Text messages currently do not support special speech bubbles.

Note that all dialogue for a text message should be indented one level to the right so as to be underneath the `compose text` statement:

```
compose text s:
    msg s "This is an example text message!" curly
    msg s "You can write lots of dialogue here."
    msg s "Just make sure it's indented at the correct level~"
```

Finally, if you would like the player to be able to reply, include the name of a label the program should jump to in order to find that reply:

```
compose text s:
    msg s "This is an example text message!" curly
    msg s "You can write lots of dialogue here."
    msg s "Just make sure it's indented at the correct level~"
    label my_reply_1
```

where `my_reply_1` is the name of the label the program should jump to in order to continue this conversation when the player opens the message. If you don't want the player to be able to reply, then you can simply omit this line. They will be able to read the text messages but won't be able to respond.

Finally, end the `after_` label with the line `return`.

```
label after_my_chatroom:

    compose text z:
        z "Hi [name]!"
        z "Hope your morning has been good so far."
        z "Did you eat breakfast yet?"
        label my_reply_1

    return
```

9.2.2 Replying to a Text Message

If you gave `compose text` a label, when the player presses "Answer", the program will jump to that label to continue the conversation. There are no restrictions on what you can call this label, but it's recommended you come up with a consistent naming scheme to avoid accidentally creating several labels with the same name (which will cause errors).

Create a new reply for the label from your text message:

```
label my_reply_1:
    menu:
        "No, I haven't eaten yet.":
            z "That's no good! You should eat something soon."
        "Yup! I ate as soon as I woke up.":
            z "That's good to hear."
            award heart z
    z "Eating breakfast in the morning is really important."
    return
```

For regular text messages, the label should immediately begin with a `menu:` so that the player is presented with a choice of answers. You can add as many or as few options as you wish. The dialogue here is written the same way as chatrooms, including awarding heart points. You can only award one heart point per reply. It can be for a character not in the conversation as well. For more information on heart points, see *Showing a Heart Icon*.

Finally, end the whole label with `return`.

Replying after the first message

If you want the player to be able to reply again after the first message, you need to tell the program where it can find the label to continue the conversation with `$ z.text_label = "menu2"` where `z` is the variable of the character who is taking part in the conversation and `menu2` is the name of the label to jump to. A text message chain with two opportunities to reply might look like:

```
label after_my_chatroom:

    compose text z:
        z "Hi [name]!"
        z "Hope your morning has been good so far."
        z "Did you eat breakfast yet?"
        label my_reply_1

    return

label my_reply_1:
    menu:
        "No, I haven't eaten yet.":
            z "That's no good! You should eat something soon."
        "Yup! I ate as soon as I woke up.":
            z "That's good to hear."
    z "Eating breakfast in the morning is really important."
    z "Do you have any plans for the day?"
    $ z.text_label = "my_reply_2"
    return

label my_reply_2:
    menu:
        "Maybe you should call me~":
```

(continues on next page)

(continued from previous page)

```
            z "!!"
            z "{image=zen_wink}" (img=True)
            z "Haha, maybe I will~"
        "Nah, I'm going to be lazy today.":
            z "Sometimes it's nice to have days like that."
    z "It was good talking with you!"
    return
```

9.2.3 How Text Message Delivery Works

When you write text messages inside an `after_` label, those messages are added to a "delivery pool" which is delivered to the player in increments after they finish playing the associated timeline item. So, if you composed text messages for characters A, B, and C, the player may receive Character A's message immediately upon returning to the home screen, and will then receive Character B and C's messages either by waiting around on the home screen or performing other actions such as replying to other text message or emails.

Similarly, if the player has replied to Character A's message, Character A's response gets added to the pool and will eventually be delivered to the player after some time has passed.

All text messages in the delivery pool are immediately delivered as soon as the player clicks to enter the timeline screen.

In this program, there are no time limits on when you can or can't reply to text messages. However, if a character sends the player a new text message and the player hasn't replied to the previous conversation, they will no longer be able to continue the older conversation.

9.3 Real-Time Text Conversations

Unlike regular text messages, real-time text conversations play out similar to a chatroom with a single character. The player receives a text message, and upon entering the conversation, cannot leave until either the conversation is over or the player exits the conversation manually and the conversation is lost.

Note: Example files to look at:

- tutorial_3_text_message.rpy
- tutorial_3b_VN.rpy
- tutorial_5_coffee.rpy
- tutorial_11_story_call.rpy

A brief overview of the steps required (more detail below):

1. Create a label using the prefix `after_` + the name of the timeline item you want to send the text messages after e.g. `label after_day_1_4`
2. Write `compose text z real_time:` where `z` is the variable for the character who is sending the message.
    1. Optionally, you can also include arguments to tell the program when to deliver the message. Acceptable options are `deliver_at 00:00` where `00:00` is the time to deliver the message at, in 24-hour time; `deliver_at random` to deliver the message randomly before the next timeline item, or `deliver_at next_item` to deliver the message sometime between the next timeline item being available and the one after that.

3. Write dialogue for the conversation, indented at least one level to the right underneath the `compose_text` line. You may also include conditionals here.
    1. You can also include the `pause` statement with a number of seconds to pause for, e.g. `pause 2.0*60`. The message delivery will be "staggered", with the message after the pause waiting the given number of seconds before sending.
4. (Optional) Include a label at the end which will be jumped to to continue the conversation e.g. `label menu1`
5. End the `after_` label with a return.
6. (Optional) Create the label the player can jump to to continue the conversation.
    1. (Optional) Allow the player to reply to the text message again by writing `$ s.text_label = "menu2"` where `s` is the variable for the character sending the message and `"menu2"` is the name of the label to jump to to continue the conversation.
7. Finish the reply label with `return`

9.3.1 Writing a Text Conversation

To have a character initiate a text conversation, like with regular text messages, you need to have a label that follows a specific naming convention. For example, if your timeline item is called

```
label casual_route_2_4:
```

then you need to create a label called

```
label after_casual_route_2_4:
```

Then you will compose the text message. This uses a special `compose text` CDS:

```
compose text ju real_time:
    ju "I didn't understand what Luciel wrote in the chatroom earlier."
    ju "What exactly does 'yeet' mean?"
```

The main difference between this and regular text messages is the addition of the `real_time` option. This tells the program that this conversation will play out in real time once the player enters the conversation rather than having all the messages delivered at once.

Dialogue underneath the `compose text` statement can be written the same way as it is for chatrooms, and allows the use of both the spreadsheet style as well as the `msg` CDS. Text message conversations can also have CGs, emojis, and special fonts.

Warning: Text messages currently do not support special speech bubbles.

Any dialogue written under the `compose text` statement will show up as "backlog" before the user enters the conversation, so it is usually brief.

You can compose text messages for as many characters as you like, and mix and match regular text messages with real-time text conversations freely. There are also additional options available to you for scheduling when the first text message is delivered. See *The compose text CDS* for more.

9.3.2 Continuing a Text Message Conversation

To allow the player to continue the text message conversation, you must provide a label for the program to jump to when the player opens that text message. This is done with the `label` argument inside the `compose text` CDS:

```
compose text ju real_time:
    ju "I didn't understand what Luciel wrote in the chatroom earlier."
    ju "What exactly does 'yeet' mean?"
    label casual_2_4_ju_reply
```

The program will then jump to the label `casual_2_4_ju_reply` to continue the conversation. You will write dialogue inside the label identically to a chatroom:

```
label after_casual_route_2_4:
    compose text ju real_time:
        msg ju "I didn't understand what Luciel wrote in the chatroom earlier."
        msg ju "What exactly does 'yeet' mean?" ser1
        label casual_2_4_ju_reply

    return

label casual_2_4_ju_reply:
    msg ju "I tried to search the Urban Dictionary, but I didn't understand the␣
↪results." ser1
    menu:
        "You say it when you throw something.":
            ju "When you throw something?"
            ju "Hmm. I don't know if I really understand."
            ju "But thank you for attempting to explain, nevertheless."
            award heart ju
        "lololol you're so funny Jumin.":
            ju "It was not my intention to be a source of entertainment."
            ju "But I suppose I am glad if I was able to bring some levity to your␣
↪day."
    msg ju "Perhaps I will ask Assistant Kang for further clarification." ser1
    return
```

You can add additional dialogue after the reply label, and as many or as few menus (for choices) as you like. Unlike with regular text messages, because the conversation occurs in real time, you may also award or take away heart points as many times as you like during the conversation using `award heart` and `break heart`. The whole label should end with `return`.

9.3.3 Leaving a Real-Time Text Conversation

Since real-time text conversations function more closely to chatrooms, if the player clicks the "Back" button during a real-time text conversation, they will be asked if they'd like to end the conversation.

Backing out of a real-time text conversation means that conversation will no longer be available to continue, though the player will retain any heart points or CGs they unlocked before they exited the conversation. There is no option to "replay" text message conversations, but unlike chatrooms a text message conversation does not expire until another text message overwrites it.

9.4 Text Message Backlog

If you would like text messages to appear as "backlog" in the player's text message inbox to serve as backstory or to set up a route, there is a special CDS to make writing this easier. You can set the text message timestamps to be several real-life days in the past, and even specify what time a message should appear to be sent at:

```
add backlog s -4 time 23:22:
    s "do u think Jumin would hire me as his assistant?"
    msg s "Maybe he'd even let me take care of Elly for him ^^" curly
    if s.heart_points < 5 or ju.heart_points > 5:
        m "Not really likely, no"
        s "Aw T_T"
        s "I would be such a good worker!!"
    else:
        m "You'd be such a good cat mom!!"
        s "Right???"
        s "Maybe I should ask to be hired as his cat nanny."
        s "He needs a cat nanny, doesn't he?"
```

`add backlog` takes the following options:

who The character whose text message this backlog is being added to. Should be a ChatCharacter object.

e.g. s

day Optional. An integer representing the number of days from the present day this message should be sent at. Negative numbers indicate a number of days in the past e.g. `-1` indicates the message should have a timestamp indicating it was sent yesterday (1 day ago). If not present, the message will appear to be sent on the current date.

e.g. -7

time ##:## Optional. A timestamp in 24-hour format that indicates the time this first message should appear to have been sent at. Requires a leading zero for times < 10am e.g. 1:05 AM is written as `01:05` and 1:30 PM is written as `13:30`.

e.g. time 05:16

Next, you will write the dialogue for the text messages. These are written like regular text messages, and you can use both the `msg` CDS as well as the spreadsheet method to write dialogue.

The text message backlog statement **does not** support the use of Python, jumps, or many other features besides dialogue inside the text message block; these messages are intended to be used as setup and are not part of active conversations, so any Python or calculations should be done outside of the `add backlog` CDS.

However, the `add backlog` CDS **does** support conditional if/elif/else statements so that you can vary messages based on certain conditions. In the example at the top of this section, you can see that the message varies based on the number of heart points the player has with certain characters.

`add backlog` does not support adding labels to jump to in order to reply to a conversation, as it is intended to setup "past" conversations. `compose text` is suitable for creating conversations that the player may respond to (see *The compose text CDS*).

9.4.1 Adjusting Timestamps

All lines, regardless of the way they're written, take the special option `time` after the dialogue. This allows you to further customize when the messages are sent at e.g.

```
add backlog ja -13:
    m "Good luck for your first day at the new job, Jaehee ^^" time 07:03
    ja "Thank you very much, [name]" time 07:15
    ja "{image=jaehee_happy}"
    ja "I will do my best!" time 07:18
```

The above example will have a timestamp 13 days in the past at 7:03 AM for the first message from the MC. The next message, from Jaehee, is also 13 days in the past but has a timestamp for 7:15 AM. The third message does not have an explicit timestamp, so the program will calculate a brief delay between the time given for the previous message and the time the program estimates it would take someone to write the current message. Since Jaehee is merely posting an emoji, this delay will be under a minute long so it will likely also have a timestamp of 7:15 AM. Finally, the last message has a timestamp at 7:18 AM.

Additionally, there is the special line `pause` which will cause the program to add the given number of seconds to the timestamp just after the pause statement. For example:

```
add backlog ju -17 time 20:32:
    ju "Good evening, [name]."
    ju "You've been with the RFA for a month now so I hoped to ask you a question."
    pause 110
    m "What is it?"
    pause 60*2
    ju "Would you consider a position at C&R International?"
```

The first message will have a timestamp of 20:32. The second message will take a few seconds to write but likely under a minute, so it will have a timestamp of 20:32 as well. After that, there is a pause statement for 110 seconds. This guarantees that the program will add 110 seconds to its calculation of how long the next message should take to type, so the MC's message will have a timestamp of 20:34.

Finally, after the MC's line there is a pause of 60*2 seconds, so 2 minutes. This will force an additional two-minute delay between the timestamp of the next message, from Jumin, and the MC's message. The last message from Jumin should have a time stamp of 20:36.

Note: The program will add several seconds to each backlog message's timestamp from the initial start timestamp unless an exact `time` argument is given. The exact number of seconds depends on the length of the message. This means that if a character has 10 messages in their text message backlog, there is likely to be at least a minute or more of difference in the timestamps of later messages to simulate the real-life time it would take to type out all 10 messages.

9.5 Manually Sending Text Messages

Normally, the program will take care of text message delivery after a story item. This allows for consistency across real-time and sequential play styles, as the program can execute an item's `after_` label without needing to check through its entire story label. However, there may be some situations where you prefer that a character sends a text message in response to some immediate event. For that, you can use the special delivery time `deliver_at now` to trigger a text message to send immediately.

```
label day_4_chatroom_5_incoming_y():
    y "[name]!! I thought of a really funny thing I wanted to share with you."
```

(continues on next page)

(continued from previous page)

```
    y "Um, one sec, I'll message you the picture."
    compose text y deliver_at now:
        y "look at this lolol"
        y "common_3" img
    y "It's good, right? I thought it was really creative.
```

In the above example, Yoosung sends the player a text message during a phone call. After Yoosung says "Um, one sec, I'll message you the picture." the player will see a text message popup with a preview of the last text message (in this case, because Yoosung sent an image, it will say "Yoosung sent an image."). This popup is non-interactive; there will not be a button to take the player to the text message immediately and they will need to wait until the phone call is over.

Note also that text messages sent in this way will only actually be delivered to the player's inbox if they complete the story item where the text message occurs. If they end or jump out of the story item prematurely, such as by hanging up or hitting the back button on chatrooms, the text messages will not be sent even though the player saw a notification during the story item itself. This prevents the player from accumulating a bunch of text messages without properly completing the story item.

Tip: Only the last message in a `compose text` block will be shown as a preview to the player. If you want to show them multiple message popups, you'll need to do so individually like:

```
compose text u deliver_at now:
    u "Ah, I found you."
z "Anyway, I've talked about myself so much aha."
z "So what are you up to?"
compose text u deliver_at now:
    u "You should log out now. I'll be there soon."
return
```

CHAPTER

TEN

STORY MODE (VN)

In this program, you can create Story Mode sections which can either be attached to a chatroom or can appear as its own timeline item. You declare "solo" Story Mode items in the route definition (see *Story Mode*).

10.1 Writing a Story Mode

Note: Example files to look at:

- tutorial_6_meeting.rpy
- tutorial_3b_VN.rpy

A brief overview of the steps required (more detail below):

1. If writing an attached Story Mode, create a label using the label of the attached chatroom + the suffix `_vn` (e.g. `label my_chatroom_vn`)
    1. (Optional) Indicate the character a Story Mode section is associated with by including their file_id as an additional suffix (e.g. `label my_chatroom_vn_s`)
    2. (Optional) Indicate that this Story Mode is the party by including the suffix `_party` (e.g. `label my_chatroom_party`)
2. Show your desired background with `scene bg my_background`.
    1. (Optional) Use transitions like `with fade` e.g. `scene bg your_bg with fade`
    2. (Optional) Write `pause` after your `scene` statement to give the player a moment to look at the background.
3. Add music with `play music your_music_var`.
4. Fill out the dialogue and character expressions.
5. End the label with `return`.

If this Story Mode is associated with a chatroom, the program uses a specific naming convention to know to set it up. For a Story Mode icon not associated with any character, you can just use the name of the chatroom + the suffix `_vn` e.g.

```
label my_chatroom:
```

will look for a generic Story Mode located at

```
label my_chatroom_vn:
```

If you would instead like to have a certain character's image associated with the Story Mode, you must add another suffix: _ + the file_id for that character e.g.

```
label my_chatroom_vn_y:
```

will show up in the program as a Story Mode associated with Yoosung.

Alternatively, you can also use the suffix `_party` to indicate a Story Mode contains the party e.g.

```
label my_chatroom_party:
```

Next, you can set up the Story Mode and begin writing dialogue. It's a good idea to start by setting a background using Ren'Py's `scene` statement:

```
scene bg rika_apartment with fade
```

`bg rika_apartment` is a background that is already defined in `vn_variables.rpy`. You can add your own backgrounds here as well to display in-game. Backgrounds should be 750x1334 pixels.

`with fade` indicates that the background should fade in from black. There are many more transitions you can use that are covered in Ren'Py's documentation Transitions.

If you want the player to have a moment to look at the background before you move on, you can write `pause` after showing the image e.g.

```
scene bg rika_apartment with fade
pause
```

10.2 Adding Music and SFX

Playing background music inside a Story Mode is the same as playing music elsewhere in the program; simply use

```
play music my_music_var
```

where `my_music_var` is a track included in the `music_dictionary` in `variables_music_sound.rpy`.

To add sound effects, use

```
play sound door_knock_sfx
```

where `door_knock_sfx` can be replaced by the name of whatever sound effect you want. There are several files already pre-defined in `variables_music_sound.rpy`.

Note that Ren'Py's built-in music and sound functions have been modified to work with audio captions for this program. The program will notify you if an audio caption has not been defined for an audio file.

If you would like to play a sound that does not have an audio caption, you can give `play music` or `play sound` the `nocaption` argument e.g.

```
play music ringtone nocaption
```

Warning: For accessibility purposes, most audio should have a caption, so this option should be used sparingly.

Previously defined audio captions can be found in `variables_music_sound.rpy`. To learn about adding your own audio, see *Adding New Audio*.

10.3 Shaking the Screen

You can shake the screen during Story Mode the same way as you do in chatrooms with `show shake`. However, for Story Mode, the screen will shake *on the line after the ``show shake`` line*. So, for example:

```
u "Hmm? What are you doing here?"
u "I told you not to come...!"
show shake
"(...!)"
u "Stop!"
```

The screen will shake on the "(...!)" line.

Note: The screen will not shake if the player has screen shake turned off in the accessibility options.

10.4 Providing Choices

Writing a menu in a Story Mode is identical to how it is written in phone calls; that is, you need to include `extend ''` after `menu:` and before the choices:

```
menu (paraphrased=True):
    extend ''
    "(Investigate the window)":
        u "Don't move."
    "(Stay where you are)":
        u "Why hello there."
```

This tells the program to display the last line of dialogue underneath the choice menu while it is on-screen.

Tip: If you don't have any dialogue *before* a menu (i.e. the menu is the first thing that happens in the story mode), you **don't** need to include `extend ''`. So, the following is correct:

```
label day_1_chatroom_1_vn():
    scene bg rika_apartment with fade
    pause
    play sound door_knock_sfx
    menu (paraphrased=True):
        "(Investigate the window)":
            u "Don't move."
        "(Stay where you are)":
            u "Why hello there."
```

There is no dialogue before the menu, so the `extend ''` is not necessary (and if included, would actually cause an error).

10.5 Awarding Heart Points

Heart points are awarded the same way as they are elsewhere in the game:

```
award heart ja
```

where `ja` is the variable of the character you're awarding a heart point for. See *Showing a Heart Icon* for more information.

10.6 Positioning Characters

There are several pre-defined positions you can move the characters to. These are:

- vn_farleft
- vn_left
- vn_midleft
- vn_center
- vn_midright
- vn_right
- vn_farright
- default

Not every position will work for every character due to spacing and differences in sprite design. You can define more positions in `vn_variables.rpy`.

`vn_center` is a unique position because it moves the character closer to the screen in addition to centering them. It's often used to imply the character is talking directly to the player. However, if you want to move a character who is currently shown in the `vn_center` position to another position, you must `hide` them first, or ensure that you reset the zoom to 1.0. See `tutorial_6_meeting.rpy` for an example of this.

To show a character at a given position, write

```
show jumin front at vn_right
```

where `jumin front` is the name + attributes of the character you want to show (e.g. expressions, outfits etc) and `vn_right` is the position to show them in. You can also add a transition like so:

```
show jumin side happy at vn_left with ease
```

where `ease` is a transition. (See Transitions in the Ren'Py documentation for more).

10.7 Changing Outfits and Expressions

To show a character, look at `character_definitions.rpy` and find the character you want to show under the **Character VN Expressions Cheat Sheet** header.

To show a character, use their name tag (usually their first name) + any additional expression, position, outfit, or accessory tags as applicable. In practice, this means if you want to show `jaehee` with the expression `happy` and wearing her `glasses`, write

```
show jaehee glasses happy
```

Since Jaehee does not have any additional positions (such as a "side" or "front" position), you don't need to include that when you show her on-screen. However, a character like V *does* have multiple positions, and so a similar statement for him would look like

```
show v side angry glasses
```

where `v` is the character, `side` is the position, `angry` is his expression, and `glasses` indicates he should be shown wearing glasses.

To change the character's outfit, when you show them you should include the attribute name of the outfit you'd like them to wear e.g.

```
show yoosung suit surprised
```

In this case, `yoosung` is the character, `suit` is his outfit, and `surprised` is his expression.

Note that attributes can be listed in any order after the character, so `show yoosung surprised suit` is equally correct.

Characters with accessories (glasses, hoods, masks) can include those keywords as well e.g.

```
show v front mint_eye worried hood_up at vn_right with easeinright
```

which will show `V` in his `front` position wearing his `mint_eye` cloak with the `hood_up` at the position `vn_right` after being "eased in" (`easeinright`) from the right side of the screen.

Luckily, after you've shown a character on-screen the first time, Ren'Py will remember which attributes you showed them with so that you don't have to write out the whole attribute list every time after that. For example:

```
show v front mint_eye worried hood_up at vn_right with easeinright
v "Some sample dialogue."
show v thinking
v "More dialogue."
```

The second `show` statement remembers the attributes `v` was shown with previously, namely `front mint_eye hood_up` since the `thinking` expression replaces the `worried` expression. So he will still be shown in his front position with the mint_eye cloak and his hood_up, just with the `thinking` expression.

10.7.1 Attribute Shorthand

There is also another way of simplifying showing characters even further – since characters are defined with an "image" tag in mind, after showing them on screen the first time you can change tags directly during dialogue like so:

```
show seven front happy party
s "Some dialogue while happy."
s worried "Some dialogue with the party outfit in the front pose, but worried."
s normal "Now in the normal outfit, and still worried and in the front position."
```

Note that you can only change expressions this way after the character has already been shown on-screen.

For more on Image Attributes, see the Ren'Py documentation pages on Dialogue and Narration.

10.7.2 Hiding Characters

When you're done showing a character on-screen, you only have to use their name to hide them:

```
show seven front happy party
s "Some dialogue while happy."
hide seven
```

10.8 Including a Story Mode During a Chatroom

Note: Example files to look at:

- tutorial_9_storytelling.rpy

A brief overview of the steps required (more detail below):

1. Create a chatroom as you usually would (see *Creating a Chatroom*.
2. When you want the story mode to interrupt the chatroom, use `call vn_during_chat("name_of_story_mode_label")` where `name_of_story_mode_label` is the name of the label where the program can find the associated story mode.
    1. If you want to change the list of chatroom participants between chatroom sections, pass the argument `reset_participants=[y, s, m]` where `[y, s, m]` is a list of the character variables who you want to show participating in the chatroom section following the story mode.
3. If you don't want to return to the chatroom section after jumping to the story mode section, use the argument `end_after_vn=True` e.g. `call vn_during_chat("vn_label", end_after_vn=True)`.
4. Create the label for the story mode and write the dialogue as normal (see *Writing a Story Mode*).
5. If the timeline item does not end on the story mode, continue writing chatroom dialogue after the call to `vn_during_chat`.
    1. (Optional) use `clear chat` to erase the chat history.
    2. (Optional) change the chat background with a `scene` statement.
    3. (Optional) clear all participants from the chatroom with `clear chat participants`.
6. End the chatroom and any story mode labels with `return`.

Tip: This section covers how to include a story mode section **in the middle of a chatroom**, such that the player will be viewing the chatroom and it will transition into a story mode section and back to the chatroom in the same "scene".

This is different from how Story Mode sections usually play out in-game; that is, first the player plays a chatroom, is returned to the timeline screen, and then must select the corresponding Story Mode section to proceed. If you are interested in the latter, see *Attached Story Mode*.

First, you'll begin by setting up a chatroom the way you usually would (see *Creating a Chatroom*). Next, wherever you would like to have the story mode appear and interrupt the chatroom, write `call vn_during_chat("my_vn_label")` where `my_vn_label` is the name of the label you will be using for your story mode section.

Warning: Be sure your mid-chatroom story mode label **doesn't** follow one of the naming conventions for declaring an attached Story Mode – so, if your chatroom is at `label my_chatroom`, then don't call the mid-chatroom story mode `label my_chatroom_vn` or the program will generate an attached Story Mode leading to that label!

Create this label and write the story mode. It is written the same way as a regular story mode – see *Writing a Story Mode*. Example:

```
label my_chatroom:
    scene morning
    play music silly_smile_again
    enter chatroom y
    y "Good morning!"
    call vn_during_chat("my_chatroom_vn_chat")
    y "And we're back! Hope you enjoyed that."
    exit chatroom y
    return

label my_chatroom_vn_chat:
    scene bg cr_meeting_room
    play music lonesome_practicalism
    show jaehee glasses happy
    ja "Oh dear. It seems the new intern has spilled coffee on this file."
    return
```

Once the story mode is finished, the program will return to where it left off in the chatroom. All the previous messages will remain on-screen in the chatlog, and the background will be what it was before the story mode.

Note: If you begin a chatroom with the `call vn_during_chat` call before any dialogue is said in the chatroom, it will begin immediately in the VN portion instead of showing the player the chatroom with a Continue button.

10.8.1 Clearing the Chat History

If you would like to clear the chat history after returning from the Story Mode section (that is, previous messages in the chatroom will be erased), you can use the `clear chat` CDS:

```
y "Anyway, I should go now."
exit chatroom y
call vn_during_chat('my_chatroom_vn_chat')
clear chat
enter chatroom y
y "Hi, [name]."
```

10.8.2 Modifying Chatroom Participants

If you want to modify the list of participants between scenes, you can either clear the participants altogether or pass `vn_during_chat` a special `reset_participants` argument. First, to remove all participants from the chatroom, use `clear chat participants`:

```
label my_chatroom:
    scene morning
    play music narcissistic_jazz
    enter chatroom z
    z "Darn... I was hoping to catch Jaehee in the chatroom at this hour."
    z "I wonder what she's up to..."
    call vn_during_chat("my_chatroom_vn_chat")
    clear chat participants
    enter chatroom ja
    ja "And the day begins to go downhill..."
    ja "I suppose everyone is busy doing something else at this hour."
    ja "I will log off as well, then."
    exit chatroom ja
    return

label my_chatroom_vn_chat:
    scene bg cr_meeting_room
    show jaehee glasses at vn_midleft
    show jumin front at vn_midright
    ju "Assistant Kang. I need you to reschedule a meeting."
    ja "Oh... I see. Which meeting do you need to reschedule?"
    ju "The one for the Cultured Citizens group this afternoon."
    ju "I received word that Elizabeth the 3rd is acting particularly lethargic today␣
↪so I will need to return to check on her."
    ja "...Understood."
    return
```

In this example, when the player returns from the Story Mode section, the chat history will be cleared and the characters who were present in the chatroom prior to the story mode (aka Zen) will no longer be shown as in the chatroom. The main character will be automatically added back to the list of people present if this chatroom is not expired; otherwise, no one will be in this chatroom when the story mode returns to the chatroom.

You can also tell the program exactly who you would like to appear in the chatroom when the story mode is finished via `reset_participants`:

```
label day_3_8:
    scene night
    play music mint_eye
```

(continues on next page)

(continued from previous page)

```
    r "I know I've been really busy lately,"
    msg r "But I wanted to let you know that I'll be over with food shortly!" curly_
↪glow
    msg r "I made it myself ^^" glow
    r "See you soon!"
    call vn_during_chat("day_3_8_story_vn", reset_participants=[s, z])
    clear chat
    scene earlyMorn
    play music narcissistic_jazz
    z "You know, we haven't heard from [name] for a few hours..."
    z "I hope [they_re] doing all right."
    s "lolol ur impatient aren't u."
    s "I'm sure [they] just logged off for the night."
    z "You're probably right. I should go to sleep, too."
    s "Toodles~!"
    exit chatroom z
    exit chatroom s
    return

label day_3_8_story_vn:
    scene bg mint_eye_room
    play sound door_knock_sfx
    "(A knock at the door)"
    menu:
        extend ''
        "(Answer the door)" (paraphrased=True):
            pass
        "Who is it?":
            r "It's me, Ray! I've come to bring you dinner."
    play sound door_open_sfx
    "(Door opened)"
    show saeran ray happy
    r "Hi, [name]. I really missed you today."
    r "I brought some food. I hope it's to your liking."
    return
```

In the above example, after the game returns from the story mode, the chat log is cleared and Zen and 707 are added to the list at the top of the chatroom to show that they are present. The main character is *not* added; in order to include them, the argument would need to be `reset_participants=[s, z, m]` instead (so, it needs to include `m`).

The background for the chatroom is also modified upon returning with another `scene` statement, and new music is played. Unless otherwise specified, any music that is already playing during the previous chatroom or story mode will be carried over into the next section until the timeline item ends. If you want to stop the music altogether, you can use the line `stop music`.

10.8.3 Ending a Timeline Item after a Story Mode Section

If you don't want the player to return to the chatroom and instead want the timeline item to end on the story mode section, you can pass the argument `end_after_vn=True` to `vn_during_chat` e.g.

```
u "Anyway, this is the end of the scene."
exit chatroom u
call vn_during_chat("my_final_vn_label", end_after_vn=True)
return
```

CHAPTER

ELEVEN

EMAILS

Note: Example files to look at:

- tutorial_2_emails.rpy
- email_template.rpy

A brief overview of the steps required (more detail below):

1. Open **email_template.rpy** and copy-paste `example_guest` into a new file.
2. Rename the guest and follow the instructions to fill out the various fields.

11.1 Writing an Email Chain

Emails in this program have several additional features:

- You may have a varying number of emails in a chain (limited only by how many email icons will fit comfortably on the email button)
- You may have as as few as one or up to five responses available to answer
- Emails can have "branching paths"; simply answering one email incorrectly does not necessarily mean the whole email chain is failed.
- Guests are automatically added to the guestbook. Some information about them is unlocked when the player invites them to the party, and additional information is added after the guest has attended the party.
- Viewing a guest's information for the first time in the guestbook awards the player an hourglass.

The easiest way to get started writing an email is to copy the definition for `example_guest` inside **email_template.rpy** and modify it to your needs. The various fields are explained with comments in the file itself so you understand what information to give it. The newest feature is the use of `EmailReply` objects, which allow email chains a greater degree of flexibility. This is explained below.

EmailReply(choice_text, player_msg, guest_reply, continue_chain=None, email_success=None)

A class intended to facilitate writing email replies.

choice_text A string. The text of the choice to reply to the email.

player_msg A string. The message the player writes after the choice is made.

guest_reply A string. The guest's reply to the player's message.

continue_chain If this email chain will continue after the player selects this reply, this is a list of EmailReply objects that will be available the next time the player is given the opportunity to reply.

email_success If explicitly set to a boolean value, this indicates if it ends the email chain in a good (True) or bad (False) way.

Both the `player_msg` and `guest_reply` fields use a special helper function called `filter_whitespace`, which removes leading and trailing whitespace (mostly the "space" character) from the string and turns a single newline into a space. Double newlines are preserved.

In practice, this means you can use line breaks to keep your email messages readable inside your code editing program, and can use indentation to better show a chain of emails.

If you want to preserve single newlines exactly as written, there is a variable in `variables_editable.rpy` called `email_newline_to_space`. By default, this value is `True` (aka it will convert single newlines to a space), but you can set it to `False` in order to have newlines remain exactly as written. Note that you'll need to start a new game in order to see changes in email display.

In order to have "branching paths" for emails, you must use the `continue_chain` field. The example below demonstrates how to write an email chain with three initial responses that each lead to a different second email:

```
default longcat = Guest(
"longcat",
"Long Cat",
"Email/Thumbnails/longcat.webp",
"Email/Guest Images/longcat.webp",
"Long Cat, the longest cat in the world.",
"Meow? Mrrrow... meow meow.",
"""To [name]:

Meow? Meow meow.

From, Long Cat""",

[ EmailReply(
    "There will be fish at the party",

    """Dear Long Cat,

    We will have fish at the party, meow!

    From, [name]""",

    """To [name]:

    Meow!!! Mew meowww. Mrow?

    From, Long Cat""",

    [ EmailReply(
        "Tuna fish",

        """Dear Long Cat,

        The fish will be tuna fish, meow!

        From, [name]""",

        """To [name]:

        Mrow... mew.
```

(continues on next page)

(continued from previous page)

```
        From, Long Cat""",

        email_success=True

    ), EmailReply(
        "Salmon",

        """Dear Long Cat,

        We will have salmon available at the party, meow!

        From, [name]""",

        """To [name]:

        Meow!!! Mew meow mew.

        From, Long Cat""",

        email_success=True

    )]

), EmailReply(
    "Laser pointers",

    """Dear Long Cat,

    We will be giving out laser pointers to the guests at the party! I
    hope you find this a source of entertainment.

    From, [name]""",

    """To [name]:

    Hiss!!! Mrow... meow.

    From, Long Cat""",

    email_success=False,

    continue_chain=[ EmailReply(
        "No laser pointers",

        """Dear Long Cat,

        My apologies for the misunderstanding; there will no longer be laser
        pointers at the party. Hope to see you there!

        From, [name]""",

        """To [name]:

        Mew mew meow!

        From, Long Cat""",
```

(continues on next page)

(continued from previous page)

```
        email_success=True

    ), EmailReply(
        "Yarn",

        """Dear Long Cat,

        Beside the laser pointers, we will also have plenty of yarn for your
        enjoyment. Hope to see you there!

        From, [name]""",

        """To [name]:

        Hisssss! Hiss!!!

        From, Long Cat""",

        email_success=False
    )]

), EmailReply(
    "Meow meow meow!",

    """Dear Long Cat,

    Meow! Mew mew mew, mrow, meow!

    From, [name]""",

    """To [name]:

    Purrrrrr.... meow.

    From, Long Cat""",

    email_success=True

)],

"Meow meow! Purr....",
s,
"Omg it's Long Cat!!! I need to find and pet them...",
"seven front party happy",
num_emails=2
)
```

Though the example above is long, it functions as follows:

```
first email:
    "There will be fish at the party" (correct)
        Second email:
            "Tuna fish" (correct)
            "Salmon" (correct)
    "Laser pointers" (wrong)
        Second email:
            "No laser pointers" (correct)
```

(continues on next page)

(continued from previous page)

```
            "Yarn" (wrong)
    "Meow meow meow!" (correct)
```

Incorrect answers decrease the guest's odds of attending the party, while correct answers will increase the odds. A "perfect" email chain (one where `num_emails` emails are replied to correctly) will guarantee the guest's attendance at the party.

11.2 Calculating if a Guest Will Attend

The program follows its own set of logic for determining if a guest will attend the party. A guest is **guaranteed** to attend the party when:

- The player has correctly answered `num_emails` with the guest
- The player has read the guest's final reply and the email is marked as Completed

A guest will **never** attend the party if:

- The player has not read the most recent email
- The email chain is not complete
- The player did not get any emails correct
- The email chain has timed out

If the player has read all the guest's emails and reached the end of an email chain with them, then the odds of the guest attending depend on how many emails the player got correct with them. In the Long Cat example above, if the player answers "There will be fish at the party" -> "Tuna fish" or "There will be fish at the party" -> "Salmon", then Long Cat is guaranteed to attend the party.

If the player answers "Laser pointers" -> "Yarn" then the email chain will be failed and Long Cat will not attend the party.

If the player answers "Laser pointers" -> "No laser pointers", then Long Cat will have a 50% (1/2) chance of attending the party.

If the player answers "Meow meow meow!" then Long Cat will have a 50% chance of attending the party, even though there are no further emails, since `num_emails` is 2 and the player only got 1 answer correct.

The odds of a guest attending the party are calculated after the email is answered. In other words, if a guest has a 50% chance of attending the party, reloading a save just before playing the party will not change whether they attend or not.

A guest will be unlocked in the guest book as soon as the player invites them to the party, but further information will only be unlocked if the guest actually attends the party.

11.2.1 Checking in-game if a Guest Will Attend

Once an email chain is completed, you can check whether or not a particular guest is attending via the `attending` field e.g.

```
if rainbow.attending:
    ja "It's so good to see Rainbow here!"
```

11.3 Inviting a Guest

After defining your guest, you can invite them to the party using the line:

```
invite your_guest
```

where `your_guest` is the variable you made when defining the guest using

```
default your_guest = Guest(...)
```

Tip: You can also use `call invite(your_guest)` to invite a guest.

After the player finishes the timeline item where the guest was invited, they will receive the first email from the guest.

11.4 How Emails are Sent

After the guest is invited to the party, the player will receive an initial email from them. From that point onwards, if the player **does not** reply to the email, after every timeline item a variable attached to every Email object called `timeout` will decrease by 1. By default, `timeout` begins at 25.

The timeout counter will reset to 25 as soon as the player replies to the email. However, if the timer reaches 0 (aka the player has gone through 25+ timeline items since they first received the email), the email will be considered **timed out** and can no longer be replied to.

Once the player replies to the email, the program calculates when it should send the guest's reply based on how many timeline items remain in the route. For example, if there are 20 remaining timeline items on the route and 2 remaining emails in the chain, then the maximum number of timeline items the program will wait before delivering an email reply is `20 / 2 = 10`. The minimum number of timeline items it will wait is 1 or `10 - 7 = 3`, whichever is larger – in this case, 3.

11.5 The Guestbook

There are several fields when defining a Guest which correspond to text which will appear in the guestbook. These are:

comment_who The ChatCharacter object of the character who will talk about this guest in the guestbook.

e.g. `s`

comment_what What the above character will say about the guest.

e.g. "I can't believe I got to meet Long Cat!"

comment_img A string that corresponds to a defined image or layeredimage attributes to display the sprite of the character who is talking about the guest.

e.g. "seven front party happy"

The player will also receive 1 hourglass the first time they view a guest's entry in the guestbook after successfully inviting them to the party. The `short_desc` field's contents are shown to the player in the guestbook after they first invite the guest, and the `personal_info` field is revealed in the guestbook only after the guest has successfully attended the party.

11.6 Testing Emails

While you are writing a route, you might want to test email chains and party comments without going through the whole game just to invite the guest. To do that, load up a save file and then click the Developer button from the home screen. Next, click on **Test Emails**.

Here you will find a list of all the guests the game could find to invite. Simply click the guest's name to invite them to the party. Some other buttons are available at the bottom of the screen:

Force email replies Causes all undelivered emails to be delivered at once. (For example, if you sent a reply to a guest, then clicking **Force email replies** would immediately deliver the guest's response to your inbox).

Invite all guests Invites every possible guest. This includes guests which aren't accessible to invite in-game but were declared as a Guest. All their emails will appear in your inbox. Note that it is possible to invite the same guest more than once with this method. Inviting the same guest multiple times can sometimes cause odd side effects.

Start Party Begins the guest showcase part of the party. Guests only attend if their email chains have been successfully completed as described in *Calculating if a Guest Will Attend*.

Indicate correct answers If turned on, the "correct" email answer will be have a checkmark in the corner when replying to emails.

11.7 Email Callbacks

You can also optionally specify a callback for a guest's email chain. This callback will be called run each time the email is marked as "read".

As an example, say you would like to set a variable to True in the specific case that the player has seen the guest's second email reply (not including the initial "introductory" email).

First, you need to set up a Guest and specify that they have a callback. For this example, we'll use the Long Cat guest from earlier. In particular, to keep this short, the EmailReply objects are **omitted**:

```
default longcat = Guest(
"longcat",
"Long Cat",
"Email/Thumbnails/longcat.webp",
"Email/Guest Images/longcat.webp",
"Long Cat, the longest cat in the world.",
"Meow? Mrrrow... meow meow.",
"""To [name]:

Meow? Meow meow.

From, Long Cat""",

## EMAIL REPLIES OMITTED FROM EARLIER EXAMPLE FOR BREVITY
[ EmailReply(...)],

"Meow meow! Purr....",
s,
"Omg it's Long Cat!!! I need to find and pet them...",
"seven front party happy",
num_emails=2,
callback=longcat_email_callback
)
```

Here you can see at the very end that the argument `callback=longcat_email_callback` was added. This is the name of a function that must be defined. Typically it is a good idea to define it above your Guest definition.

An email callback takes one parameter, the email object associated with the callback. You can then access the various email fields to get information about the guest or email in progress. An example email callback for longcat might look like:

```
init python:
    def longcat_email_callback(email):
        global saw_longcat_email2
        if email.msg_num >= 2:
            saw_longcat_email2 = True
        return

default saw_longcat_email2 = False
```

This sets the variable `saw_longcat_email2` to True when the player reads Long Cat's second reply (not counting Long Cat's initial email message). There are many other email fields you can access to determine what the program should do.

Note: This callback will be triggered every time the player opens the email chain. Thus, the callback generally should not have what are called "side effects", things that you wouldn't want to happen every time the player opened the email. For example, if you want to increase a counter, you should have a flag to ensure you only increase the counter once, on the first viewing of an email e.g.

```
def longcat_email_callback(email):
    global saw_longcat_email2, my_email_counter
    if email.msg_num >= 2 and not saw_longcat_email2:
        saw_longcat_email2 = True
        my_email_counter += 1
```

The check `and not saw_longcat_email2` ensures that the code `my_email_counter += 1` only happens once, the first time the Long Cat email chain is marked as read. Without it, `my_email_counter` will increase by 1 every time the player views the email (which is probably not what you intended).

CHAPTER

TWELVE

SETTING UP A ROUTE

Note: Example files to look at:

- route_setup.rpy
- route_example.rpy

A brief overview of the steps required (more detail below):

1. Define a list of `RouteDay` objects. The first item in the list should be the name of the ending e.g. `"Good End""`.
2. The first parameter of the RouteDay object is the name of the day e.g. "1st". The second is a list of timeline items. Possible timeline items are:
    1. ChatRoom("title", "label", "trigger_time", [participants])
    2. StoryMode("title", "label", "trigger_time", associated_character)
    3. StoryCall("title", "label", "trigger_time", caller)
    4. TheParty("label", "trigger_time")
3. Create a `Route` object and fill in the information for your new route.
4. Customize the route select screen to include a button to your new route.
5. Create a short introduction for your new route.
6. Select **Start Over** from the settings to test out your new route.

To set up a route and test out your own chatrooms, story mode, phone calls and more in the game, first you need to define a few special variables. The first is a list of `RouteDay` objects. Each `RouteDay` contains the information the program needs to know for one in-game day.

First, go to `route_setup.rpy`, where you'll see an example definition called `tutorial_good_end`.

At its most basic level, your definition will look as follows:

```
default my_route_good_end = [ "Good End",
    RouteDay("1st"),
    RouteDay("2nd"),
    RouteDay("3rd"),
    # (...)
    RouteDay("Final")
]
```

The first item in the list should be a string with the name of the ending as it will appear in the History log. In this example, it is `"Good End"`.

The remaining items in the list are `RouteDay` objects. They have the following fields:

day A string containing the name of the day as it should appear in the timeline.

e.g. "1st"

Tip: A RouteDay with the day field as "Final" will have a special icon above it in the timeline screen.

---

archive_list Optional, although empty days have no content. A list of timeline items. See below for more.

e.g. [ChatRoom("Example Chatroom", "example_chat", "00:01")]

day_icon Optional. A string with the name of the icon to use for this day in the timeline. Defaults to `"day_common2"`. Existing images can be found in `variables_editable.rpy` under the header **DAY SELECT IMAGES**.

e.g. "day_ju"

branch_vn Optional. If this day has a Story Mode that should be shown as soon as it's merged onto the main route after a plot branch, then it will be stored here.

e.g. BranchStoryMode('jaehee_bre2_vn', who=ja)

save_img Optional. The file path or short form to the save image which should be used for all timeline items on this RouteDay. Existing images can be found in `variables_editable.rpy` under the header **SAVE & LOAD IMAGES**.

e.g. "zen"

auto_label Optional. A string which is used as the pattern to automatically name all item labels inside this RouteDay's `archive_list`. Each item in `archive_list` with None as its label will be given a label with this prefix + a number in increasing order.

e.g. '**casual_d3_**'

exclude_suffix Optional. A boolean value that controls whether the " Day" suffix is added to the end of the day name on the timeline screen. The default behaviour is to append " Day" to the end of the title given in the `day` field, so "1st" appears as "1st Day" on the timeline screen. If `exclude_suffix` is True, however, then " Day" will not be appended to the name of the day so it would appear simply as "1st".

e.g. True

12.1 Adding Timeline Items

The main way you will add content to your route is by filling out a RouteDay's `archive_list`. There are four main timeline items.

12.1.1 Chatrooms

To add a Chatroom, you will use the `ChatRoom` class. It contains all the information the program needs for a single chatroom along with any accompanying phone calls or a story mode section, if applicable. You will need one `ChatRoom` object for each chatroom in your route.

A typical chatroom definition looks like the following:

```
ChatRoom("Fight over cats", 'casual_d2_example_4', '08:05', [ja, ju])
```

For this example, the title of the chatroom is "Fight over cats". It can be found at the label `casual_d2_example_4` and will appear at 8:05 am. The characters `ja` and `ju` begin in the chatroom.

There are some additional fields as well, each of which is explained below.

title The title of the chatroom as it should appear in the timeline. A string.

e.g. "Yoosung's omelette rice"

chatroom_label The name of the label the program will jump to in order to play this chatroom. You must define this label yourself. It should be passed to this field as a string.

e.g. "casual_d2_example_3"

trigger_time The time this chatroom should appear at. This should be written in military time with leading zeroes, so a time like 1:00 AM becomes "01:00" and 1:38 PM becomes "13:38".

e.g. "23:42"

participants Optional. A list of the characters who begin in this chatroom. If this field is ommitted, no one begins in the chatroom. This should be a list of ChatCharacter objects.

e.g. [ja, ju]

story_mode Optional. Allows you finer control over the StoryMode object associated with this chatroom. The program will automatically try to find appropriately labelled story mode labels and create its own StoryMode object for this field.

e.g. StoryMode("", "my_vn_label")

> **Warning:** It is generally recommended that you let the program take care of defining StoryMode objects attached to chatrooms by using properly named labels. See *Attached Story Mode*.

plot_branch Optional. Indicates that there should be a plot branch after this chatroom. See *Plot Branches* for more on plot branches.

e.g. PlotBranch(True)

save_img Optional. A string with the name of the save image to use for this particular chatroom. This takes precedent over a save image set by the RouteDay. It is typically an image indicating which route the player is on. The prefix `"save_"` is automatically added to this field.

e.g. "casual"

box_bg Optional. If you want the chatroom's timeline image box to have one of the special backgrounds, you'll specify which one – either "colorhack" or "secure". "secure" displays a gear image beneath the title and participant pictures, and "colorhack" displays several coloured squares. If omitted, the regular box background is used.

e.g. "colorhack"

Chatrooms also have "expired" versions. This means that if the player is playing real-time mode and doesn't play this chatroom before the next timeline item appears, it will "expire". The program will look for the expired version of a chatroom under the chatroom's regular label + the suffix `_expired`. So, if the chatroom is found at

```
label day_4_5_coffee_chat:
```

Then the expired label should be located at

```
label day_4_5_coffee_chat_expired:
```

Typically, an expired chatroom covers most of the same topics as the original chatroom, though the player is not present. For more on expired chatrooms and real-time mode, see *Expired Timeline Items and Real-Time Mode*.

Attached Story Mode

Chatrooms can have an attached Story Mode, which becomes available to play after the chatroom has been played. The easiest way to do this is to create a label which follows a specific naming scheme. This will automatically define a StoryMode object for the associated chatroom.

If your chatroom's label is `casual_day_1_4`, then if you create a label called `casual_day_1_4_vn` (note the `_vn` suffix), a general StoryMode will be created that will lead to that label.

You can also specify which character should appear on the Story Mode icon in the timeline screen by adding their file_id to the end of the label e.g. `casual_day_1_4_vn_ja` would create a Story Mode attached to the chatroom found at `casual_day_1_4` which will show Jaehee's image on the icon. Most of the existing characters have an associated Story Mode icon defined for them. You can also define one for a new character: see *Story Mode Timeline Images* for more.

An attached Story Mode is written the same way as other story mode sections. See *Writing a Story Mode* for more.

12.1.2 Story Mode

To add a **standalone** Story Mode section, you will use the `StoryMode` class. It contains all the information the program needs for a single story mode along with any accompanying phone calls, if applicable. You will need one `StoryMode` object for each standalone story mode in your route.

A Story Mode definition looks like the following:

```
StoryMode("Gone to the store", "extra_day_5_3", "20:16", y, save_img='y', plot_
↪branch=PlotBranch(False))
```

For this example, the title of the Story Mode is "Gone to the store". It can be found at the label `extra_day_5_3` and will appear at 8:16 pm. The story mode icon in the timeline will show Yoosung's story mode image, and if the player saves the game while this is the most recent timeline item, it will show a save icon associated with "y". Additionally, there is a plot branch after this Story Mode. For more on Plot Branches, see *Plot Branches*.

There are some additional fields as well, each of which is explained below.

title The title of the story mode as it should appear in the timeline. A string.

e.g. "Things that matter"

vn_label The name of the label the program will jump to in order to play this story mode. You must define this label yourself. It should be passed to this field as a string.

e.g. "casual_d2_example_3"

trigger_time The time this story mode should appear at. This should be written in 24-hour time with leading zeroes, so a time like 1:00 AM becomes "01:00" and 1:38 PM becomes "13:38".

e.g. "03:42"

who Optional. The ChatCharacter this story mode should be associated with. Leaving this out causes the story mode to use a "general" story mode icon on the timeline screen.

e.g. ja

plot_branch Optional. Indicates that there should be a plot branch after this story mode. See *Plot Branches* for more on plot branches.

e.g. PlotBranch(False)

party True if this Story Mode is the party. In general, you should instead use `TheParty` to define the party for your route. See *The Party*.

e.g. False

save_img Optional. A string with the name of the save image to use for this particular story mode. This takes precedent over a save image set by the RouteDay. It is typically an image indicating which route the player is on. The prefix `"save_"` is automatically added to this field.

e.g. "s"

12.1.3 Story Calls

To add a **standalone** Story Call, you will use the `StoryCall` class. It contains all the information the program needs for a single story call along with any accompanying regular phone calls, if applicable. You will need one `StoryCall` object for each standalone story call in your route.

A Story Call definition looks like the following:

```
StoryCall("Did you hear?", "new_years_3", "16:10", s)
```

For this example, the title of the Story Call is "Did you hear?". It can be found at the label `new_years_3` and will appear at 4:10 pm. The person calling the player is `s`.

There are some additional fields as well, each of which is explained below.

title The title of the story call as it should appear in the timeline. A string.

e.g. "Good morning!"

phone_label The name of the label the program will jump to in order to play this story call. You must define this label yourself. It should be passed to this field as a string.

e.g. "new_years_3"

trigger_time The time this story call should appear at. This should be written in 24-hour time with leading zeroes, so a time like 1:00 AM becomes "01:00" and 1:38 PM becomes "13:38".

e.g. "11:11"

caller The ChatCharacter object of the character who will be calling the player.

e.g. r

plot_branch Optional. Indicates that there should be a plot branch after this story call. See *Plot Branches* for more on plot branches.

e.g. PlotBranch(False)

save_img Optional. A string with the name of the save image to use for this particular story call. This takes precedent over a save image set by the RouteDay. It is typically an image indicating which route the player is on. The prefix `"save_"` is automatically added to this field.

e.g. “ju”

Story calls, like chatrooms, also have “expired” versions. This means that if the player is playing real-time mode and doesn’t play this story call before the next timeline item appears, it will “expire”. The program will look for the expired version of a story call under the story call’s regular label + the suffix `_expired`. So, if the story call is found at

```
label my_first_story_call:
```

Then the expired label should be located at

```
label my_first_story_call_expired:
```

Typically, an expired story call is treated as though the caller phoned the player and left a voicemail. For more on expired story calls and real-time mode, see *Expired Timeline Items and Real-Time Mode*.

12.1.4 The Party

There is a special convenience function intended to help you define a standalone party for a route. It has the following fields:

vn_label The label the program should jump to to play the party.

e.g. “emma_route_party”

trigger_time The time the party should appear at. This should be written in 24-hour time with leading zeroes, so a time like 1:00 AM becomes “01:00” and 1:38 PM becomes “13:38”.

e.g. “12:00”

save_img Optional. A string with the name of the save image to use when this is the most recent timeline item. This takes precedent over a save image set by the RouteDay. It is typically an image indicating which route the player is on. The prefix `"save_"` is automatically added to this field.

e.g. “em”

An example may look like:

```
TheParty("emma_route_normal_party", "12:00")
```

12.1.5 Example Route Day

An example route day might look like the following:

```
RouteDay('1st',
    [ChatRoom('Welcome!', 'day_1_emma_1', '00:01'),
    ChatRoom('Relaxing','day_1_emma_2', '06:11', [z, em]),
    ChatRoom('How are you doing?', 'day_1_emma_3', '09:53', [r]),
    ChatRoom('Something strange...', 'day_1_emma_4', '11:28', [s]),
    StoryMode('Do you...?', 'day_1_emma_5', '15:05', em),
    ChatRoom('Kimchi Sandwich', 'day_1_emma_6', '18:25', [em, ja]),
    ChatRoom('Very mysterious', 'day_1_emma_7', '20:41'),
    StoryCall('Will you visit?', 'day_1_emma_8', '22:44', em,
```

(continues on next page)

(continued from previous page)

```
        plot_branch=PlotBranch(True)),
    ChatRoom("Happily Ever After", 'day_1_emma_9', '23:26')
    ], save_img="em")
```

A list of RouteDays like this make up a single path on a route:

```
default emma_good_end = ["Good End",
    RouteDay('1st',
        [ChatRoom('Welcome!', 'day_1_emma_1', '00:01'),
        ChatRoom('Relaxing','day_1_emma_2', '06:11', [z, em]),
        ChatRoom('How are you doing?', 'day_1_emma_3', '09:53', [r]),
        ChatRoom('Something strange...', 'day_1_emma_4', '11:28', [s]),
        StoryMode('Do you...?', 'day_1_emma_5', '15:05', em),
        ChatRoom('Kimchi Sandwich', 'day_1_emma_6', '18:25', [em, ja]),
        ChatRoom('Very mysterious', 'day_1_emma_7', '20:41'),
        StoryCall('Will you visit?', 'day_1_emma_8', '22:44', em,
            plot_branch=PlotBranch(True)),
        ChatRoom("Happily Ever After", 'day_1_emma_9', '23:26')
        ], save_img="em"),
    RouteDay('2nd', [ChatRoom(...)]),
    RouteDay('3rd', [ChatRoom(...)]),
    RouteDay('4th', [ChatRoom(...)]),
    RouteDay('5th', [ChatRoom(...)]),
    RouteDay('6th', [ChatRoom(...)]),
    RouteDay('7th', [ChatRoom(...)]),
    RouteDay('8th', [ChatRoom(...)]),
    RouteDay('9th', [ChatRoom(...)]),
    RouteDay('10th', [ChatRoom(...)]),
    RouteDay('Final', [ChatRoom(...)])]
```

Note that `[ChatRoom(...)]` is shorthand for a list of many more timeline items.

Tip: The above RouteDay example could also be simplified as:

```
RouteDay('1st',
    [ChatRoom('Welcome!', None, '00:01'),
    ChatRoom('Relaxing', None, '06:11', [z, em]),
    ChatRoom('How are you doing?', None, '09:53', [r]),
    ChatRoom('Something strange...', None, '11:28', [s]),
    StoryMode('Do you...?', None, '15:05', em),
    ChatRoom('Kimchi Sandwich', None, '18:25', [em, ja]),
    ChatRoom('Very mysterious', None, '20:41'),
    StoryCall('Will you visit?', None, '22:44', em, plot_branch=PlotBranch(True)),
    ChatRoom("Happily Ever After", None, '23:26')
    ], save_img="em", auto_label="day_1_emma_")
```

Note the use of `None` instead of a label name, and `auto_label="day_1_emma_"` on the RouteDay itself. This will automatically name all the labels inside the RouteDay to be equivalent to the previous example.

12.2 Setting up the Route

Now you should have at least one definition of a route path with a list of RouteDay objects. In order for the route to show up in the History screen and be playable, you also need to define a `Route` object. Do this after you have defined lists of RouteDay objects for each branching path on your route.

An example Route may look like the following:

```
default new_years_route = Route(
    default_branch=new_years_normal_end,
    branch_list=[new_years_ju, new_years_ja,
        new_years_s, new_years_y, new_years_z],
    route_history_title="New Year's",
    history_background="Menu Screens/Main Menu/new_years_route_bg.webp"
)
```

This defines a special New Year's Eve Route with endings for five of the characters, as well as a normal ending. Each of the fields is explained below.

default_branch The "default" path for this route to take. This should generally be the longest path from start to finish, and isn't necessarily the "good" end. It will be the path the player begins on when they start a new game on this route.

e.g. `emma_good_end`

branch_list A list of all the other paths this route can branch onto after a plot branch. These are the variables that hold the RouteDay lists you defined earlier. Typically this includes all the bad, normal, and bad relationship ends, etc. This may also be `None` if the route does not branch.

e.g. `[emma_bad_end_1, emma_bad_end_2, emma_normal_end]`

route_history_title The name of this route as it should appear in the History screen e.g. "Emma Route" or "Common Route". The suffix " Route" is automatically appended to this title.

e.g. "Emma"

has_end_title True if this route should have a title labelling the ending type over the last timeline item in the History. If the variable you used for `default_branch` begins with a title like "Good End" (e.g. `emma_good_end = ["Good End", RouteDay("1st", [...])]`, then if `has_end_title` is True, "Good End" will appear just before the last item on `emma_good_end`.

If you don't set this to True/False yourself, the program will automatically set it to True if there is a `branch_list` and False otherwise. An example of a time when this variable should be False even with branching paths is seen in `route_example.rpy` for Casual Route – there is no "Casual Route Good End", since successfully completing Casual Route means the player has branched onto a character route. However, there *is* a Casual Route Bad End inside the Route's branch_list.

e.g. True

history_background Optional. The path to the image that should be used for this route's background in the History screen. This should be 650x120 px for best results. The corners are automatically cropped to fit the button frame.

e.g. "Menu Screens/Main Menu/jaehee-route-bg.webp"

Both `route_setup.rpy` and `route_example.rpy` have definitions of Routes and their associated branching paths so you can get an idea of how routes are defined.

12.3 Expired Timeline Items and Real-Time Mode

In this program, you can switch between two different play styles: real-time, and sequential. Sequential is currently the default. You can toggle real-time from the **Developer** settings button in the chat home screen or on the main menu.

In sequential mode, timeline items unlock sequentially. In other words, once you finish a timeline item like a chatroom, the next one will automatically unlock. Chatrooms and story calls don't expire unless you back out of them (aka hitting the Back arrow while in an active chatroom or hanging up in the middle of a story call), and you can proceed through the story regardless of what the current real-life time is.

In real-time mode, timeline items unlock based on the current time. You also have the option to buy the next 24 hours' worth of timeline items in advance. If an old chatroom or story call has not been viewed before a new one unlocks it will expire, and you will miss any incoming calls that were triggered to occur after the now-expired item (though you can usually call the characters back).

Each chatroom and story call you create should have both a "regular" version and an "expired" version. The expired version is the version the player will play through if the item has expired and they have not bought it back. Generally this means the player will not have the opportunity to participate in the chatroom or make choices, and expired story calls function as though the caller left the player a voice message.

To create an expired timeline item, simply take the name of the regular item's label and add `_expired`. So, if your chatroom has the label

```
label mychat:
```

then the expired chatroom should have the label

```
label mychat_expired:
```

The rest can be filled out as any other chatroom.

Note: Story Mode sections, standalone or attached, do not expire even in real-time mode. However, if its time has already passed, any associated text messages *will* be delivered and associated phone calls will either be missed or require the player to call the character back. The difference is that there is no expired version of story mode sections; they will always play out the same way.

12.3.1 Changing Content If Expired

Besides the `_expired`-style labels for story calls and chatrooms, there is also a special variable called `was_expired` which can be used to modify content depending on whether the associated timeline item was expired or not. For example, in the `after_` label for a chatroom, you can check if the player missed the chat or not and change the resulting content accordingly:

```
label after_my_chatroom():
    if was_expired:
        # The player played the chatroom after it expired and missed it
        compose text s:
            s "[name]..."
            s "I missed u in the chat lol"
            s "What were you up to?"
        label my_chatroom_a_text_msg
    else:
        compose text s:
            s "lolol ur so funny in the chatrooms [name] lolol"
```

(continues on next page)

(continued from previous page)

```
            s "u should come by more often~"
    return
```

If the player did not play through the `my_chatroom` label, 707 will send the first set of text messages. If the player *did* play this chatroom, however, they will instead see the second set of text messages.

You can see an example of this in `tutorial_11_story_call.rpy`.

12.3.2 Backing out vs. real-time expiry

There are two different ways for timeline items to expire: first, items expire if you are playing in real-time and miss playing an item before the next one triggers. Second, items expire if you use the back arrow during a chatroom you haven't seen before or if you hang up in the middle of a story call.

In the first case (expiry due to real-time mode being active), the following will happen:

- You will receive a missed call from any character who was going to call you after the expired item. You can call that character back to receive that conversation.
- Any text messages that would have been delivered after the item will be automatically delivered to your inbox
- Any outgoing calls that were to be made available after the item will be made available

Note that phone calls will "time out" two timeline items after they were set to appear. So, for example, say you have three chatrooms: A, B, and C. Emma is supposed to call you after chatroom A. If chatroom B becomes available before you've seen chatroom A, then chatroom A will expire and you will receive a missed phone call from Emma. You can call Emma back to receive this phone call up until chatroom C becomes available, at which point that phone call will become unavailable and you won't be able to call Emma back to get that conversation anymore.

In the second case, where the player backs out of an active chatroom or hangs up in the middle of a story call and causes it to expire, the following will happen:

- Any incoming calls that would have been triggered after the item are instead turned into outgoing calls, **though the player receives no missed call notification**
- Any text messages that would have been delivered after the item will be automatically delivered to your inbox
- Any outgoing calls that were to be made available after the item will be made available

As you can see, the only real difference is in the first point. Incoming phone conversations will still be available, but you will not receive a missed call notification for it.

12.4 Ending a Route

When the player has reached the end of the route, you need to tell the program which ending screen to show them by setting the `ending` variable. The built-in options are "bad", "normal", and "good" e.g.

```
u "This is the end of the route."
exit chatroom u
$ ending = "normal"
return
```

By setting `$ ending = "normal"`, the program knows that when this timeline item ends, the route is over. The player will be shown the Save & Exit screen (if applicable) and then the ending image will be shown.

You can also use your own image for the end of the route by passing `ending` the full image path or the name of a defined image e.g.

```
image my_custom_ending = "Endings/my_custom_ending.webp"

label end_of_my_route():
    scene black
    show black effect
    enter chatroom u
    u "This is the end of the route."
    exit chatroom u
    $ ending = "my_custom_ending"
    # $ ending = "Endings/my_custom_ending.webp" also works
    return
```

The program will show the desired image before taking the player back to the main menu. If they click "Original Story", they will be prompted to select a route for a new game.

Note: The end of a route is also a good place to set any persistent variables if you're keeping track of a player's route completion to unlock routes/images/extras later.

12.5 Customizing the Route Select Screen

Now that you have your route defined, you can customize the route select screen to include a button to your new route. You will need to create a label for your introduction as well. This can be called whatever you like e.g.

```
label new_years_introduction:
```

Warning: Even if you don't want an introduction and just want to take the player directly to the home screen, you **must still** create an introduction label. This label includes important instructions to set up the route properly.

See *Creating an Introduction* for more information on how to write your introduction and set up the route. The important part here is the name of the label, which will be used for the button on the route select screen.

Mysterious Messenger has a special screen called `custom_route_select_screen`, found inside the file `screens_custom_route_select.rpy`. Inside the **Developer** settings found on the main menu on on the home screen after loading a game, there is an option called **Use custom route select screen**. Checking this option will cause the game to use the `custom_route_select_screen` instead of the default version (which contains buttons for Tutorial Day and Casual Story).

Your custom route select screen should look like the following:

```
screen custom_route_select_screen():
    vbox:
        style_prefix 'route_select' # Remove this if you want your own styles
        button:
            ysize 210 # Set the height of the button
            # The image that goes on the left of the button
            add 'Menu Screens/Main Menu/route_select_tutorial.webp':
                align (0.08, 0.5)
            action Start()
            # The box with text on the right side of the button
            frame:
                text "Tutorial Day"
```

You're free to replace the image "Menu Screens/Main Menu/route_select_tutorial.webp" with your own image or leave it out altogether. The important thing is the action on the button, currently `Start()`. You need to give this the name of the label where it can find your route introduction:

```
screen custom_route_select_screen():
    vbox:
        style_prefix 'route_select' # Remove this if you want your own styles
        button:
            ysize 200 # Set the height of the button
            action Start('new_years_introduction')
            # The box with text on the right side of the button
            frame:
                text "New Year's Story"
```

Note that the action is now `Start('new_years_introduction')`, which will start the game at the label `new_years_introduction`.

12.5.1 Providing Multiple Routes

If you want to provide multiple routes the player can begin on, you will need a button for each route on the route select screen. By default, the buttons on the route select screen are organized inside something called a `vbox`, which will stack its contents on top of each other vertically. This makes it easy to create multiple buttons.

For example, a version of the route select screen which allows the player to choose between a "Casual" or "Deep" story might look like the following:

```
screen custom_route_select_screen():
    vbox:
        style_prefix 'route_select'
        button:
            ysize 210
            add 'Menu Screens/Main Menu/route_select_casual.webp':
                align (0.08, 0.5)
            action Start('casual_story_start')
            frame:
                text "Casual Story"
        button:
            ysize 210
            add 'Menu Screens/Main Menu/route_select_deep.webp':
                align (0.08, 0.5)
            action Start('deep_story_start')
            frame:
                text "Deep Story"
```

Note: The image 'Menu Screens/Main Menu/route_select_deep.webp' does not currently exist in the game; you would need to create it yourself.

You're free to change the button backgrounds, text, spacing, and other styles. The above example provides two buttons. The first takes the player to a label called `casual_story_start`, and the second to a label called `deep_story_start`.

12.5.2 Unlockable Routes

If you want to keep certain routes "locked" until the player has fulfilled a condition of your choosing (e.g. preventing the player from going through Character B's route until after they have completed Character A's route), you need to set up persistent variables to keep track of whether the player has fulfilled your desired condition or not. You can define these variables anywhere you like, though it's a good idea to keep them in a separate `.rpy` file for organization. Since this variable affects the route select screen, you could put it in `screens_custom_route_select.rpy`.

For this example, the program will check whether or not the player has successfully gotten the Good End in Tutorial Day. First, define a variable:

```
default persistent.tutorial_good_end_complete = False
```

`tutorial_good_end_complete` is the name of your created field in the `persistent` object. `persistent` is a special class in Ren'Py that is saved across playthroughs. The above statement first initializes this variable to `False`, and you will set it to `True` after the player has successfully gone through the Good End.

The final label that a player will see if they achieved the good end on Tutorial Day is called `label tutorial_good_end_party` and can be found in `tutorial_8_plot_branches.rpy`. The last lines of that label are currently as follows:

```
$ ending = 'good'
return
```

This ends the route and takes the player back to the menu, so just before the `return` statement you can set your new persistent variable to `True`:

```
$ persistent.tutorial_good_end_complete = True
$ ending = 'good'
return
```

Note: You're not limited to putting your variable check right at the end of a route – you can also put a variable like `$ persistent.confessed_to_emma = True`, which might happen in the middle of a route and unlock a special Valentine's Day event or something else. Remember that `persistent` variables *persist* across multiple playthroughs, so they're best used for events you only want to unlock once.

Note: If you *don't* need the program to remember a variable across different playthroughs (for example, if you want the game to remember you told Emma "I own a cat" so you can later have her mention the cat in a different conversation), you **do not** need `persistent` in front of your variable definition and can simply use a variable like:

```
default owns_cat = False
label emma_route_3_8:
    scene morning
    play music mystic_chat
    em "Hi, [name]!"
    em "I had a question -- do you own a cat?"
    menu:
        "Yes, I have a cat.":
            $ owns_cat = True
            em "Cool!"
        "No, I don't.":
            $ owns_cat = False
            em "Aw, okay."
```

The variable `owns_cat` will be set to False whenever the player begins a new game.

Now you can customize the route select screen based on the value of your persistent variable:

```
vbox:
    style_prefix 'route_select'
    button:
        ysize 210
        add 'Menu Screens/Main Menu/route_select_tutorial.webp':
            align (0.08, 0.5)
        action Start()
        frame:
            text "Tutorial Day"
    # Casual/Jaehee's route is only available to a player who
    # has completed the Good End on Tutorial Day
    button:
        add 'Menu Screens/Main Menu/route_select_casual.webp':
            align (0.08, 0.5)
        frame:
            if persistent.tutorial_good_end_complete:
                text "Casual Story"
            else:
                hbox:
                    align (0.4, 0.5)
                    add 'plot_lock' align (0.5, 0.5)
                    text "Casual Story"
        if persistent.tutorial_good_end_complete:
            action Start('example_casual_start')
        else:
            action CConfirm("This route is locked until you've "
                + "played the Good End on Tutorial Day")
            hover_foreground None
```

There are a lot of parts here, so each will be explained separately.

```
vbox:
    style_prefix 'route_select'
    button:
        ysize 210
        add 'Menu Screens/Main Menu/route_select_tutorial.webp':
            align (0.08, 0.5)
        action Start()
        frame:
            text "Tutorial Day"
```

This part contains a button for Tutorial Day. There are no conditions on it, so it is always available to the player.

```
# Casual/Jaehee's route is only available to a player who
# has completed the Good End on Tutorial Day
button:
    add 'Menu Screens/Main Menu/route_select_casual.webp':
        align (0.08, 0.5)
    frame:
        if persistent.tutorial_good_end_complete:
            text "Casual Story"
        else:
            hbox:
```

(continues on next page)

(continued from previous page)

```
                align (0.4, 0.5)
                add 'plot_lock' align (0.5, 0.5)
                text "Casual Story"
```

This makes a button for Casual Story. However, there is a conditional – `if persistent.tutorial_good_end_complete`, the variable defined earlier. If this variable is True – aka the player has seen the Good End on Tutorial Day – then the button just contains the text "Casual Story". However, if it is false, there is an `hbox`, which organizes its items side-by-side. This hbox shows a "locked" image next to the text "Casual Story" to indicate that Casual Story is currently locked.

```
if persistent.tutorial_good_end_complete:
    action Start('example_casual_start')
else:
    action CConfirm("This route is locked until you've "
        + "played the Good End on Tutorial Day")
    hover_foreground None
```

Finally, if the player hasn't gone through the Good End on Tutorial Day, they shouldn't be able to play Casual Story yet, so there is another conditional statement with the action inside it. If the player has seen the Good End, the game will start at the label `example_casual_start`.

However, if the player *hasn't* seen the Good End, the button will not light up (`hover_foreground None`), and the special action `CConfirm` is used. `CConfirm` shows a confirmation prompt to the user with the given message. In this case, it tells the player that they cannot access this route until they've played the Good End on Tutorial Day.

You could leave the `else` out altogether, which would cause the button to be inactive if the player hasn't played the Good End, but including it helps the player understand why they can't play this route yet and what they have to do to unlock it.

12.6 Creating an Introduction

Now that you have a button on the route select screen that leads to your new route, you may want to have an "introduction" to the route before the player is taken to the home screen. At the very least, your introduction label needs a few lines to finish setting up the route before it takes the player to the home screen.

The introduction functions a bit differently from writing a regular chatroom/phone call/story mode/etc, so it's recommended you have a good grasp of how to create regular timeline items before you work on a custom introduction.

12.6.1 Essential Setup Functions

Everything for the introduction will go inside the label name you gave the `Start()` action. So, for the above examples on the page, the button on the route select screen had an action of `Start('example_casual_start')`, which means the label for the introduction is `example_casual_start`.

For this example, the introduction to the New Year's Route from the *Plot Branches* page will be created. Its definition is below:

```
default new_years_route = Route(
    default_branch=new_years_normal_end,
    branch_list=[new_years_ju, new_years_ja,
        new_years_s, new_years_y, new_years_z],
    route_history_title="New Year's",
    history_background="Menu Screens/Main Menu/new_years_route_bg.webp"
)
```

Assume there is a button with the action `Start("new_years_intro")`:

```
label new_years_intro():

    $ new_route_setup(route=new_years_route, participants=[ja, ju, z])
```

`new_route_setup` tells the program which route to set up for the player. In this case, you are setting up the New Year's Route, which was defined in the variable `new_years_route`. That variable is passed to `new_route_setup`.

There is also a second parameter, `participants`. This is given a list of the characters who should start in the introductory chatroom. This is optional; if omitted, no one begins in the chatroom.

Note: The introduction is assumed to be a chatroom unless otherwise indicated.

Next, you may want to adjust some of the variables for this route. For the New Year's Route, only the characters `ja`, `ju`, `s`, `y`, and `z` should have profiles on the home screen and contacts in the phone's contacts list. By default, `v`, `r`, and `ri` are included in the character list, so you will need to tell the program to not include them.

```
label new_years_intro():

    $ new_route_setup(route=new_years_route, participants=[ja, ju, z])
    $ character_list = [ju, z, s, y, ja, m]
```

This tells the program which characters are included on the home screen.

Warning: If you want to update the character list *after the game has already begun*, you should use the special function `update_character_list` to update it. That looks like this:

```
$ update_character_list([ju, z, s, y, ja, r, m])
```

This will ensure the game is updated to accommodate the new character.

Similarly, you may want to set which characters show up on the Profile screen with an indicator of their collected heart points. Since the New Year's route only has endings for `ja`, `ju`, `s`, `y`, and `z`, you should set up the `heart_point_chars` list to reflect this:

```
label new_years_intro():

    $ new_route_setup(route=new_years_route, participants=[ja, ju, z])
    $ character_list = [ju, z, s, y, ja, m]
    $ heart_point_chars = [ju, z, s, y, ja]
```

Like the `character_list`, by default `v`, `r`, and `ri` are normally included in this list, so you will need to specify if the list should include or exclude some of these characters.

Next, you should decide whether this route will include paraphrased choices or not. For more information, see *Paraphrased Choices*. You will set this up here in the introduction label:

```
label new_years_intro():

    $ new_route_setup(route=new_years_route, participants=[ja, ju, z])
    $ character_list = [ju, z, s, y, ja, m]
    $ heart_point_chars = [ju, z, s, y, ja]
    $ paraphrase_choices = False
```

This should be `False` if you want the main character to directly say the dialogue in the choices, and `True` if you want to always type out what dialogue should be said after a choice.

Finally, you may want to set up a particular profile picture callback function for the main character. See *Profile Picture Callbacks* for more information. If so, set the variable `mc_pfp_callback` to the name of your callback function:

```
label new_years_intro():

    $ new_route_setup(route=new_years_route, participants=[ja, ju, z])
    $ character_list = [ju, z, s, y, ja, m]
    $ heart_point_chars = [ju, z, s, y, ja]
    $ paraphrase_choices = False
    $ mc_pfp_callback = bonus_pfp_dialogue
```

Now that the main variables for your introduction are set up, you can begin writing the actual introduction or jump directly to the home screen without further ado.

If your introduction is a chatroom, you can set up the background with a `scene` statement and write the chatroom as normal, including heart icons, choice menus, and more. The introduction should end with a `return` statement. If you would like to include a phone call or story mode section in your introduction, this is handled differently (see below).

12.6.2 Leaving out the Introduction

While you *must* include the line `$ new_route_setup(route=your_route)` where `your_route` is replaced with the variable name of your actual route, you do not actually need an introduction in order to begin your route. Simply include the line `jump skip_intro_setup` at the end of your introduction label:

```
label new_years_intro():

    $ new_route_setup(route=new_years_route, participants=[ja, ju, z])
    $ character_list = [ju, z, s, y, ja, m]
    $ heart_point_chars = [ju, z, s, y, ja]
    $ paraphrase_choices = False
    $ mc_pfp_callback = bonus_pfp_dialogue

    jump skip_intro_setup
```

This will set up the necessary variables and then jump immediately to the home screen to begin the route.

12.6.3 Additional Introduction Features

For the introduction of your route, you may want to include a combination of features such as phone calls, chatrooms, and story mode. In order to switch between these features, you must use special calls and functions to set up the screens properly.

Including an Incoming Call

To include an incoming call, use the line:

```
call new_incoming_call(u)
```

where `u` is the ChatCharacter variable of the character who is phoning the player. This will cause that character to phone the player. The player is unable to "miss" this call and must answer it to proceed. You can then write the phone dialogue as you normally would.

If you would then like to switch from the phone call to a chatroom or story mode, you must include special calls to set up the appropriate variables.

Including a Story Mode

If you would like to begin with a story mode section, or switch to story mode after a phone call, you must use the line:

```
call vn_begin()
```

to set up the appropriate variables. You can then write dialogue and show character portraits the way you normally would during a story mode section.

If you'd like to switch to story mode in the middle of a chatroom, see instead *Including a Story Mode During a Chatroom*.

Including a Chatroom with Other Features

If you begin your introduction with a phone call or a story mode and want to return to and/or end on a chatroom, you must include the line:

```
call chat_begin('morning')
```

to set up the appropriate chatroom variables. `'morning'` should be the name of the background you are setting up.

12.6.4 Introduction Example

For example, an introduction which combines several elements into one introduction may look as follows:

```
label my_new_route_intro():

    $ new_route_setup(route=my_new_route)
    $ character_list = [ju, z, s, y, ja, m]
    $ heart_point_chars = [ju, z, s, y, ja]

    # First, this should begin with a Story Mode section
    call vn_begin()
    scene bg hallway with fade
```

(continues on next page)

(continued from previous page)

```
    pause
    "(My phone is ringing)"
    menu (paraphrased=True):
        extend ''
        "Answer it.":
            call new_incoming_call(u)
        "Ignore it.":
            "(The phone goes silent)"
            show saeran mask
            u "Oh? You're not going to answer that?"
            u "Hmm, that's too bad."
            u happy "Well, nothing to be had for it. You need to come with me."
            # This will end the route with the 'Bad Ending' screen
            $ ending = 'bad'
            return

    # This is now the beginning of the phone call
    u "Haha, I can't believe you picked up."
    u "Um, I need you to do a favour for me. Can you do that?"

    menu:
        extend ''
        "Sure, I guess.":
            u "Great! That's so great. Alright, I'll send you some instructions over␣
↪the messenger."
            u "Don't worry if you see some flashy effects or anything."
            u "That just means you're getting access."

    # Show a chatroom
    call chat_begin('hack')
    show hack effect
    scene hack
    enter chatroom u
    u "Hello~"
    u "Thanks for coming."
    u "I need a favour from you, like I said in the phone call."
    u "You're in an apartment hallway, right?"
    u "There's a door there that says \"RFA\" on the handle."
    u "Type in the password 58439. Okay? It should open."

    # Jump to story mode during the chatroom
    call vn_during_chat("unlock_apt_door")
    # Returned to the chatroom
    m "It's open."
    u "Yay!"
    u "All right, see you soon~"
    return

# This is the label the program jumps to for the story mode
# that is in the middle of a chatroom for the introduction.
label unlock_apt_door():
    scene bg hallway with fade
    pause
    menu (paraphrased=True):
        "(Approach the door)":
            pass
    scene bg rika door closed with fade
```

(continues on next page)

(continued from previous page)

```
    pause
    menu (paraphrased=True):
        "(Type in the password)":
            pass
    scene bg rika_door_open with dissolve
    pause
    return # Takes the player back to the chatroom
```

CHAPTER

THIRTEEN

PLOT BRANCHES

Note: Example files to look at:

- route_setup.rpy
- route_example.rpy
- tutorial_8_plot_branches.rpy

A brief overview of the steps required (more detail below):

1. In your list of RouteDay objects, add `plot_branch=PlotBranch(False)` or `plot_branch=PlotBranch(True)` as a parameter to the timeline item you want the plot branch to appear on.
2. Define another list of `RouteDay` objects for the path the player will branch onto. This should begin on the same RouteDay as the plot branch.
3. After the timeline item with the plot branch, create another label with the item label + the suffix `_branch` e.g. `label casual_day_4_10_branch`
4. Fill out your criteria for branching. You can then either write `$ continue_route()` to have the player continue down the current path, or use `$ merge_routes(my_route_normal_end)` where `my_route_normal_end` is the path you defined in Step 2.
5. `return` at the end of the plot branch label.

To begin, you need a list of `RouteDay` objects for every path you want the user to be able to branch onto. For more on defining routes, see *Setting up a Route*. If you want this branch to have a title, the first entry in the list should be a string like "Emma Good End".

This example will demonstrate a New Year's Eve route with branching paths for the different characters. The New Year's Eve Route is shown below:

```
default new_years_route = Route(
    default_branch=new_years_normal_end,
    branch_list=[new_years_ju, new_years_ja,
        new_years_s, new_years_y, new_years_z],
    route_history_title="New Year's",
    history_background="Menu Screens/Main Menu/new_years_route_bg.webp"
)
```

This example will focus on the definitions of three of the paths, the default branch `new_years_normal_end`, and the branches `new_years_ju`, and `new_years_z`.

The default branch of the route is defined as below.

```
default new_years_normal_end = [ "Normal End",
    RouteDay("31st",
        [ChatRoom("The end of another year...", None, '00:30', [ja, ju, s, y, z]), # 1
         ChatRoom("Resolutions", None, '06:43', [ja]), # 2
         StoryMode("At the office", None, '09:44', ju), # 3
         ChatRoom("A celebration?", None, '12:12', [y, z]), # 4
         StoryCall("Endings and beginnings", None, '15:29', s), # 5
         ChatRoom("Cats and chats", None, '18:13', [s, ju], plot_
↪branch=PlotBranch(True)), # 6
         ChatRoom("Too busy...", None, '17:50') # 7
        ], auto_label='new_years_t'
    )
]
```

The plot branch will appear after the chatroom titled “Cats and chats”. A PlotBranch object takes one field:

branch_story_mode If True, if the player proceeds through this plot branch and remains on this route, an attached Story Mode should appear after this item’s chatroom.

This is *only* applicable if the associated timeline item is a ChatRoom, as it is the only timeline item which may have an attached story mode.

By default, this field is False.

Because this plot branch has a `branch_story_mode`, there must be an associated Story Mode with the “Cats and chats” chatroom. The automatic labelling means that “Cats and chats” is found at `new_years_t6`, so it should have a corresponding Story Mode at a label like `new_years_t6_vn` (optionally with a character’s file_id as the suffix to associate this Story Mode with that character). This Story Mode will only appear if the player proceeds through the plot branch and remains on the Normal End.

Next, you must define the other paths for the route, in this case, `new_years_ju` and `new_years_z`:

```
default new_years_ju = [ "Jumin New Year's End",
    RouteDay("31st",
        [StoryMode("Visiting the penthouse", 'jumin_ny_end', '18:00', ju)],
        save_img='ju', day_icon='ju')
]

default new_years_z = [ "Zen New Year's End",
    RouteDay("31st",
        [StoryMode("A special visit", 'zen_ny_end', "18:00", z)],
        save_img='z', day_icon='z',
        branch_vn=BranchStoryMode('zen_ny_branch_vn', z)
    )
]
```

Note that the RouteDay is named the same as the RouteDay on the day with the plot branch – namely, “31st”. This tells the program that the timeline items on this RouteDay should be merged onto “31st Day” on the main route.

For the `new_year_z` definition, there is a special field called `branch_vn` on the RouteDay:

branch_vn If given, an attached StoryMode will be created with the provided label (and optionally the provided character image). If the player merges onto this path after a plot branch, the chatroom which had the plot branch will have this Story Mode attached to it.

e.g. BranchStoryMode(“example_jaehee_be1_vn”)

A BranchStoryMode is a special convenience function to create an attached StoryMode for a plot branch. It has two fields:

vn_label The label that this StoryMode is found at.

e.g. “example_jaehee_be2_vn”

who Optional. The character whose image should be shown on the icon for this StoryMode in the timeline.

e.g. ja

13.1 Determining Which Path to Branch To

Now that the paths are all set up, you need to create a special label which will tell the program which path to branch onto when the player proceeds through the plot branch. This is the name of the label after which the plot branch occurs + `_branch`. So, for the New Year's route, the automatic labelling puts the plot branch at the label `new_years_t6`. That means you will create the label:

```
label new_years_t6_branch:
```

Inside this label, you need to create a conditional statement to tell the program how to proceed. There are many examples in `tutorial_8_plot_branches.rpy` as well as `route_example.rpy`.

The two main ways to branch are `continue_route` and `merge_routes`. If the player should continue on the same path they're currently on (so, for the example, if they should continue on to the Normal End), then use `$ continue_route()` under your conditional statement.

On the other hand, if the player should be moved onto a separate route, then you need to use `$ merge_routes(new_year_z)` where `new_year_z` is the path the player should branch onto.

For the New Year's route, the branch label may look like the following:

```
label new_years_t6_branch:
    # First, check if z or ju have more heart points
    if z.heart_points >= 10 and z.heart_points > ju.heart_points:
        $ merge_routes(new_year_z)
    elif ju.heart_points >= 10:
        $ merge_routes(new_year_ju)
    # Otherwise, the player doesn't have enough heart points with
    # either character, so they get the Normal End
    else:
        $ continue_route()

    return
```

This checks first if the player has at least 10 heart points with Zen, and then also if they have more points with Zen than with Jumin. If so, the player get's Zen's route.

Otherwise (`elif`), the program checks if the player has at least 10 heart points with Jumin. If so, they get Jumin's route. Because of how the previous conditional was set up, if the player has the same number of points with Zen *and* Jumin, they will get Jumin's ending.

Finally, if the player doesn't have enough points with either character, they will get the Normal End (by continuing down the current path).

Tip: If the plot branch label returns without running either `continue_route` or `merge_routes`, `continue_route` will be run automatically and the player will continue down the current route.

13.2 Branching on the Party

You may also want to cause a branch to occur when the player clicks to enter the party, for example, to check how many guests are attending so that the player can branch onto either the Good End or the Normal End, as appropriate. This does not require any additional code in the route definition. So long as your timeline item is marked as the party – either via using `TheParty` to define it, using something like `StoryMode("The Party", "my_party_label", "12:00", party=True)`, or calling your attached Story Mode something like `my_timeline_label_party` (with the suffix `_party`) – then the program will automatically search for a branch label.

The branch label is searched for with the suffix `_branch`, so, if your party is found at the label `emma_route_good_party`, then the program will execute the label `emma_route_good_party_branch` before the player plays the party.

You can use this branch label as you would with any other branch label; however, if you `merge_routes`, the path which is being merged onto the main route will need to have a party on it that will be swapped out for the current party. e.g.

```
default emma_normal_end = [ "Normal End",
    RouteDay("Final", [TheParty('emma_route_normal_end', '12:00')])
]
```

When the player clicks on the party, the program will check if there is a branch label. If so, it will execute the code in the branch label, which may involve merging onto a new path. If the player is merged onto a new path, they will go to the party label found on the merged route.

13.3 Plot Branch Examples

13.3.1 Checking for Participation

You can check the percentage of timeline items the player has actively participated in (aka that haven't expired) with a special function:

```
if participated_percentage(1, 4) > 32:
```

This checks if the percentage of chatrooms the player has participated in across days 1-4 is over 32%. Note that `4` in this case does not necessarily correspond to "4th Day"; since you may name your days whatever you please. If your route is defined like:

```
default my_route = [
    RouteDay("25th"),
    RouteDay("31st"),
    RouteDay("45th"),
    RouteDay("62nd")
]
```

Then `participated_percentage(1, 4)` would be checking for the participation on the "25th" (1) day through to the "62nd" (4) day. This number is *inclusive*; aka `participated_percentage(1, 4)` will total the number of timeline items found on the 25th/31st/45th/62nd days vs the number of timeline items participated in on those days and return that fraction as a whole number percentage (rounded down).

If you want to check for participation on a single day, you must pass it the same number for both the "first" and "last" days. For example, if you wanted to check for participation on just the "31st" day (from the above example), since it is the 2nd item in the list you would check:

```
if participated_percentage(2, 2) > 32:
```

If you want to check the participation from a particular day all the way to the end of the route, you can leave the second field blank e.g.

```
if participated_percentage(5) > 32:
```

Note: `participated_percentage` only returns the proportion of participated timeline items vs *available* timeline items for a particular time range.

So, if you have 12 timeline items on the "6th" day and the plot branch is after the second item, if you calculate participation percentage on the "6th" day, the program will ignore the 10 timeline items which aren't available. That means that if the player played the item just before the plot branch but missed the first item, they will have a 50% participation percentage that day since only 2/12 items were available and the player participated in 1/2 (50%) of those items.

13.3.2 Checking for Guest Attendance

You may also want to check how many guests are attending the party to determine which path to branch the player onto. The function `attending_guests()` returns the number of guests who will be attending the party at the time the function is called:

```
if attending_guests() >= 10:
    $ merge_routes(my_route_good_end)
else:
    $ merge_routes(my_route_normal_end)
```

This is best used for the plot branch when clicking the party (see *Branching on the Party*). You can also check if individual guests are attending the party; see *Checking in-game if a Guest Will Attend*.

13.3.3 Comparing Heart Points:

Typically, the easiest way to compare heart points between characters is to check with:

```
if s.heart_points > ju.heart_points:
    $ merge_routes(seven_route)
else:
    $ merge_routes(ju_route)
```

However, if you have a complex plot branch in which you want to compare several characters' heart points, you may find the following functions useful:

```
$ sorted_heart_points = sorted(heart_point_chars,
    key=lambda c: c.good_heart, reverse=True)
# sorted_heart_points now contains a list of the heart_point_chars, sorted
# in descending order from most heart points to least

# who is now equal to the character who has the most heart points
$ who = sorted_heart_points[0]

# First, check if the character with the most heart points has at least
# 10 total heart points
```

(continues on next page)

(continued from previous page)

```
if who.heart_points < 10:
    # They don't have 10 heart points; continue down the Normal End
    $ continue_route()
elif who == ja:
    $ merge_routes(new_year_ja)
elif who == ju:
    $ merge_routes(new_year_ju)
elif who == s:
    $ merge_routes(new_year_s)
elif who == y:
    $ merge_routes(new_year_y)
elif who == z:
    $ merge_routes(new_year_z)

return
```

The sorted list saves you from writing out lengthy comparison statements such as:

```
if ja.heart_points > 10 and ja.heart_points > ju.heart_points and ja.heart_points > s.
↪heart_points and...
```

CHAPTER

FOURTEEN

CREATING CHARACTERS

While Mysterious Messenger does come with several characters already defined, you may want to define your own characters to participate in chatrooms or phone the player. There are a few definitions and several images you will need to set up in order for your character to work within the program.

The characters that come with the program by default are:

- ja (Jaehee Kang)
- ju (Jumin Han)
- m (The main character (MC)/"you")
- r (Ray)
- ri (Rika)
- s (707)
- sa (Saeran)
- u (Unknown)
- v (V)
- va (Vanderwood)
- y (Yoosung)
- z (ZEN)

There are also a few characters who are defined only for use in Story Mode:

- sarah_vn (Sarah)
- chief_vn (Chief Han)

On this page, the examples will show how to add a character named Emma to the program.

14.1 Checklist for a New Character

There are many definitions and images you need to set up in order for a new character to work in the program. To ensure you don't miss any steps, this page outlines what the necessary definitions and images are.

Any tasks prefaced with **(Optional)** are optional. Read the description to determine if you want this feature for your new character.

Items prefaced with **(May be required)** are dependent on whether or not you have previously implemented an **(Optional)** task. For example, if you have added a character to the `character_list` variable, you **must** define a phone contact image, but if the character is not in `character_list` then they won't need this image.

- Define a ChatCharacter object in `character_definitions.rpy` under the heading **Chatroom Characters** (*Adding a New Character to Chatrooms*)
  - This step is NOT required if this character will never appear in a chatroom
- **(Optional)** Add your character to the `character_list` in `character_definitions.rpy` if you want their profile to appear on the home screen and allow the player to call them. (*Showing Your Character on the Home Screen*)
- **(Optional)** Add your character to the `heart_point_chars` list in `character_definitions.rpy` if you want the player to see how many heart points they have earned with this character. (*Showing Your Character on the Home Screen*)
- **(May be required)** Define a `greet` image for your character. This is **required** if you have included the character in `heart_point_chars` (see above) AND/OR if you want them to have greetings on the main menu. (*Greeting Images*)
- Define a Character object in `character_definitions.rpy` under the heading **Story Mode**. (*Adding a New Character to Story Mode*)
  - This step is NOT required if this character will never appear in a Story Mode section, OR if you've already defined a ChatCharacter object for them.
- Either: **1)** in the definition for your Story Mode Character or ChatCharacter, include the `voice_tag` argument (`voice_tag="em_voice"` where `em` is the character's file_id), OR **2)** add their ChatCharacter variable to the `novoice_chars` list in `character_definitions.rpy`. (*Note on voiced characters*)
- **(Optional)** Define a `layeredimage` for your character if you want to display their image during Story Mode (VN) sections. (*Declaring a LayeredImage for a New Character*)
- **(May be required)** Define a Story Mode timeline image for your character if you want to display a Story Mode associated with them on the timeline screen. (*Story Mode Timeline Images*)
- Define a Character object in `character_definitions.rpy` under the heading **Phone Call Characters**. (*Adding a New Character to Phone Calls*)
  - This step is NOT required if this character will never appear in a phone call, OR if you've already defined a ChatCharacter object for them.
- **(May be required)** Define a phone contact image for your new character. **Required** if you have added them to the `character_list` variable. (*Adding a Phone Contact Image*)
- **(Optional)** Define a CG album for your character. Requires a `cg_label`, `album_cover`, and two album variables (one persistent and one regular). Add the character's file_id to the `all_albums` list. (*Adding a CG Album*)
- **(Optional)** Add a spaceship thoughts image for your new character. (*Giving a New Character Spaceship Thoughts*)
- **(Optional)** Add a day select image for your new character. (*Adding a Day Select Image*)
- **(Optional)** Add a Save & Load image for your new character. (*Adding a Save/Load Image*)
- **(Optional)** Add bonus profile pictures for your new character. (*Bonus Profile Pictures*)

14.2 Adding a New Character to Chatrooms

All characters that currently exist in the program are defined in `character_definitions.rpy`. Open that file and scroll down to the header **Chatroom Characters**.

As mentioned, these examples will show how to add a character named Emma to the program. First, you need to give Emma a ChatCharacter object so she can speak in chatrooms. A definition for Emma might look like the following:

```
default em = ChatCharacter(
    name="Emma",
    file_id="em",
    prof_pic="Profile Pics/Emma/emma1.webp",
    heart_color="#F995F1",
    participant_pic="Profile Pics/em_chat.webp",
    cover_pic="Cover Photos/emma_cover.png",
    status="Emma's Status",
    bubble_color="#FFDDFC",
    glow_color="#D856CD",
    homepage_pic="Profile Pics/main_profile_emma.webp",
    vn_name="Emma",
    window_color="#B400A4",
    image="emma",
    voice_tag="em_voice"
)
```

Usually the actual variable name – in this case, `em` – is short. It is recommended that this be two characters long or more; usually the first two letters of the character's name. The program already uses `ja`, `ju`, `m`, `r`, `ri`, `s`, `sa`, `u`, `v`, `y`, and `z`.

> **Warning:** New ChatCharacter variables should be at least two letters long to avoid conflicts with engine code.

Each of those fields is explained below:

name A string. This is the name of the character as it should appear above their chatroom messages and in some other locations, like in a text message conversation with them.

e.g. `"Emma"`

file_id A string. This is used for many things internally to associate images and other variables with the character. For example, if a character's file_id is `"em"`, then the program will look for incoming phone calls from this character with the suffix "_em" e.g. `my_chatroom_incoming_em`. This must be the string version of what you called the ChatCharacter variable, as it is often used to find the ChatCharacter object itself in some circumstances.

e.g. `"em"`

prof_pic The profile picture for this character. Usually it is a string with the file path of the image.

> **Tip:** Profile pictures should be 110x110 pixels large. A larger version, up to 314x314 pixels, can also be provided with the same file name + "-b" (for 'big').
>
> e.g. If your profile picture is `"ja-default.webp"`, then the program will look for a larger version with the filename `"ja-default-b.webp"`.

e.g. "Profile Pics/Emma/emma1.webp"

heart_color A string containing hex colour code of the heart icon that appears when awarding the player a heart point for this character. It is not case-sensitive.

e.g. "#F995F1"

The remaining fields are optional or semi-optional depending on where this character will appear and what other variables or images are defined.

participant_pic Optional. The file path to the image that should be used on the timeline screen to indicate that the character was present in a chatroom. If not provided, their default profile picture is used.

e.g. "Profile Pics/em_chat.webp"

The following two fields either must be given a colour, or you will need to place a special image file inside the game's `images/Bubble` folder to use as the background for the character's dialogue bubbles.

bubble_color Optional; however, if this is not defined **you must provide an image** in `game/images/Bubble/` called `em-Bubble.webp` if the character's file_id is `em`.

Otherwise, `bubble_color` should be a string containing a colour code. The character's regular speech bubble will have this colour as its background. Unless the `who_color` parameter is also supplied, this will also be the colour of the character's name during Story Mode.

e.g. "#FFDDFC"

glow_color Same as bubble_color, however, if `glow_color` is not provided the game will look for an image in `game/images/Bubble/em-Glow.webp` if the character's file_id is `em`.

e.g. "#D856CD"

If this character will appear on the home screen with a clickable profile, you should define the following fields:

cover_pic The file path to the image used for this character's cover photo on their profile screen.

e.g. "Cover Photos/emma_cover.webp"

status A string containing the character's current status.

e.g. "I ate a sandwich today."

homepage_pic The file path to the image that should be displayed on the home screen. The player clicks this image to view the character's profile. This should generally be a headshot of the character with a transparent background. If not given, the character's default profile picture will be used.

e.g. "Profile Pics/main_profile_emma.webp"

If the character will appear in story mode sections, you should define the following fields:

window_color A string containing the colour code that will be used for the dialogue window of this character during Story Mode. Replaces the `window_background` property. If not provided, defaults to a grey colour.

e.g. "#C8954D"

vn_name Optional; if not provided, the character will use the `name` field. However, if the chatroom name is more of a nickname (e.g. "Emma"), then you may want to provide this field so that the name appears as "Emma" during story mode.

e.g. "Emma"

who_color Optional. The colour of the character's name during story mode. If the field `bubble_color` is provided (the colour of the character's default speech bubbles), it will be used for the `who_color` field unless an alternative is provided. If neither `bubble_color` or `who_color` is provided, it is "#FFF5CA" by default.

e.g. "#FFDDFC"

image Optional. Ren'Py will apply this tag to images if you include attribute tags during a character's dialogue (See Say with Image Attributes).

e.g. "emma"

voice_tag Optional. If this character will speak in phone calls or during story mode, then this is the tag associated with them when they speak. Including this allows players to switch voice acting for this character on and off. This should be the character's file_id + "_voice". Otherwise, by default this character's voice will fall under the "other_voice" tag in the Sound preferences.

e.g. "em_voice"

14.2.1 Note on voiced characters

Warning: If your new character does not have their own voice tag and should not include their own voice toggle in the Settings, then you must also include them in the special `novoice_chars` list found in `character_definitions.rpy` e.g.

```
default novoice_chars = [u, sa, m, em]
```

This will prevent the program from generating a voice toggle button for them.

You can also fine-tune the properties of the characters used for phone calls and story mode (VN) sections via the following fields. However, typically you will not need to manually define these:

phone_char A Character object defined for this character for phone calls.

e.g. `em_phone`

vn_char A Character object defined for this character for Story Mode.

e.g. `em_vn`

Finally, `ChatCharacter` has some additional optional fields that are either currently unused or not necessary to set manually during definition time:

voicemail A string with the name of the label to jump to for this character's voicemail.

e.g. "voicemail_1"

right_msgr False by default, but True if this character should appear on the right side of the messenger. Typically this variable is False for everyone but the main character.

emote_list A list of the "{image=...}" lines corresponding to all emojis associated with this character. Used in the chatroom creator, but has no effect in-game. See `emoji_definitions.rpy` for the values of the currently-defined characters' `emote_list` parameters.

pronunciation_help A screen reader-friendly spelling of the character's name for use with self-voicing.

e.g. `pronunciation_help` for 707 is `"seven-oh-seven"`

If this character will be used during story mode, you may also provide additional properties for the story mode character, such as `what_prefix` to further customize how their dialogue is displayed. See Ren'Py's documentation for more information.

14.2.2 Showing Your Character on the Home Screen

Finally, beneath all the ChatCharacter definitions in `character_definitions.rpy`, there are two lists. The first of these is

```
default character_list = [ju, z, s, y, ja, v, m, r, ri]
```

If you want Emma to show up on the home screen with a clickable profile, or to appear as a contact in the player's phone contacts, you must add her to this list e.g.

```
default character_list = [ju, z, s, y, ja, v, m, r, ri, em]
```

Note: Note that you can also set this on a per-route basis; see *Setting up a Route* for more information. If you change the `character_list` variable during a route to add a new character, you need to use the special function `update_character_list` to update it. That looks like this:

```
$ update_character_list([ju, z, s, y, ja, r, m])
```

This will ensure the game is updated to accommodate the new character.

The second list is

```
default heart_point_chars = [ c for c in character_list if not c.right_msgr ]
```

This list contains all the characters in `character_list` unless they have the property `right_msgr`, which generally means it includes everyone in the `character_list` unless they are the MC. An equivalent definition would look like:

```
default heart_point_chars = [ ju, z, s, y, ja, v, r, ri ]
```

Characters in `heart_point_chars` will appear on the player's Profile screen with an indicator of how many points the player has with them. If you want to add Emma to this list, then you need to define an image called `greet em` since `em` is Emma's file_id.

Greeting Images

You can find the existing characters' images in `variables_editable.rpy` under the heading **GREETING IMAGES**. Greeting images are approximately 121x107 px up to 143x127px.

A greeting image for Emma might look like:

```
image greet em = "Menu Screens/Main Menu/em_greeting.webp"
```

This image is also used on the character's profile picture screen to indicate how many heart points the player has earned with this character and can spend on bonus profile pictures (if not provided, the character's homepage_pic is used instead).

When all is said and done, you should now be able to write dialogue for Emma anywhere in the program. For example, some chatroom dialogue might look like:

```
em "How are you?"
msg em "It's a lovely morning~" glow
```

14.3 Adding a New Character to Story Mode

All characters that currently exist in the program are defined in `character_definitions.rpy`. Open that file and scroll down to the header **Story Mode**.

If you have already defined a ChatCharacter for your new character, they will automatically be able to speak in story mode and phone calls without needing to define anything else. However, if you have a character who will *only* speak during story mode (e.g. a character like Echo Girl), you will define them in this way:

```
define em_vn = Character("Emma",
    kind=vn_character,
    who_color="#FFDDFC",
    image="emma",
    window_color="#b7b7b7",
    voice_tag="em_voice"
)
```

Tip: You can also define a character for Story Mode separately from a character's ChatCharacter definition, and pass it the character variable in the `vn_char` field. This allows you to have more control over the different properties of the story mode character. However, for most purposes, the character that the program automatically defines during the ChatCharacter setup will be sufficient.

The definition fields are explained below.

name This is the name of the character as it should appear in the dialogue box and history log during story mode.

e.g. "Emma"

kind In order to simplify Character definitions, this field allows a Character object to "inherit" from an existing Character. In this case, using `kind=vn_character` sets up many of the properties that are consistent across all Characters for story mode.

e.g. vn_character

who_color Optional. The colour of the character's name during story mode. By default, it is "#FFF5CA". The existing characters use the background colour of their chatroom speech bubbles as their `who_color`.

e.g. "#FFDDFC"

image Optional. Ren'Py will apply this tag to images if you include attribute tags during a character's dialogue (See Say with Image Attributes).

e.g. "emma"

window_color A string containing the colour code that will be used for the dialogue window of this character during Story Mode. Replaces the `window_background` property. If not provided, defaults to a grey colour.

e.g. "#C8954D"

voice_tag Optional. If this character will speak in phone calls or during story mode, then this is the tag associated with them when they speak. Including this allows players to switch voice acting for this character on and off. This should be the character's file_id + "_voice". Otherwise, by default this character's voice will fall under the "other_voice" tag in the Sound preferences.

e.g. "em_voice"

Warning: See *Note on voiced characters* if your new character should not have their own voice toggle on the Settings screen.

You can then write dialogue during story mode like:

```
em_vn "Brr, it's cold outside!"
```

14.3.1 Declaring a LayeredImage for a New Character

At the bottom of `character_definitions.rpy` are all the layered image definitions for the existing characters. A layered image allows you to show the characters on-screen during story mode and easily change their expressions. In order to show Emma on-screen, you will need to define a `layeredimage emma` here. As this is unchanged from the usual way of defining a layered image, you can look into Ren'Py's layered image documentation for more information on how to declare a layered image.

If possible, expressions should be separate from the character's body, and accessories such as glasses and masks should be separate from facial expressions. If a character has multiple positions, such as a side and front view, then those should be in two separate layeredimage definitions (see the definitions for characters like Jumin and Zen for how this is done).

14.3.2 Story Mode Timeline Images

If you would also like to define a timeline image for Story Mode that is associated with your new character, in `variables_editable.rpy` under the heading **STORY MODE/VN IMAGES** you can see a list of existing images for story mode. This should be `vn_` + your character's file_id, so for Emma it would look like:

```
image vn_em = "Menu Screens/Day Select/vn_em.webp"
```

The image should be 555x126 px to fit inside the story mode frame on the timeline. You will then be able to use the suffix `_vn_em` on a label to associate a story mode with Emma. See *Writing a Story Mode* for more information on writing a Story Mode section.

Note that this is only applicable to characters with a ChatCharacter definition (and therefore with a `file_id`). Characters who only speak during story mode do not automatically have this defined and cannot have their own automatic story mode timeline images.

14.4 Adding a New Character to Phone Calls

All characters that currently exist in the program are defined in `character_definitions.rpy`. Open that file and find the header **Phone Call Characters** at the top.

If you have already defined a ChatCharacter object for your new character, there is no need to define a separate phone character (unless you want more control over the phone character definition). It is automatically defined for you already.

However, if you want to define a character who only speaks during phone calls, they will need to have a Character object defined for them. A definition for Emma may look like the following:

```
default em_phone = Character("Emma",
    kind=phone_character,
    voice_tag="em_voice"
)
```

Tip: You can also define a character for phone calls separately from a character's ChatCharacter definition, and pass it the character variable in the `phone_char` field. This allows you to have more control over the different properties

of the phone call character. However, for most purposes, the character that the program automatically defines during the ChatCharacter setup will be sufficient.

The definition fields are explained below.

name This is the name of the character. This is not currently shown during a phone call but is kept for history purposes and to display the caller ID.

e.g. "Emma"

kind In order to simplify Character definitions, this field allows a Character object to "inherit" from an existing Character. In this case, using `kind=phone_character` sets up many of the properties that are consistent across all Characters for phone calls.

e.g. phone_character

voice_tag Optional. If this character will speak in phone calls or during story mode, then this is the tag associated with them when they speak. Including this allows players to switch voice acting for this character on and off. This should be the character's file_id + "_voice". Otherwise, by default this character's voice will fall under the "other_voice" tag in the Sound preferences.

e.g. "em_voice"

Warning: See *Note on voiced characters* if your new character should not have their own voice toggle on the Settings screen.

You can then write dialogue during a phone call like:

```
em_phone "How are you, [name]?"
```

Note: You can also make "Phone-only" characters who can call the player and have a profile picture and can speak dialogue during phone calls, but aren't part of the player's contact list and won't appear fully in the game. For more information, see *Phone-Only Characters*.

14.4.1 Adding a Phone Contact Image

In order for your new character to appear as a contact in the Contacts tab of the phone, you will need to define a contact image for them.

In `variables_editable.rpy` under the header **PHONE CONTACT IMAGES** you will see several images already defined. This is the file_id of the character + `_contact`. So, for Emma, it may look like:

```
image em_contact = "Phone Calls/call_contact_emma.webp"
```

This image is 188x188 and round.

14.5 Giving a New Character Spaceship Thoughts

In the file `variables_editable.rpy` under the header **SPACESHIP THOUGHT IMAGES** you will see several images defined for the various characters. To give Emma a spaceship thought, you must add an image here with her file_id:

```
image em_spacethought = "Menu Screens/Spaceship/em_spacethought.webp"
```

The important thing is to call the image `em_spacethought` where `em` is the character's file_id. This image should be 651x374 px and is typically rounded and slightly transparent. It will be shown behind the thought the character has when the spaceship icon is clicked on the home screen.

You can then give Emma a "SpaceThought" by updating the `space_thoughts` list, which can be found in `variables_editable.rpy` under the header **SPACESHIP/CHIP BAG VARIABLES**.

The variable `space_thoughts` is a special `RandomBag` object, which will shuffle the thoughts to display to the player in a random order. It takes one parameter, a list of items to be shuffled.

A SpaceThought takes two parameters. The first is the character's ChatCharacter object, and the second is the contents of the thought itself, as a string:

```
default space_thoughts = RandomBag( [
    SpaceThought(ja, "I should have broken these shoes in better before wearing them␣
↪to work today."),
    SpaceThought(ju, "I wonder how Elizabeth the 3rd is doing at home."),
    SpaceThought(s, "Maybe I should Noogle how to get chip crumbs out of my keyboard..
↪."),
    SpaceThought(y, "Yes! Chocolate milk is on sale!"),
    SpaceThought(z, "Maybe I should learn how to braid my hair..."),
    SpaceThought(r, "I can't believe I accidentally used one of the other Believer's␣
↪shampoo. My hair smells like lemons."),
    SpaceThought(ri, "Hmm... the soup tastes different today."),
    SpaceThought(sa, "So... sleepy..."),
    SpaceThought(v, "The weather is so very lovely today. Maybe I'll go for a walk."),
    SpaceThought(em, "I know I should take my dog for a walk but I'm so tired...")
    ] )
```

You can see that `SpaceThought(em, "I know I should take my dog for a walk but I'm so tired...")` was added to the end of the list. Now when the player clicks on the spaceship, Emma's space thought has a chance of appearing. For more on adding or changing the spaceship thoughts during a route, see *Spaceship Thoughts*.

14.6 Adding a Day Select Image

If you want a particular image to be used on the timeline screen when choosing a day, you can define it in `variables_editable.rpy` under the header **DAY SELECT IMAGES**. There are no restrictions on what this can be called other than it must begin with `day_` (i.e. there is no need to use a character's file_id). You will use this in the `day_icon` field when defining a route.

So, if you wanted to define a particular image to use on days corresponding with Emma's route, your definition may look like:

```
image day_em = "Menu Screens/Day Select/day_em.webp"
```

Then, when defining your route, you could have the `save_img` field like:

```
default emma_good_end = ["Good End",
    RouteDay("5th",
        [ChatRoom("Bright and early...", 'emma_d5_c1', '00:33', [em])],
        day_icon='em'
    )
]
```

Note that the string given to `day_icon` is "em" because the image was defined as `day_em`, so you drop the `day_` prefix and pass the rest of the image name (in this case, "em").

14.7 Adding a Save/Load Image

If you want a particular image to be used for a save file's icon, you can define it in `variables_editable.rpy` under the header **SAVE & LOAD IMAGES**. There are no restrictions on what this can be called other than it must begin with `save_` (i.e. there is no need to use a character's file_id). You will use this in the `save_img` field when defining a route.

So, if you wanted to define a particular image to be used during a route for Emmma, it might look like:

```
image save_emma = "Menu Screens/Main Menu/save_img_emma.webp"
```

Then, when defining your route, you could have the `save_img` field like:

```
default emma_good_end = ["Good End",
    RouteDay("5th",
        [ChatRoom("Bright and early...", 'emma_d5_c1', '00:33', [em], save_img="emma
↪")]
    )
]
```

Note that the string given to `save_img` is "emma" because the image was defined as `save_emma`, so you drop the `save_` prefix and pass the rest of the image name (in this case, "emma").

14.8 Adding Greeting Messages

To give Emma greeting messages on the main menu, she'll first need a greeting image (if you haven't defined it already). Find instructions here: *Greeting Images*.

Next, head to `01_mysme_engine/stable/variables_start_screen.rpy`. Below a few Python and variable definitions are a bunch of constants called `morning_greeting`, `afternoon_greeting` etc.

To add greetings for Emma, you'll need to add to these dictionaries. This example will show how to add greetings to `morning_greeting`:

```
define morning_greeting = {
    'em' : [ DayGreeting('Emma/Morning/emma-1',
                "Welcome to Mysterious Messenger!",
                " !"),
             DayGreeting('Emma/Morning/emma-2',
                "Good morning"
                " ")],

    'ja': [ DayGreeting('Jaehee/Morning/ja-m-1',
```

(continues on next page)

(continued from previous page)

```
                "A brand new day has started",
                "   "),
            DayGreeting('Jaehee/Morning/ja-m-2',
                "Did you have breakfast?",
                "  ?"),
            DayGreeting('Jaehee/Morning/ja-m-3',
                "I pray that I can get off work on the dot today.",
                "    ."),],

    # (Rest of the definition omitted for brevity)
```

First, start by adding a key for Emma. Her `file_id` is "em", so add that as the key:

```
define morning_greeting = {
    'em' : [


    ],
```

The value corresponding to this key is a list, hence the `[]` square brackets. This list will contain all the possible greetings for Emma during the specified time period (in this case, the morning).

Next, you'll add `DayGreeting` objects to the list. A `DayGreeting` object consists of three parts: the file where the audio can be found, the English text of the greeting, and the Korean text of the greeting.

sound_file The path to the sound file for the greeting. It is automatically prefixed with `greeting_audio_path` and then `greeting_audio_extension` is appended to it. These values are defined near the beginning of the `variables_start_screen.rpy` file. By default, this means that something like `'Emma/Morning/emma-1'` will become `"audio/sfx/Main Menu Greetings/Emma/Morning/emma-1.wav"`. You can provide your own extension as well as one of the arguments.

english Optional. The English text of the greeting. If not provided, defaults to "Welcome to Mysterious Messenger!".

korean Optional. The Korean text of the greeting. If not provided, defaults to " !".

extension Optional. The sound file extension to use. If not provided, defaults to `greeting_audio_extension`.

Greetings are separated into different definitions for the time of day. Make sure you add the correct greeting to the correct definition so it shows up at the right time on the main menu screen.

Tip: Any character who has a greeting must have a corresponding greet image to display on the menu along with the greeting text, but they don't need a full ChatCharacter definition.

CHAPTER

FIFTEEN

CG ALBUMS

15.1 Adding a CG Album

15.1.1 Albums Associated a Character

For this tutorial, this example shows how to add a character named Emma to the game. You can view Emma's character definition and more in *Adding a New Character to Chatrooms*.

Albums are defined in `gallery_album_definitions.rpy`. There are two images to define and one album variable.

Note: Prior to v3.3.0, the game required two album definitions, a variable and a persistent version. The current version combines both of these into one. If you were using the previous version, your album images should automatically update on game launch, and then you can switch over to the current way of defining albums.

First, under the heading **Album Cover Images**, you will see images that look like `cg_label_ja`. This is the background of the small label that contains the name of the album. For Emma, you will define an image like:

```
image cg_label_em = "CGs/label_bg_em.webp"
```

The important thing is that it is called `cg_label_em` where `em` is the character's file_id. You can look at the existing images to see what they look like. The CG label is typically 241x64 px.

Below that is a set of images like `ja_album_cover`. This is the image that will be displayed as a button to click and open the album. It typically has an image of the associated character and is 157x137 px. Emma's image definition will look like:

```
image em_album_cover = "CGs/em_album_cover.webp"
```

The important part is that it's called `em_album_cover` where `em` is the character's file_id.

Next, you need to define the album:

```
define em_album = []
```

The album should be the character's file_id + album, so in this case it's `em_album`.

At the bottom of the file, you will also see a definition for the variable `all_albums`. You need to add Emma's file_id here so that her album shows up in the Album screen:

```
default all_albums = [
    'ju', 'z', 's', 'y', 'ja', 'v', 'u', 'r', 'common', 'em'
]
```

You can reorganize this list as you see fit. Albums are organized in rows of three on the Album screen. So, the following definition would be equally correct, depending on which characters you wanted to have albums:

```
default all_albums = [
    'ju', 'z', 's', 'y', 'ja', 'em', 'common'
]
```

Tip: If a player has already unlocked an image inside an album, it will be automatically added to `all_albums`. If you're making a new route and don't want some characters' albums to appear, you must both 1) remove them from `all_albums` and 2) Use `Reset Persistent` from the Ren'Py launcher to cause Ren'Py to forget which CGs you unlocked over other routes.

15.1.2 Albums not Associated With a Character

If you want an album that isn't associated with any particular character, such as a "Common" album, the program expects you to follow certain naming conventions so it can find the correct album cover, label, and variables.

For example, if you wanted to add a "New Year's" album, you must do the following:

1. Add "new year's" to the `all_albums` definition. If a string is all lowercase, the first letter will be capitalized when it is displayed on the Album screen, but otherwise capitalization is preserved. Also note the use of double quotes (") instead of single so you don't have to escape the apostrophe.

```
default all_albums = [
    'ju', 'z', 's', 'y', 'ja', 'v', 'u', 'r', 'common', "New Year's"
]
```

2. Determine what the program will expect the variables to be called.

The program uses a specific naming scheme to process strings into file and variable names:

1. Spaces are turned into underscores (e.g. "bonus images" -> "bonus_images")
2. Apostrophes are removed (e.g. "valentine's day -> "valentines_day")
3. Entire word is in lowercase (e.g. "RFA Bonus" -> "rfa_bonus")

So, using the rules above, "new year's" in the `all_albums` definition will become "new_years". You can then define the two images and album definition like you did for Emma above:

```
image cg_label_new_years = "Image for your label.png"
image new_years_album_cover = "Image for your album cover.png"
define new_years_album = []
```

15.2 Defining a CG

For any CG you would like to show in-game, you must first go to `gallery_album_definitions.rpy` and define an image under the **CGs** header. For this example, a fourth CG in the **Common** album will be added. CG images should take up the entire screen, which is 750x1334 px. CGs of other sizes may not display correctly.

First, define the image:

```
image cg common_4 = "CGs/common_album/cg-4.webp"
```

The name of the cg must be `cg` + the name of the album it is found in, minus "album", plus an underscore and some identifier for the image such as a number (`4`), or a descriptor of the CG. Other possible CG definitions might be:

```
image cg common_flower = "CGs/common_album/cg-flower.webp"
image cg ju_meeting = "CGs/ju_album/ju-meeting-office.webp"
```

After defining your image, you must add it to the correct album. See *Adding a CG Album* for more on creating new albums as well.

```
define common_album = [
    GalleryImage("cg common_1"),
    GalleryImage("cg common_2"),
    GalleryImage("cg common_3"),
    GalleryImage("cg common_4")
]
```

In this example, no unique thumbnail was specified. If the program can find an image with the suffix `-thumb` before the file extension, it will use that as the thumbnail. So, since the image is found at "CGs/common_album/cg-4.webp", the program will look for a thumbnail image at "CGs/common_album/cg-4-thumb.webp".

Otherwise, you can also manually specify a thumbnail as the third argument to GalleryImage or just by specifying it using `thumbnail=` in the definition:

```
GalleryImage("cg common_4", thumbnail="CGs/thumbnails/common_4_thumbnail.webp")
```

Typically thumbnails are 150x150 px. If one is not provided, the given CG is cropped and resized to the appropriate size.

The full list of arguments to the GalleryImage definition is below:

name A string. Typically this ends up being the same as `img` below if you define your CGs as described above, but you could also do something like:

```
GalleryImage("cg common_5", "CGs/Common/special.webp")
```

This would have the same effect as if you'd defined `cg common_5` using the `image` declaration.

img A Displayable, typically a string with the name of the image for this CG as it should appear in the album. Can be omitted if it is the same as `name`. If this is the file path to the image, you should use the `name` property to give the CG a "script-friendly" name, as that's the one you'll use to unlock it in-game.

thumbnail Optional. An image path or displayable for the thumbnail as it should appear in the Gallery. Should be 155x155 pixels. If not provided, the CG is scaled and cropped to fit the thumbnail size.

locked_img Optional. The file path to the image that will be used as the "locked" thumbnail icon in the gallery. Should be 155x155 pixels. Typically has a "?" or a locked symbol somewhere to indicate this gallery image hasn't been unlocked yet.

This is often used to indicate, for example, that an image comes from a particular DLC.

chat_img Optional. A Displayable (typically an image path or string containing the name of a defined image) which will be shown to the player at full-size in the chatroom only. The supplied `img` field will be used when viewing this image full-size in the gallery instead.

Use this to do things like supply blurry or edited CGs in the chatroom, but provide their regular unedited version in the gallery.

chat_preview Optional. A Displayable (typically an image path or string containing the name of a defined image) which will be shown to the player in the chatroom only. Clicking this image will show either `chat_img` (if available) or `img` (if not) full-screen. Typically this is about 35% of the full screen size, or 263x467 pixels, but can be whatever dimensions you like.

Use this to customize the image preview in the chatroom, such as blurring the thumbnail for "spoilers" or to put the thumbnail focus on a particular part of the image.

Note: The equivalent property was called `chat_thumb` in the `Album` object prior to v3.3.0.

So, for example, you could define a GalleryImage object like so:

```
GalleryImage("cg common_1",
    thumbnail="common_1_thumb",
    chat_img="cg common_1_edit",
    chat_preview="CGs/spoiler_img.png"
)
```

(Note that this assumes you have `image cg common_1 = "..."`, `image common_1_thumb = "..."`, `image cg common_1_edit = "..."` etc.)

You would then still be able to use the CG in the usual manner described below to show in chatrooms or text messages.

15.2.1 Large Thumbnails

Recall our additional CG image:

```
image cg common_4 = "CGs/common_album/cg-4.webp"
```

For better compatibility with the new profile picture system, you may also want to provide a "larger" version of a thumbnail for use in profile pictures on the profile screen. The program will search for an image with the name of the thumbnail + the suffix `-b` before the file extension. So, if "cg common_4" isn't given a different thumbnail, it will look for the large version of the thumbnail at "CGs/common_album/cg-4-thumb-b.webp".

If you provided a different thumbnail, as in `GalleryImage("cg common_4", thumbnail="CGs/thumbnails/common_4_thumbnail.webp")`, then the large version is expected to be called "CGs/thumbnails/common_4_thumbnail-b.webp".

15.2.2 Adding a CG After Starting the Game

In some cases, you may not want a CG to appear in a character's album unless some condition is met, such as a particular DLC being unlocked. In such a case, you can make use of the `condition` property of a `GalleryImage` object.

For example, say you have a variable, `persistent.new_years_dlc`, which tracks whether the player has the New Year's DLC or not. You only want to show "missing" New Year's DLC images in their album if they already have the DLC. In such a case, you can do:

```
image ja_ny1 = "CGs/Jaehee/ny1.webp"
define ja_album = [ GalleryImage(
        "ja_ny1",
        locked_img="CGs/new_years_locked.webp",
        condition="persistent.new_years_dlc"
    )
]
```

This will define a gallery image named "ja_ny1" with the locked thumbnail at "CGs/new_years_locked.webp" and the condition of `"persistent.new_years_dlc"`. The fact that `persistent.new_years_dlc` is in quotes is

important - the program uses this to evaluate the condition when it's going to show the gallery (as opposed to right at the start of the game).

You can use this for more complex conditions as well, so long as they are enclosed in quotes. Some examples:

```
"'jaehee' in persistent.new_years_dlc_endings"
"persistent.new_years_dlc_endings >= 1 and persistent.new_years_dlc and 'jaehee' in␣
↪persistent.new_years_dlc"
```

Typically you should use persistent variables for the conditions, because the player can view their album from the main menu as well so any save file-specific values won't necessarily be used.

Warning: The following code *will only* work if you use the old definition format for albums (namely, the one with a `default` album and a `persistent` version). The updated version of this code (as of v3.3.0) can be found above, and should be used in preference to the version below. The following version is maintained for compatibility purposes only.

While in most cases you should define your CGs in `gallery_album_definitions.rpy`, you can also add new CGs to an album after the game has already started with the function `add_to_album`. This function takes two parameters:

album The album variable that should contain this new CG.

e.g. `ja_album`

photo_list An `Album` or list of `Album` objects which should be added to the given album variable above.

e.g. [Album("cg s_4"), Album("cg s_5")]

Typically you would use this function at the beginning of a route, particularly if the route is DLC since this will allow you to add images to the album without having to directly modify the `gallery_album_definitions.rpy` file. An example may look like:

```
label new_year_prologue():

    $ new_route_setup(route=new_years_route, chatroom_label='new_year_prologue',
    participants=[ja])
    $ paraphrase_choices = True

    # Album definitions for this new route
    $ add_to_album(ja_album, Album('cg ja_ny_1'))
    $ add_to_album(ju_album, Album("cg ju_ny_1"))
    $ add_to_album(s_album, [ Album("cg s_ny_1"), Album("cg s_ny_2") ])

    $ character_list = [ju, z, s, y, ja, m]
    $ heart_point_chars = [ju, z, s, y, ja]

    # Route prologue begins here
```

15.3 Hiding Albums Until Unlocked

In some situations, you may want an album to not show up in the player's photo album until they have unlocked an image contained in it. For example, if you include a New Year's scenario, you may want to put related CGs in a New Year's album, but if the player hasn't unlocked or played through the New Year's scenario, you don't want the New Year's album to show up in their photo album screen.

In this case, you can use the line

```
$ hide_albums(["new year's"])
```

to hide this album unless the player has unlocked a photo in it. The best place to put this is just before setting up a new route e.g.

```
$ hide_albums(["new year's"])
$ new_route_setup(route=my_new_route)
```

If the player has already unlocked images in this album, it will continue to be shown. Otherwise, this album will only appear in the player's photo album once they have unlocked an image in it (this is taken care of automatically).

Note that since you are passing a list, you can pass multiple albums to be hidden e.g.

```
$ hide_albums(["new year's", "christmas", "b"])
```

15.4 Showing a CG in a Chatroom or Text Message

In chatrooms or text messages, sometimes characters will post images that the player can click on the view full-size. These images will automatically be unlocked in the appropriate Album once the player has seen them.

To show a CG in the chatroom, put the name of the CG in the character's dialogue e.g.

```
s "cg common_4" (img=True)
# or with the msg CDS
msg s "cg common_4" img
```

You can also omit `cg` at the beginning, so long as you remember to mark it as an image:

```
s "common_4" (img=True)
msg s "common_4" img
```

The program will take care of resizing the CG for the chatroom and allowing the player to view it full-size. It will also unlock the CG in the appropriate album and notify the player if they have not yet seen the image in the album. If this is the first time the player has seen this image, it will also become available for use as a bonus profile picture (see *Bonus Profile Pictures*).

You can see an example of a CG posted in a text message in `tutorial_3b_VN.rpy`, and an example of a CG posted during a chatroom in `tutorial_5_coffee.rpy`.

15.5 Showing a CG during Story Mode

All you need to do to have an image unlock after showing it in a Story Mode section is to show it to the player. This can be done through the `scene` or `show` statements. `scene` will clear the screen of any existing character sprites/backgrounds etc before showing the image. It's highly recommended that if you are showing a CG outside of a chatroom that you use the `image` statement to declare a "short" name for the CG (e.g. `image cg common_4 = "CGs/Common/com4.webp"`).

You can display it in script with `scene` or `show`, as mentioned:

```
ju "I wanted to show you how the lounge has been decorated."
scene cg common_4
show jumin front neutral
ju "Do you like it?"
```

or

```
ja "Oh, no, I've spilled the flour everywhere."
show cg common_4
ja "Could you get something to clean this up with?"
```

In most cases, you will probably use `scene` to show a CG image to the player instead of `show`.

The CG can be cleared from the screen either by replacing it with another `scene` statement or by explicitly hiding it with `hide cg`:

```
ju "I wanted to show you how the lounge has been decorated."
scene cg common_4
show jumin front neutral
ju "Do you like it?"
scene bg meeting_room # This clears the CG from the screen
ju "I think it turned out rather well."
```

or

```
ja "Oh, no, I've spilled the flour everywhere."
show cg common_4
ja "Could you get something to clean this up with?"
hide cg # This clears the CG from the screen
show jaehee happy
ja "I'm sorry for the trouble."
```

You can see an example of a CG posted during a Story Mode section in `tutorial_8_plot_branches.rpy`.

CHAPTER

SIXTEEN

MISCELLANEOUS

16.1 Pronoun Integration

Mysterious Messenger allows the player to change their pronouns and gender whenever they desire during the game. This means that any reference to the player's gender or use of pronouns to refer to the player must be taken care of via variables.

At the top of `variables_editable.rpy` you will see a function called `set_pronouns` and several existing variables defined under the header **PRONOUN VARIABLES**. These can be used in script:

```
s "Aw, it doesn't look like [name] is logged in. I wonder what [they_re] doing?"
```

For a player with she/her pronouns, the final part of this dialogue will appear as "I wonder what she's doing?" whereas a player with they/them pronouns will see "I wonder what they're doing?"

It's important to remember that many verbs conjugate differently for "he/she" than for "they", which is why you should use the pronoun variables. Ren'Py's interpolation is very flexible; for example, you can negate the `do_does` variable like:

```
ju "I think [name] said [they] [do_does]n't know if [they]'ll be free this weekend."
```

For a player with they/them pronouns, this displays as "I think [name] said they don't know if they'll be free this weekend", meanwhile, a player with he/him pronouns will see "I think [name] said he doesn't know if he'll be free this weekend." (Note: `[name]` is replaced with the player's name)

Note: In Mysterious Messenger, players can pick their gender separately from their pronouns. This means, for example, that a player can identify as nonbinary and use she/her pronouns. You are welcome to extend this functionality as well to have the player further clarify how they want to be referred to.

Note that because a player's pronouns will not necessarily directly correlate to their gender, it may be good to ask the player if they would like to be referred to as a woman, man, person, or something else, if that comes up in your writing. You can use input prompts to get more specific information from the player as well (see *Getting Input from the Player*).

16.1.1 Defining Additional Pronoun Variables

If you would like to define additional variables to help with scripting dialogue, you must define the variable in `variables_editable.rpy` under the **PRONOUN VARIABLES** header and also in the `set_pronouns` function.

For this example, a variable called `go_goes` will be defined for the three main pronoun options.

First, in `variables_editable.rpy` under the **PRONOUN VARIABLES** header, add

```
default go_goes = "go"
```

Then in the `set_pronouns()` function under all the `global` declarations, add

```
global go_goes
```

at the top of the function.

Next, under both `if persistent.pronoun == "she/her"` and `elif persistent.pronoun == "he/him"` add the line

```
go_goes = "goes"
```

and under `elif persistent.pronoun == "they/them"` add

```
go_goes = "go"
```

And you're done! To use your new variable in dialogue, you can type

```
y "Yeah, [they] said [they] usually [go_goes] out on Fridays."
```

If the player has they/them pronouns, in-game this displays as "Yeah, they said they usually go out on Fridays", but a player with she/her pronouns will see "Yeah, she said she usually goes out on Fridays.".

Variables are capitalization-sensitive; if you need a capitalized version of a variable you can either create another variable (see `They` vs `they` for an example of this), or you can write the variable with `[is_are!cl]` to get the first letter capitalized (so, "Is" or "Are") or `[is_are!u]` to get the whole word in capitals (so "IS" or "ARE"). See https://www.renpy.org/doc/html/text.html#interpolating-data for more information on interpolation flags.

There is no limit to how many pronoun variables you can make, so feel free to create as many as you need to write your script more easily while supporting the different pronoun options.

16.1.2 Adding Additional Genders

Mysterious Messenger comes with three possible genders that the player can choose from on the profile page: nonbinary, female, and male. You are welcome to add more options to this list via the `gender_options` variable found in `variables_editable.rpy`. The default list looks like so:

```
define gender_options = ["nonbinary", "female", "male"]
```

The first item in the list is what will appear on the profile page when the player first starts the game. If you would like to add more options, simply add them to this list e.g.

```
define gender_options = ["nonbinary", "female", "male", "agender", "genderfluid"]
```

If you ever want to change dialogue based on the player's gender, you can write statements like:

```
if persistent.gender == "female":
    s "I'm sure Jaehee appreciates not being the only lady lol"
elif persistent.gender == "male":
    s "It's nice to have a guy around here who understands my jokes lolol"
# Can add more elif clauses in here as needed
else:
    s "It's been so much fun having you here since Day 1, [name] ^^"
```

16.1.3 Using Gendered Terms

Since the program allows players to choose their gender and pronouns, you may need to consider these factors when writing dialogue. Luckily, there is a built-in function, `get_term`, which is designed to assist with this.

The function can be found in `variables_editable.rpy` and currently functions as follows:

- If the player has she/her pronouns *and* identifies as female, the "feminine" term will be used
- If the player has he/him pronouns *and* identifies as male, the "masculine" term will be used
- Otherwise, the neutral term is used

To use it, you will call the function like so:

```
$ cutie = get_term("cute girl", "cute boy", "cutie")
```

in which the first argument ("cute girl") is the term that should be used for female players, the second argument ("cute boy") for male players, and the final argument for a neutral term. A similar use-case might be:

```
$ datefriend = get_term("girlfriend", "boyfriend", "datefriend")
s "So... can I call u my [datefriend] lol"
```

You can use this function anywhere you might need to refer to the player with a term that can be considered gendered.

16.2 Custom Emojis

If you'd like to add your own emojis to the program, you need to define it as an image and add a few lines to the `emoji_lookup` dictionary found in `emoji_definitions.rpy`. Emojis are saved as a series of frames, found in the `images/Gifs/` folder.

The bottom of `emoji_definitions.rpy` contains several image definitions which look like:

```
image jaehee_angry:
    "Gifs/Jaehee/emo_jaehee_angry1.webp"
    0.5
    "Gifs/Jaehee/emo_jaehee_angry2.webp"
    0.5
    repeat
```

The name of this image is `jaehee_angry`. It is made up of two separate images with a half-second pause between them. The first image is "Gifs/Jaehee/emo_jaehee_angry1.webp" and the second is "Gifs/Jaehee/emo_jaehee_angry2.webp". `0.5` tells the program to wait 0.5 seconds before showing the image on the next line. `repeat` lets the program know that it should loop indefinitely over these two images.

You're not limited to just two frames; for a smoother animation, something like:

```
image my_new_emoji:
    "Gifs/Example/frame1.webp"
    0.25
    "Gifs/Example/frame2.webp"
    0.25
    "Gifs/Example/frame3.webp"
    0.3
    "Gifs/Example/frame4.webp"
    0.25
    repeat
```

will also work. This will cycle through frames 1-4 on repeat. Note that the time between each frame can be adjusted according to the animation's needs.

In order for the emoji to have an accompanying sound, you need to add it to the `emoji_lookup` dictionary near the top of the file. Assuming your image is named `my_new_emoji` as shown in the example definition above, you need to create an entry for your image like so:

```
"{image=my_new_emoji}": "audio/sfx/Emotes/Example/some_audio_file.mp3"
```

`my_new_emoji` is the name of your defined image, and `audio/sfx/Emotes/Example/some_audio_file.mp3` is the path to the audio file that should play when this emoji is shown.

If you're adding this to the end of the dictionary, ensure there is a comma after each entry before the last e.g.

```
    '{image=zen_shock}': 'audio/sfx/Emotes/Zen/zen_shock.mp3',
    '{image=zen_well}': 'audio/sfx/Emotes/Zen/zen_well.mp3',
    '{image=zen_wink}': 'audio/sfx/Emotes/Zen/zen_wink.mp3', # <- Comma

    "{image=my_new_emoji}": "audio/sfx/Emotes/Example/some_audio_file.mp3"
}
```

Now when you want to show your new emoji in a chatroom or text message, you just need to type

```
ja "{image=my_new_emoji}" (img=True)
```

and if you've added it to the emoji_lookup dictionary, the following will also work:

```
msg ja "{image=my_new_emoji}"
```

Note: You may also notice that there are a few variables defined under *emoji_lookup* such as *jaehee_emotes*; these are used in the chatroom generator. In a character's ChatCharacter declaration, you can include the *emote_list* parameter and set it to a list with the names of all their emojis. This is only relevant if you would like a character's emojis to appear in the chatroom creator; they will work in-game without requiring this list.

16.3 Spaceship Thoughts and Chip Prizes

16.3.1 Spaceship Thoughts

When the floating spaceship on the home screen isn't giving out chips, you can click it to view a random thought from one of the characters. You can find the initial list of thoughts in `variables_editable.rpy` under the header **SPACESHIP/CHIP BAG VARIABLES** in the variable `space_thoughts`.

The variable `space_thoughts` can be modified here to change the spaceship thoughts the player sees upon starting the game. They can also be modified in any `after_` label during the game. You can have as many or as few `SpaceThought` objects in the list as you like – even multiple thoughts for the same character. A `SpaceThought` only has two fields:

char The ChatCharacter object of the character whose thought this is. Used to find the background image for this thought.

e.g. ja

thought A string with the thought this character is thinking.

e.g. "I wonder if the cafe downstairs is open..."

To change spaceship thoughts during the game, in the `after_` label of a timeline item, use:

```
$ space_thoughts.new_choices([
    SpaceThought(ja, "I wonder if the cafe downstairs is open..."),
    SpaceThought(ju, "The stripe on this sleeve is 3mm wider than the other stripes.
↪"),
    SpaceThought(s, "I think my body is made of PhD Pepper and Honey Buddha Chips."),
    SpaceThought(y, "Oh no, I'm going to be late for class again!"),
    SpaceThought(z, "Is there a typo in the script here?")
])
```

You can see an example of this in `tutorial_6_meeting.rpy`. Using `space_thoughts.new_choices` will overwrite any of the previous thoughts in the list. If you would like to simply add a thought instead, you can use:

```
$ space_thoughts.add_choices(
    SpaceThought(z, "Wish I could take a nap right now...")
)
```

Or you can add multiple thoughts by using a list:

```
$ space_thoughts.add_choices([
    SpaceThought(z, "Wish I could take a nap right now..."),
    SpaceThought(s, "Oh, a shooting star! How lucky~")
])
```

You can see an example of this in `tutorial_0_introduction.rpy` as part of a profile picture callback (see *Profile Picture Callbacks*).

16.3.2 Chip Prizes

Occasionally when finishing a timeline item, the chip bag on the home screen will give the player heart points and/or hourglasses. These heart points don't count toward any particular character. You can add to the list of possible prizes any time you like. The initial list of prizes is in `variables_editable.rpy` under the header **SPACESHIP/CHIP BAG VARIABLES** in the variable `chip_prize_list`.

You can modify this variable so that the initial chip prizes are different upon starting the game, or modify the list in the `after_` label of any timeline item. You can have as many or as few prizes as you like.

Prizes are listed in something called a tuple with three items. The first is the text that should be shown when the player gets this prize e.g. "A clump of cat hair". The second and third items are numbers. The first number is the approximate amount of heart points the player should receive when they get this prize, and the second number is the number of hourglasses they should receive.

An example prize might look like:

```
("A clump of cat hair", 50, 1)
```

In this case, the prize message is "A clump of cat hair", and the player will receive approximately 50 heart points and exactly 1 hourglass.

The number of heart points is slightly randomized; the number will be within 10% of the value given, so the actual number of heart points the player will receive will be 45-55 in the above example. Hourglasses are not randomized in this way and the player will always receive the exact number of hourglasses specified.

You can add or replace prizes in the same way as Spaceship Thoughts. To replace all the current prizes with a new list of prizes, use the method `new_choices` e.g.

```
$ chip_prize_list.new_choices([
    ("Trash. How unfortunate.", 25, 0),
    ("Leftover snacks Yoosung was eating.", 80, 0),
    ("A month's worth of rent!", 120, 2)
])
```

To add to the existing choices, use `add_choices`:

```
$ chip_prize_list.add_choices([
    ("Bubbly bubbles", 30, 0),
    ("Yoosung's bus is waiting for you!", 130, 1),
    ("A single potato chip", 75, 0)
])
```

16.4 Getting Input from the Player

In addition to choice menus, you can also prompt the player to enter text to get information such as nicknames, age, and other personalized information about the player, and then use this to shape the story later on. These input prompts work in many areas of the game, including chatrooms, story mode, text messages, and phone calls.

To show an input prompt to the player, you'll call the special label `get_input`:

```
get_input(the_var, prompt="", default="", length=20, allow=None,
    exclude=None, show_answer=None)
```

The parameters are explained below:

the_var A string corresponding to the name of a variable where the input will be saved to as a string. For example, if you want to save the player's name to a variable called `nickname` you might have a line like the following:

```
$ nickname = ""
call get_input('nickname', "Enter a nickname")
```

prompt Optional. If included, this text will appear above the input box to remind the player what information they're supposed to be entering. See the example above.

default Optional. A string that will automatically be filled into the input box as the "default" value, which the player can then delete and change if they wish. By default, this is the empty string "" (so, no text will be filled in automatically).

length By default, 20. An integer equal to the number of characters you will allow in the input box. Depending on where you are using the result, excessively long input values may not display correctly everywhere in the game.

allow A string containing all the letters, numbers, symbols etc that are permitted to be entered as input. By default, all characters accepted by the font are allowed (except for `{}` which can cause program issues). For example, you can use this to ensure you receive a number as input:

```
msg u "How old are you?"
$ player_age = "23"
call get_input('player_age', "Enter your age", "23", allow="0123456789")
if player_age == "":
    msg u "So you won't tell me, huh?"
    $ player_age = 0
else:
    $ player_age = int(player_age)
    msg u "You're [player_age]?"
```

The line `$ player_age = int(player_age)` ensures that the input, which is always a string, is then converted to an integer value which you can do math on if you so please.

There is a pre-defined variable, `allowed_alphabet`, which you can give to `allow` to only accept letters from the alphabet as well as spaces, dashes (-), and apostrophes e.g.

```
msg z "Do you have any nicknames you want us to use?"
$ nickname = ''
menu (paraphrased=True):
    "I do have a nickname (enter nickname)":
        msg m "I do have a nickname." pv 0
        call get_input('nickname', "Enter a nickname", allow=allowed_alphabet)
        # This loops so long as the player leaves the input blank
        while not nickname:
            z "Oh, so you don't have a nickname then?"
            menu:
                "No, I don't have a nickname.":
                    jump no_nickname
                "Yes, I do. I want to input it again.":
                    pass
            call get_input('nickname', "Enter a nickname", allow=allowed_alphabet)
        msg m "Call me [nickname]" pv 0
        z "[nickname], huh?"
        z "Sounds cute!"
    "I don't have a nickname." (paraphrased=False):
        z "That's fine!"
```

exclude Like allow, a string which contains all the letters, numbers, symbols etc which are *not* permitted to be entered as input.

accept_blank If True, the default, the prompt will accept the empty string ("") as input. If False, the player must enter at least one valid character for the Confirm button to be active.

show_answer By default, this is True during chatrooms and text messages and ignored during other times (such as phone calls and Story Mode). It will show the answer button at the bottom of the screen which, when clicked, will display the input prompt. If you set this to False, the input prompt will show up right away, without requiring the player to click to answer at the bottom of the screen.

16.5 Hacked Effects

In your route, you may want to cause the program to appear as though it has been hacked. There are a few features to help you achieve this look. First, there is a variable called `hacked_effect` that, if set to `True`, will cause the chatroom timeline to have additional "broken" backgrounds, and will also show a glitchy screen tear effect every 10 seconds or so. You can set this variable to `True` at any point during your route; simply including the line

```
$ hacked_effect = True
```

in an `after_` label, for example, will activate it. Similarly, `$ hacked_effect = False` will get rid of these effects.

While you are testing a route, there is also an option to toggle the `hacked_effect` on/off in the **Developer** settings from the home screen.

> **Warning:** Many players may find the hacked effects distracting or irritating, so they should be used sparingly. There are also options in the Settings screen to turn off these effects, so know that not every player will be able to see or appreciate them.

16.5.1 The tear screen

There is a special screen called the `tear` screen which will cause the screen to be split into several smaller pieces that are offset a little from their original position. It's used to create the hacked effect in the chatroom timeline screen, but can also be shown in the middle of any ordinary chatroom/phone call/Story Mode/etc. You can call it like so:

```
call show_tear_screen(num_pieces=25, xoffset_min=-10, xoffset_max=30,
                      idle_len_multiplier=0.4, move_len_multiplier=0.2,
                      w_time=0.2, p=0.5)
```

The arguments are as follows:

num_pieces An integer. The number of horizontal slices the screen will be split up into. More pieces increase the distortion, but are also more resource-taxing. Typically it is good to keep this number under 50 or so for a full-screen image (like a screenshot, the default), and even less for images which are smaller. More pieces may take longer to render.

xoffset_min An integer. The minimum number of pixels the piece can be moved from its original starting position. Negative numbers move the piece to the left of its starting position.

xoffset_max An integer. The maximum number of pixels the piece can be moved from its original starting position. Positive numbers move the piece to the right of its starting position. This, along with `xoffset_min`, are used to generate a random number for the xoffset of the torn piece.

idle_len_multiplier A float. A multiplier for how long the piece stays in an "idle" state (that is, it appears normally as opposed to being offset). Larger values mean the piece spends more time in its original position.

move_len_multiplier A float. A multiplier for how long the piece stays in its offset position. Larger values mean the piece spends more time away from its original position.

w_timer A float. How long the program should display the torn image for, in seconds, before it is automatically hidden. If this is not provided, the screen must be hidden manually.

p A float. The number of seconds the program should pause for after showing the torn screen. This is provided as an argument to the call so that it can be reproduced in replays. You should not write a pause after showing the torn screen.

The tear screen also takes an image argument and width and height arguments to size it.

img A displayable, usually the file path to an image or the name of a defined image. This is the image that will be torn into horizontal slices for the tearing effect, rather than a screenshot of the current screen.

width An integer. The width, in pixels, of the provided image above.

height An integer. The height, in pixels, of the provided image above.

If the `img`, `width`, and `height` arguments are given to `show_tear_screen`, then instead of tearing the current screenshot into vertical slices, the provided image will be torn. It is displayed in the center of the screen. Otherwise, the rest of the arguments (`num_pieces`, etc) all apply identically to the provided image when tearing it.

You can also briefly show an image on-screen before showing the tear screen in order to have that image "torn" along with the rest of the screen. For this, you can use the special screen *display_img*:

```
show screen display_img([ ['vn_party', 200, 400] ])
pause 0.0001
call show_tear_screen(40, 0.4, -100, 100, 0.2, 0.3, 0.3)
hide screen display_img
```

The `display_img` screen will show an image in the specified position. It takes a lists of lists as its sole parameter. Each item in the list should be a list of three items: the image to display (which can be the name of a previously declared image, as above, a string with a path to the file name, or a Transform with the image, among other things), then the xpos, and then the ypos. So, in this case, the program displays the image `'vn_party'` at an xpos of 200 and a ypos of 400.

The short pause after showing the `display_img` screen gives the program enough time to register the images before it takes a screenshot for the tear screen. Call the tear screen as normal. Be sure to hide the *display_img* screen at the end.

Note: Prior to v3.3.0, a different label and screen were used to tear the screen into pieces. This label, `tear_screen`, still exists in the program and can be used with its original arguments (aka you don't have to update it to work on 3.3.0). However, for new torn screen sections, you should use the new label call and arguments as described above.

The argument names have also been changed for the new label, but remain unchanged for the old method. `number` is now `num_pieces`, `offsetMin`/`offsetMax` is now `xoffset_min`/`xoffset_max`, `offtimeMult`/`ontimeMult` is now `idle_len_multiplier`/`move_len_multiplier`. `w_timer` and `p` remain unchanged.

16.5.2 The hack_rectangle screen

The `hack_rectangle` screen will also help create a "hacked" effect. It will show several random rectangles on the screen, and works well when paired with the tear screen. For example:

```
call hack_rectangle_screen(t=0.2, p=0.01)
call tear_screen(number=10, offtimeMult=0.4, ontimeMult=0.2,
    offsetMin=-10, offsetMax=30, w_timer=0.18, p=0.18)
```

This will show the `hack_rectangle` screen for 0.2 seconds (`t=0.2`), and after pausing for 0.01 seconds (`p=0.01`) to give the program time to register the images on-screen, it shows the *tear* screen for 0.18 seconds.

16.5.3 Static white squares

The program also has a screen called `white_squares` which randomly shows a sequence of white "static" squares on top of the screen. It is used for the chatroom select screen when `hacked_effect` is True. To call it, use

```
call white_square_screen(t=0.16, p=0.17)
```

where `t=0.16` is how long to show the screen for (0.16 seconds) and `p=0.17` tells the program how long to pause for before continuing (0.17 seconds).

16.5.4 Inverting the screen colours

Finally, there is also a screen called `invert` which will take a screenshot of the currently displayed screen and invert the colours:

```
call invert_screen(t=0.19, p=0.2)
```

where `t=0.19` is how long to show the screen for (0.19 seconds) and `p=0.2` is how long the program should pause for before continuing (0.2 seconds). This works well with screens that were previously mentioned. For example:

```
call hack_rectangle_screen(t=0.2, p=0.01)
call invert_screen(t=0.19, p=0.01)
call tear_screen(number=10, offtimeMult=0.4, ontimeMult=0.2,
        offsetMin=-10, offsetMax=30, w_timer=0.18, p=0.01)
call white_square_screen(t=0.16, p=0.17)
```

16.6 Removing Messages from the Chatlog

Besides just adding "hacked" effects, above, you may also want to remove entries from the chatlog for dramatic effect. This should typically be used sparingly, as it may be jarring for the reader.

Here is an example from `tutorial_7_hacking.rpy`:

```
menu:
    "I don't want to freak them out exactly...":
        sa "You don't, hmm?"
        sa "Just you wait a moment"
        m "Show me what to do."
        # This deletes the last three items in the chatlog, discounting
        # the most recent message.
```

(continues on next page)

(continued from previous page)

```
        # You might have to experiment with how many messages to
        # delete/where to put the delete line since the program sometimes
        # has "hidden" chatlog entries that aren't shown to the user
        # In general you can put it one message after the last message
        # you want to delete
        call remove_entries(num=4)
```

In this case, the automatic message posted by the player ("I don't want to freak them out exactly…"), as well as Saeran's next two lines ("You don't, hmm?" / "Just you wait a moment") get removed from the chatlog via the line `call remove_entries(num=4)`. As can be seen in this case, this number isn't always precisely the number of message you want to remove, as there are some internal calculations that take place when removing messages. Typically you would put `call remove_entries` *after* the first message which you want to remain on-screen due to the order in which messages are posted.

You may need to experiment with the placement of this line + the number of messages to delete until you get the desired result.

16.7 Adding New Audio

In order to support audio captions, defining new music and sound effects to play in the program requires a few extra steps.

16.7.1 Defining New Music

All music currently in the game is defined in `variables_music_sound.rpy`. Open that file and you will see several statements for the existing background music.

For this example, a song called "Jingle Bells" will be added.

First, define a variable that leads to the audio file:

```
define jingle_bells = "audio/music/jingle_bells.ogg"
```

Next, in the `music_dictionary` variable, add `jingle_bells` to the end like so:

```
define music_dictionary = {
    # (Complete definition omitted)
    april_mystic_chat : "Upbeat 8-bit music",
    april_mysterious_clues : "Sinister 8-bit music",
    april_dark_secret : "Suspenseful 8-bit music",

    jingle_bells : "Jolly Christmas music"
}
```

This should be a pair of the name of your music variable (`jingle_bells`) and the description as will be displayed in an audio caption. Try to keep the description short while still conveying the general mood/feel of the song.

You can now play your music in-game via:

```
play music jingle_bells
```

16.7.2 Defining New Sound Effects

All sound effects currently in the game is defined in `variables_music_sound.rpy`. Open that file and you will see several statements for the existing background music.

For this example, a sound effect called "glass_breaking" will be added.

First, define a variable that leads to the audio file:

```
define glass_breaking_sfx = "audio/sfx/glass_breaking.ogg"
```

Next, in the `sfx_dictionary` variable, add `glass_breaking_sfx` to the end like so:

```
define sfx_dictionary = {
    car_moving_sfx : "Sound of a car moving",
    door_knock_sfx : "A knock at the door",
    door_open_sfx : "The door opens",
    glass_breaking_sfx : "A glass shatters"
}
```

This should be a pair of the name of your sound effect (`glass_breaking_sfx`) and the description as will be displayed in an audio caption. The description should briefly describe the action or event the sound effect is meant to convey.

You can now play your sound effect in-game via:

```
play sound glass_breaking_sfx
```

16.7.3 Adding New Ringtones, Text Tones, & Email Tones

In `variables_music_sound.rpy` under the header **RINGTONES** are three variables, one for email tones, text tones, and ringtones. The method of adding a new tone is the same for each, so only adding a new text tone is described below.

First, you need to add a new category to the corresponding tone variable. In this case, `text_tones`. Categories are handled with a special class called `ToneCategory`, which takes the following arguments:

category The name of the category as it should appear in the tone list.

e.g. "Bonus"

folder The folder and/or prefix that should be prepended to the start of each tone name in this category.

e.g. "audio/ringtones/" or "audio/ringtones/**bonus_**"

ext The extension of each file in this category.

e.g. "ogg" or "wav"

condition Optional. If used, this should be given at the end of the definition. A string which evaluates to a condition that determines if this category of tones should be shown to the player. Can be used to unlock bonus ringtones etc after the player has reached the ending of a route, for example.

e.g. condition="persistent.ray_good_end"

A typical ToneCategory definition looks like the following:

```
ToneCategory("Bonus", "audio/sfx/Ringtones etc/text_tone_bonus_", "wav",
    "Bonus Unknown", "unknown",
    "Bonus V", "v",
```

(continues on next page)

(continued from previous page)

```
    "Bonus Rika", "ri",
    condition="name == 'Rainbow'")
```

First, the category's title is given as "Bonus". The second argument, "audio/sfx/Ringtones etc/**text_tone_bonus_**" is combined with the next argument, "wav", to create a file name for each of the tones in this category.

After "wav", each of the following arguments comes in a pair – first, the title of the tone, and second, the name of the tone as combined with the folder and ext arguments to construct a file path to the audio file. So for the above example, the first tone is called "Bonus Unknown", and its sound file is found at "audio/sfx/Ringtones etc/text_tone_bonus_unknown.wav". The last file in this category is called "Bonus Rika" and its associated sound file is "audio/sfx/Ringtones etc/text_tone_bonus_ri.wav".

Finally, there is the special argument `condition`. By default, a ToneCategory is automatically shown to the player. However, you can also add a condition after all the tone definitions. This condition will determine whether the tone is shown to a player.

For the above example, the condition is `condition="name == 'Rainbow'"`. Note that the condition *must* be a string. When evaluating whether this category should be available, the program will take apart the string and evaluate it. In this case, it will evaluate `name == 'Rainbow'`. Therefore, if the player currently has their name set to "Rainbow" when they check the text tones list, they will see the "Bonus" tone category.

You can permanently unlock tones for the player using persistent variables in a similar method to the one described in *Unlockable Routes*. For example, if you want to unlock the bonus text tones after the player has seen the good end of Tutorial Day, you could create a variable called `persistent.tutorial_good_end_complete` which you set to True at the end of the Good End on Tutorial Day. Your condition for the ToneCategory would then be:

```
ToneCategory("Bonus", "audio/sfx/Ringtones etc/text_tone_bonus_", "wav",
    "Bonus Unknown", "unknown",
    "Bonus V", "v",
    "Bonus Rika", "ri",
    condition="persistent.tutorial_good_end_complete")
```

Overall, the `text_tones` variable might look like the following if you were to add a Bonus category:

```
define text_tones = [
    ToneCategory("Basic", "audio/sfx/Ringtones etc/text_basic_", "wav",
        "Default", "1",
        'Jumin Han', "ju",
        'Jaehee Kang', "ja",
        '707', "s",
        'Yoosung', "y",
        'ZEN', "z"),
    ToneCategory("Bonus", "audio/sfx/Ringtones etc/text_tone_bonus_", "wav",
        "Bonus Unknown", "unknown",
        "Bonus V", "v",
        "Bonus Rika", "ri",
        condition="persistent.tutorial_good_end_complete")
]
```

You can add as many tones and categories as you like. The `condition` field is always optional; if omitted, that category will always be available.

16.8 Bonus Profile Pictures

16.8.1 Profile Pictures for the Characters

In this program, the player can change the other characters' profile pictures on that character's profile screen by clicking their current profile picture. In-game, this is treated as a "bonus" profile picture and does not affect dialogue. The bonus profile picture is applied "on top of" the existing profile picture that is set by the game during the story. To return to the intended profile picture, there is a button captioned "Revert to default" which will restore the intended story profile picture.

By default, all the images in a character's corresponding "Profile Pics/" folder are available to choose as a profile picture, as are all the image in their corresponding CG album. This is set up on a per-character basis in `variables_editable.rpy` under the header **BONUS PROFILE PICTURES**.

If you want a new character to have bonus profile pictures, you must set up a variable for them in `variables_editable.rpy`. For this example, the character Emma from *Adding a New Character to Chatrooms* will be given bonus profile pictures.

Because Emma's file_id is `em`, the variable will be called `em_unlockable_pfps`:

```
define em_unlockable_pfps = combine_lists(
    register_pfp(folder="Profile Pics/Emma/", filter_out='-b.'),
    register_pfp(folder="CGs/em_album/", filter_keep='-thumb.')
)
```

The functions used are explained below.

First, `combine_lists` is a special function which will combine all the given arguments into one large list, removing duplicates. It can take individual items or lists of items as arguments. For the most part, you should just make sure that you pass all your bonus profile pictures into this function as shown in the existing variable definitions.

The second and most important function is `register_pfp`. This has many parameters which you may take advantage of that are explained below:

files Should be a string or a list of strings corresponding to file paths that lead to images you want to use for profile pictures. Can be used in combination with the `folder` parameter to prepend folder names to file names.

e.g. ["em-1.webp", "em-2.webp"]

condition By default, this is 'seen', which means that the images passed to the function will become visible on the profile picture screen once the player has seen the image in-game (either as a CG or as another character's profile picture).

Otherwise, the program expects a string that evaluates to a Python condition that determines when these images should be unlocked. For example, if you want the pictures to be unlocked after setting a particular variable, you could write `condition="persistent.saw_casual_bad_end"`, for example. It is best to use persistent variables for conditions so they persist throughout different playthroughs.

e.g. "persistent.seen_endings > 3"

folder A string with the folder path where the program should look for files. This is automatically prepended to each file name while searching.

e.g. "Profile Pics/Emma/"

ext The extension for this file. Automatically appended to each file name when searching.

e.g. "webp"

filter_out Requires the `folder` parameter. If included, searches through files in the given folder that do **not** contain the string in filter_out.

e.g. "-b" (this will *exclude* images in the folder that contain the string "-b")

filter_keep Requires the `folder` parameter. If included, searches through images in the provided folder that **do** contain the string in filter_keep.

e.g. "-thumb" (this will **only** include files if they include the string "-thumb")

These fields can be combined in many ways to intelligently create a filter which will add all your desired images to the character's profile picture list. For example, the current characters have two `register_pfp` statements:

```
define ja_unlockable_pfps = combine_lists(
    register_pfp(folder="Profile Pics/Jaehee/", filter_out='-b.'),
    register_pfp(folder="CGs/ja_album/", filter_keep='-thumb.')
)
```

The first statement looks in the provided folder ("Profile Pics/Jaehee/") and adds all images in that folder so long as they do not contain the string "-b.". This is because the "big" version of each profile picture is saved as something like "jae-1-b.webp", and since the large versions of the images are all duplicates of their smaller counterparts, they don't need to be included in the unlockable profile pictures.

The second statement looks in the folder "CGs/ja_album/" and only adds images if they have the string "-thumb." in them. Since this is a folder that holds the CGs for Jaehee, the full-size CGs shouldn't be added. The square thumbnails for the CGs are always called something like "cg-1-thumb.webp", so the program should only keep images that have the "-thumb." string in their file path.

Other valid uses of the `register_pfp` statement might look like:

```
register_pfp(files=["em-1", "em-2", "em-3"], folder="Profile Pics/Emma/",
    ext="webp")
```

This will include the images "Profile Pics/Emma/em-1.webp" up to "Profile Pics/Emma/em-3.webp".

```
register_pfp(files="Profile Pics/Bonus/em-bonus.webp",
    condition="persistent.saw_emma_end")
```

This will include the image "Profile Pics/Bonus/em-bonus.webp", which will be unlocked when the variable `persistent.saw_emma_end` is True (you are in charge of setting this variable yourself).

Finally, there is also a variable in `variables_editable.rpy` under the **BONUS PROFILE PICTURES** header called `pfp_cost`; this is the number of heart points the player must earn with a character in order to unlock a profile picture with the "seen" condition. All other conditions unlock automatically once their condition is fulfilled.

16.8.2 Profile Pictures for the Player

Whenever the player sees a new CG in-game or a character changes their profile picture, that image is automatically unlocked for the player to use as their own profile picture. Each profile picture requires the player spend 1 hourglass to unlock it.

Note: For testing, if **Testing Mode** is turned on in the Developer Settings, than all profile pictures (for the MC and the NPCs) don't cost heart points or hourglasses.

Unlike bonus profile pictures for the other characters, the player's own profile picture can be commented on in-game and is treated as their current profile picture (see *Profile Picture Callbacks*). Bonus profile pictures remain unlocked across all playthroughs.

The initial set of images available to the player are all the images in the `Drop Your Profile Picture Here` folder. If you would like to add new options for a player who is beginning a certain route, then you can use the special `add_mc_pfp` function:

```
$ add_mc_pfp("Profile Pics/MC/mc_bonus_1.webp")
```

This will add the image "Profile Pics/MC/mc_bonus_1.webp" to the set of profile pictures available to the player, so that the player can unlock it on the profile pictures screen with an hourglass. This function also takes a list of images as its first parameter e.g.

```
$ add_mc_pfp([
    "Profile Pics/MC/mc_bonus_1.webp",
    "Profile Pics/MC/mc_bonus_2.webp",
    "Profile Pics/MC/mc_bonus_3.webp"
])
```

If you would like the images to be added already unlocked, there is a parameter `unlocked` you can set to True:

```
$ add_mc_pfp([
    "Profile Pics/MC/mc_bonus_1.webp",
    "Profile Pics/MC/mc_bonus_2.webp",
    "Profile Pics/MC/mc_bonus_3.webp"
], unlocked=True)
```

You can use this in combination with the `register_pfp` function as well to filter out file names from folders; however, the `condition` field will be ignored.

```
$ add_mc_pfp(register_pfp(folder="Profile Pics/MC Bonus/",
    filter_out='-b.'), unlocked=True)
```

This will add all the images in the "Profile Pics/MC Bonus/" folder without the string "-b." in the file name to the list of available profile pictures, and the images will come already unlocked.

16.9 Profile Picture Callbacks

Note: Example files to look at:

- tutorial_0_introduction.rpy
- variables_editable.rpy

A brief overview of the steps required (more detail below):

1. Create a Python function in an `init -1 python:` block which takes four parameters – the time difference, previous profile picture, current profile picture, and character associated with the current profile picture.
2. Set the variable `mc_pfp_callback` to the name of your function.
3. Write some conditionals inside your callback function to determine which label the program should jump to if the player sets their profile picture to a given image.
4. Create a label for the callback. You can have characters call the player, send text messages, update their status, etc.

If the player changes their profile picture, a special "callback" function is called that allows the characters to comment on their profile picture. This callback function can be different for different routes, and may also be changed in the middle of a route, if desired.

To create a callback function, you must first put it inside an `init -1 python:` block. There are two examples of callback functions in the game – the first is in `variables_editable.rpy` called `bonus_pfp_dialogue`, and the second is the callback that is active during Tutorial Day, called `tutorial_pfp_dialogue`.

A callback function is given four parameters – the time difference since the callback was last called, the player's previous profile picture, their current profile picture, and the ChatCharacter the profile picture is associated with (e.g. if it belongs in a particular character's CG album or they used it as a profile picture). A typical profile picture callback function may look like the following:

```
init -1 python:
    def casual_route_pfp(time_diff, prev_pic, current_pic, who):
```

While you can call these parameters anything you like, they will be explained with the names given above.

time_diff A MyTimeDelta object. This is the difference between the time the player last changed their profile picture and the current time. This can be helpful to prevent the player from constantly changing their profile picture just to see dialogue – at the beginning of your profile picture callback, for example, you can include a conditional statement that will automatically return if less than 30 minutes have passed since the player last changed their profile picture.

A `MyTimeDelta` object has several useful fields you can access to compare the time the player last changed their profile picture at. Each field is rounded **down** to the nearest whole number; so, if 2 minutes and 45 seconds (2.75 minutes) have passed since the player last changed their profile picture, `time_diff.minutes` will be equal to 2 and `time_diff.seconds` will be equal to 165.

The available fields are `days`, `hours`, `minutes`, and `seconds`.

prev_pic The file path of the image that was used as the profile picture before the new one. This will never be the same as `current_pic`; if they are the same, then the profile picture callback will not be called.

current_pic The file path to the image the player just set as their profile picture. This, like `prev_pic`, may also be a string with a colour e.g. "#000".

who The ChatCharacter object of the character this image is associated with. This is determined by checking if the image is in a particular character's `unlockable_pfps` variable (see *Profile Pictures for the Characters*). If the image is not in any character's unlockable profile pictures set, then it will be equal to None.

This can be useful if you want a character to react generally to any image associated with themselves.

You can use all or none of the passed parameters in your callback function. The callback function must either return `None` (either directly, via `return None` or `return`, or implicitly by reaching the end of the function) or return a string or list of strings that correspond to a label the program can call for this profile picture callback.

If the profile picture callback returns a string corresponding to the name of a label, the program will check if it has already jumped to this particular label. If so, then the player has already seen this callback and the program will do nothing. Otherwise, the program will execute the contents of the label before returning to the regular game context. This means that the contents of the label should **not** be real-time – e.g. you can't include a phone call directly in the callback label, but you **can** add a new phone call to the list of available calls, which will then jump to a label with the contents of that phone call.

You can also return a list of label names, in which case the program will check the list until it either finds a label which hasn't yet been executed or reaches the end of the list.

Note: If **Testing Mode** is turned on from the developer settings, profile picture callbacks can execute more than once. Otherwise, they will only activate once on any given playthrough.

Profile picture callbacks should typically be treated the same way as an `after_` label is, and can include the same sorts of functions. Some examples of things that you can do in a profile picture callback label are:

- Have a character send the player a text message
  - You can include a label that will be jumped to to allow the player to reply and/or continue the conversation
- Change a character's status, profile picture, or cover photo
- Add a new spaceship thought
- Have a character call the player, or make a new phone call available for them
- Unlock bonus ringtones or profile pictures

Convenience functions and CDSs are provided to do the most common actions, but the true limit is your own programming ability. You could even use profile picture callbacks to branch the story, add new chatrooms to a route, or show a popup, for example.

16.9.1 Common Profile Picture Callback Examples

The following examples will include conditional statements that will go inside a profile picture callback function – while this function can be called anything, for the purposes of these examples assume the callback is set up like so:

```
default mc_pfp_callback = example_pfp_callback

init -1 python:
    def example_pfp_callback(time_diff, prev_pic, current_pic, who):
```

You can assume that unless otherwise specified, the example code should be inside the profile picture callback function.

Checking the Last Time the Picture was Changed

```
if time_diff.minutes < 30:
    # It has been less than 30 minutes since the player changed their
    # profile picture, so don't execute a callback.
    return
```

```
init -1 python:
    def example_pfp_callback(time_diff, prev_pic, current_pic, who):
        if time_diff.seconds < 60:
            # It has been less than a minute since the player changed their pfp
            return 'quickly_changing_pfp'

# Outside of the main function
label quickly_changing_pfp:
    compose text s:
        s "Wow I thought you just changed your profile picture lol"
    return
```

```
init -1 python:
    def example_pfp_callback(time_diff, prev_pic, current_pic, who):
        if time_diff.days > 3:
            # It has been more than 3 days since the player changed their pfp
            return 'long_time_no_change'

label long_time_no_change:
    $ space_thoughts.add_choices(
        SpaceThought(y, "[name] finally changed [their] profile picture! I was␣
↪starting to feel self-conscious changing mine so much...")
```

(continues on next page)

(continued from previous page)

```
    return
```

Checking if an Image is Associated with a Character

```
if who == r:
    # Profile picture is associated with `r` (Ray)
    return ['ray_pfp_change_1', 'ray_pfp_change_2']
```

This means that the first time the player changes their profile picture to an image associated with Ray (`r`), the label `label ray_pfp_change_1` will execute. The second time the player does so, `label ray_pfp_change_2` will execute.

Checking For a Particular Image

```
if "CGs/ja_bonus_cg" in current_pic:
    return "ja_bonus_cg_dialogue"
```

You should use the `if x in y` terminology to check for particular images, and it is generally a good idea to leave out the file extension (e.g. .png or .webp) to ensure the correct image is matched. The above condition would be True if `current_pic` was "images/CGs/ja_bonus_cg.png" or "CGs/ja_bonus_cg.webp", for example.

The same should be done when comparing a string to the `prev_pic` variable e.g.

```
if "Profile Pics/Ray/ray_03" in prev_pic and not who == r:
    return 'changed_ray_pfp'
```

This would cause `label changed_ray_pfp` to be executed the first time the player changes their profile picture from an image like "Profile Pics/Ray/ray_03.png" to a different image that isn't associated with Ray.

16.10 Paraphrased Choices

Inside `variables_editable.rpy` is a variable called `paraphrase_choices` under the header **MISCELLANEOUS VARIABLES**. If this variable is set to True, choices are treated as "paraphrased" – that is, it is your responsibility to write out exactly what you want the MC to say after a choice. This was the default behaviour prior to v3.0.

However, if `paraphrase_choices` is set to False, then the main character will automatically say the dialogue provided in a choice caption. This means that the following code is equivalent:

```
$ paraphrase_choices = True
menu:
    "I don't want to go.":
        m "I don't want to go." (pauseVal=0)
        ju "I understand."
    "I'll come with you.":
        msg m "I'll come with you." pv 0
        ju "That's good to hear."

# Is equivalent to
$ paraphrase_choices = False
```

(continues on next page)

(continued from previous page)

```
menu:
    "I don't want to go.":
        ju "I understand."
    "I'll come with you.":
        ju "That's good to hear."
```

In both cases, the main character (`m`) will say "I don't want to go." after the player chooses that option (same with "I'll come with you."). This feature works for all choices – i.e. text messages, chatrooms, phone calls, and story mode.

You should set the value of `paraphrase_choices` at the beginning of a route, generally just after the `new_route_setup` function e.g.

```
label new_year_prologue():

    $ new_route_setup(route=new_years_route, chatroom_label='new_year_prologue',
    participants=[ja])
    $ paraphrase_choices = False
    jump skip_intro_setup
```

After setting it, it is expected to remain that way the rest of the route. However, you can toggle it on/off on a per-menu or per-choice basis.

For example, if you have `paraphrase_choices = False` but want to have a menu include paraphrased choices, you can provide the menu argument `paraphrased=True` so that all the choices in that menu will be considered paraphrased:

```
menu (paraphrased=True):
    "(Don't say anything)":
        ri "...I see."
    "(Try to reason with her)":
        m "Rika, really, you don't have to do this."
```

Tip: Setting `paraphrased` for a menu will work with timed menus (see *Paraphrased Choices*) as well as regular menus.

You can also set the `paraphrased` argument for individual choices as well:

```
menu (paraphrased=True):
    "(Don't say anything)":
        ri "...I see."
    "(Try to reason with her)":
        m "Rika, really, you don't have to do this."
    "You're better than this." (paraphrased=False):
        ri "Oh? What makes you think that?"
```

In this case, though the menu is treated as paraphrased (that is, the main character won't directly say the dialogue in the choices), the final choice *is not* paraphrased, so the main character will say "You're better than this" before Rika says "Oh? What makes you think that?".

Paraphrasing can be particularly useful for timed menus, which have limited space for choice text:

```
timed menu:
    s "Ugh, so tired..."
    s "[name], help lol"
    s "I need something to do."
```

(continues on next page)

(continued from previous page)

```
    s "Maybe Yoosung will come by."
    "(Sympathize)" (paraphrased=True):
        msg m "Right? It's been such a slow day." pv 0
        msg m "Sometimes u just wanna be lazy."
    "Get a hobby lol":
        s "Omg 0_0"
        s "ya maybe ur right lololol"
```

In this case, `paraphrase_choices` is set to False for the whole route, but in this menu the choice "(Sympathize)" is paraphrased.

16.10.1 Changing the Main Character

As non-paraphrased dialogue is automatically said by the main character, you may also want to change who the main character is. By default, it is `m`, which uses the name and profile picture set by the player, but you can change it on a per-route basis or change the default definition in `character_definitions.rpy`.

```
default main_character = m
```

If you set this variable equal to a different character (e.g. `em` from the character examples), then `em` will say non-paraphrased choice dialogue instead of `m`. `em` will also appear on the right side of the screen when saying dialogue in chatrooms or text messages.

16.11 Reserved Names

Mysterious Messenger uses several variables, transforms, class names, and other features in order to work. As a result, you can run into errors if you accidentally use the same name as something already in the engine. If you're getting mysterious errors, you might try checking if you've accidentally replaced one of these reserved names.

16.11.1 Transform Names

- album_tilt
- alpha_dissolve
- cg_swipe_left
- cg_swipe_left2
- cg_swipe_left_hide
- cg_swipe_right
- cg_swipe_right2
- cg_swipe_right_hide
- chat_title_scroll
- chip_anim
- chip_wobble
- chip_wobble2
- choice_anim

- choice_disappear_hourglass
- cloud_shuffle1
- cloud_shuffle2
- cloud_shuffle3
- cloud_shuffle4
- cloud_shuffle5
- continue_appear_disappear
- continue_appear_disappear_first
- delayed_blink
- delayed_blink2
- dropdown_horizontal
- dropdown_menu
- fadein_out
- fade_in_out
- flash_yellow
- flicker
- guest_enter
- hacked_anim
- heart
- heartbreak
- hide_dissolve
- hourglass_anim
- hourglass_anim_2
- incoming_message
- incoming_message_bounce
- invisible
- invisible_bounce
- large_tap
- lightning_cloud_flash
- lock_spin
- med_tap
- moon_pan
- move_clouds
- new_fade
- notify_appear
- NullTransform

- null_anim
- participant_scroll
- reverse_topbottom_pan
- scale_vn_bg
- shake
- shooting_star
- shrink_away
- slide_in_out
- slide_up_down
- slide_up_down
- slow_fade
- slow_pan
- small_tap
- spaceship_chips
- spaceship_flight
- speed_msg
- stack_notify_appear
- star_fade_in
- star_place_randomly
- star_rotate
- star_twinkle_in
- star_twinkle_out
- star_twinkle_randomly
- text_footer_disappear
- toggle_btn_transform
- topbottom_pan
- topbottom_pan2
- tutorial_anim
- vn_center
- vn_farleft
- vn_farright
- vn_left
- vn_midleft
- vn_midright
- vn_right
- wait_fade

- yzoom_in

16.11.2 Variable Names

Note: This list is not exhaustive! If you are running into an error you think may be caused by conflicting variable names, try searching for `define var_name`, `image var_name`, `transform var_name`, `default var_name`, or `var_name =` to see if you come across any unexpected matches.

- all_characters
- all_guests
- answer
- available_calls
- bubble_list
- call_countdown
- call_history
- character_list
- chat_pause
- current_call
- email_list
- example_guest
- filler
- fullsizeCG
- heart_point_chars
- incoming_call
- in_phone_call
- ja
- ju
- m
- main_character
- new_cg
- novoice_chars
- r
- rainbow2
- ri
- s
- sa
- special_msg

- u
- unseen_calls
- v
- va
- y
- z

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search