sure

MYSTERIOUS MESSENGER USER GUIDE

Last Updated 2019-05-18

Table of Contents

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.

I'm sure you're eager to get started, so I'll keep this brief. To begin, I recommend taking a look at the table of contents and going to the section you're interested in. 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:

```
## Sample code here
```

Most of these code excerpts are taken directly from the program itself and can be copied in to use for your own test purposes. You'll also see some text boxes like the one on the right. These provide more insight into some of the calls and functions the program used.

chat_begin

chat_begin(background, clearchat=True, resetHP=True)

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

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

1. [...]

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.

Credits

I'd also like to take this space to thank some of the people that contributed to this program.

Sakekobomb for giving me permission to use their art in this program

Manami from saeran-sexual.tumblr.com for letting me use their art edits

mvngx.gif for contributing many of the assets used in the program

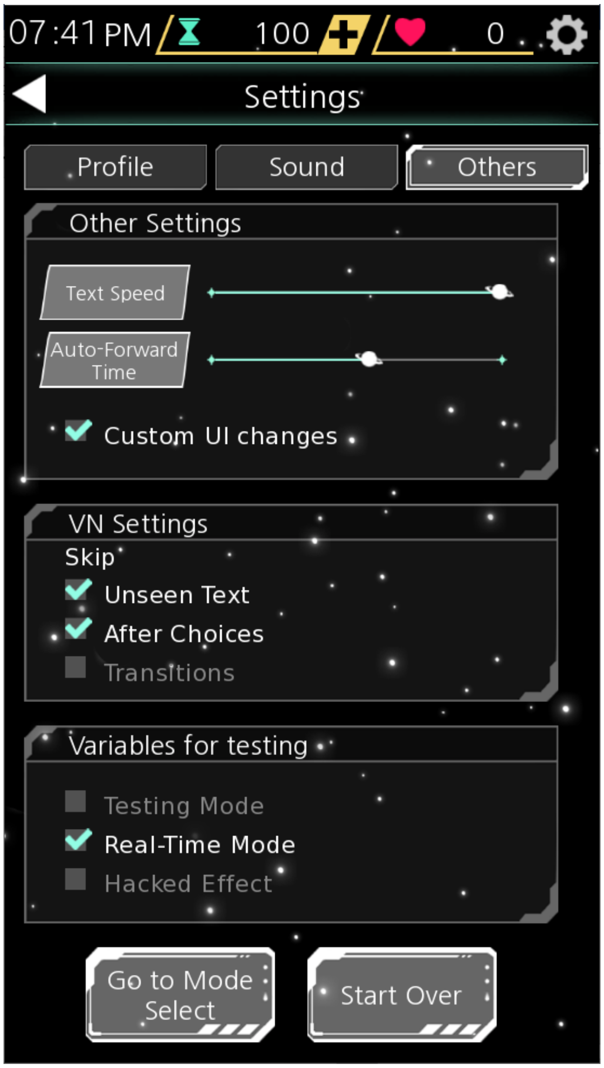
The people of the Lemma Soft forums and Ren'Py Discord for many tutorials, answers, and resources

RenpyTom for assistance in fixing some of the errors and animation issues I ran into

That's all from me. Enjoy the program!

Note: 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. They will be explained briefly below.



Other Settings

Text Speed

Auto-Forward Time

☑ Custom UI changes

VN Settings

Skip

☑ Unseen Text

☑ After Choices

□ Transitions

Variables for testing

□ Testing Mode

☑ Real-Time Mode

□ Hacked Effect

Text Speed - By default, this slider is set to maximum, which will make text appear instantaneously. However, if set to slower speeds, it will cause phone call text and Story Mode (VN) text to appear letter-by-letter at the desired speed.

Auto-Forward Time - This program includes an 'auto' feature for VN sections as well as phone calls. Setting this bar 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.

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

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. It remembers which text you have seen across playthroughs.

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

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

Testing Mode - If checked, this ensures that you never go into "observing" mode. In other words, if you replay a chatroom or Story Mode section you've already seen, you will be able to play it as if it's your first time playing that chatroom/VN section and won't be restricted to choosing options you've already seen. It should be turned off for a player, but is useful to have checked when testing your own code.

Real-Time Mode - This is explained further in 1.5 Expired chatrooms and real-time mode. It is often easier to leave this unchecked when testing so you're not waiting around for chatrooms to appear in real-time, though you can also buy 24 hours' worth of chatrooms at any time with no cost when real-time is active.

Hacked Effect - From here, you can toggle the **hacked** variable on and off as desired for testing purposes. This triggers some aesthetic modifications to the chatroom timeline screen in particular.

1. Chatrooms

1.1 Creating a Chatroom.

Example files to look at: **Example Chat.rpy**, **Coffee Chat.rpy**

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

1. Create a new .rpy file (optional, but recommended)
2. Create a label for the chatroom
   a. You may also want to fill this in to the **Script Generator.xlsx** spreadsheet so you can set up your route later (see 7. Setting up a route)
3. After the label, write **call chat_begin('noon')** where 'noon' is replaced by whatever time of day/background you want to use
4. Add a **play music yourmusic loop** where **yourmusic** is replaced by the desired music file/variable
5. Write the chatroom (you'll probably want to use **Script Generator.xlsx**)
6. End the chatroom with **jump chat_end**

The first thing you should do when creating a chatroom is to make a new .rpy file and name it something descriptive so you'll know where to find that chatroom. If you're planning on making a lot of chatrooms (say, for a route), I'd also recommend coming up with a consistent naming scheme such as "Day 1-Chatroom 3.rpy" or "Chatroom 1-3.rpy" so it's easier to find the correct files later on.

In your newly created .rpy file, start off by making a label:

```
label day1_1:
```

Don't forget about the colon after the label name. Your label name also can't have any spaces in it, nor can it begin with a number. Next, we'll begin the chat. Note that everything under the label **must be indented at least one level to the right.** Look at the example files mentioned above if you're not sure what this means. To begin the chat, we use the call:

```
call chat_begin("earlyMorn")
```

The text in the quotes tells the program what background to use. Your options are:

- morning
- noon
- evening
- night
- earlyMorn
- hack
- redhack
- redcrack

chat_begin

chat_begin(background, clearchat=True, resetHP=True)

Note that it *is* case-sensitive, so you need to get the capitals right. The **chat_begin** function will also clear the chatlog (aka your message history, so when you begin a new chatroom there are no messages on the screen at the beginning), but you can pass it a second argument so that it doesn't clear the chatlog. That looks like this:

```
call chat_begin("hack", False)
```

The other thing the **chat_begin** function does is resets the heart points you've gathered in chatrooms prior to that point. If you don't want it to do this, you need to pass it a third argument as False:

```
call chat_begin("hack", True, False)
```

Note that you *need* to have a second argument in order to stop the chatroom from resetting your heart points. That second argument tells the program to clear messages (True) or not (False) as described above.

Besides preventing the program from resetting heart points, if **resetHP** is **False**, the program will also assume you're calling **chat_begin** in the middle of a chatroom and thus don't want the participants reset. Otherwise, the list of participants shown at the top of the screen during a chatroom will also display people who have entered and then exited the chatroom. If you're not calling **chat_begin** in the middle of a chatroom, you usually want **resetHP** to be **True**.

So now that we've got the chat set up, we probably want to play some music. We can do that like so:

```
play music mystic_chat loop
```

Where **mystic_chat** can be replaced by the name of whatever music you want. There are several files already pre-defined in **variables.rpy** under the heading **BACKGROUND MUSIC DEFINITIONS**.

Finally, to end your chatroom, you need to write:

```
jump chat_end
```

To learn more about how to write dialogue for your chatroom, check out 1.3 Using the chatroom spreadsheet.

1.2 Useful Chatroom Functions

You can see many of these functions in use in **Example Chat.rpy**

1.2.1 How to let the player make a choice

```
call answer
menu:
    "Choice #1":
           m "Choice #1" (pauseVal=0)
        (more dialogue here)
    "Choice #2":
        (more dialogue here)
(regular dialogue continues here)
```

The main thing to remember is to write **call answer** before **menu:**, which will bring up the 'answer' button at the bottom of the screen and pause the chat.

You can add as many choices as you want to the menu, although only 5 options will fit on the screen at one time. Any dialogue that is indented after a choice will only be shown to the player if they pick that choice. Dialogue indented at the same level as **menu:** will be shown to players regardless of what option they picked in the menu. You can the TAB key to indent text an additional level to the right, though make sure your editor is using spaces to indent code (since Python isn't fond of actual TAB characters).

After a choice, you'll generally want the MC to have dialogue - if so, you can see that under **"Choice #1"** we've written another line. **m** is the variable we use to make MC speak. Unless the choice menu options are paraphrased, you need to copy the line of dialogue here for MC to say.

Secondly, there's also another argument after MC's dialogue: **(pauseVal=0)**. There's an option to have the spreadsheet automatically write this in for you (see 1.3 Using the chatroom spreadsheet). What it does is tell the program to not wait before posting this message. Otherwise, after choosing an answer, the program would pause for a moment to simulate "typing time" for the MC to send her message. Since this can be rather disorienting after a choice, adding **(pauseVal=0)** after the dialogue removes this wait time.

However, the MC doesn't need to send a message after a choice, or you can have other characters send things. For example, one of your choices could be **"(Remain silent)",** in which case MC probably won't send a message after the choice. (You can see an example of this in **plot branch.rpy**).

1.2.2 How to show a heart icon

```
call heart_icon(s)
```

Where **s** is the variable of the character whose heart point you'd like to show. The options built into the program are:

heart_icon

heart_icon(character, bad=False)

- **ja** (Jaehee)
- **ju** (Jumin)
- **r** (Ray/Saeran)
- **ri** (Rika)
- **s** (Seven)
- **sa** (Saeran/Ray)
- **u** (Unknown, a white heart point)
- **v** (V)
- **y** (Yoosung)
- **z** (Zen)

If you'd like to add your own character to give a heart point to, check out 6. Creating new characters. Both Ray and Saeran's heart points count towards the same character, Saeran.

There is also an optional second argument you can add to the **heart_icon** call:

```
call heart_icon(s, True)
```

The **True** argument tells the program that this heart is a "bad" heart icon - in other words, it indicates that this answer, while awarding the player a heart point, also counts towards a bad ending. In this way, you can conceal which answers lead to the "Good" end since you award the player a heart point for both the good and the bad ending answers. However, when you get to a plot branch, you can have the program calculate whether the player has more "good" or "bad" heart points with a character. See 7.3 Plot Branches for more information on this.

1.2.3 How to show the scrolled "hacking" effect

```
call hack
```

OR

```
call redhack
```

It's as simple as putting that where you'd like to use it.

1.2.4 How to show a banner

```
call banner('name of banner')
```

where 'name of banner' is replaced with the actual name e.g. **call banner('heart')** The available banners are:

- lightning
- heart
- annoy
- well

The names are case-sensitive when you call them.

1.2.5 How to make a character enter/exit the chatroom

To get the message "Character has entered the chatroom" type:

```
call enter(y)
```

where y is the variable of the character who is entering (see 1.2.2 How to show a heart icon for a list)

To get the message "Character has left the chatroom" type:

```
call exit(y)
```

where y is the variable of the character who is exiting.

1.2.6 How to update a character's profile picture

To update a character's profile picture, use

```
$ ja.set_prof_pic('Profile Pics/Jaehee/your-pic.jpg')
```

where **ja** is the character whose profile picture you'd like to change, and **'Profile Pics/Jaehee/your-pic.jpg'** is the path to the profile picture you'd like to update to. Profile pictures should be at minimum 110x110 pixels and at maximum 314x314 pixels for best resolution in the profile screen. The image should have equal height and width (i.e. be square).

You'll generally write this right at the beginning of a new chatroom, but you can also do it in the **after_** label if you like (See 7.2 Chatrooms with content afterwards for more information on the **after_** label).

However, updating MC's profile picture is handled differently. Currently the program has 5 default images that the player can cycle through by clicking on the profile picture in the Profile tab on the Settings screen. If you have a particular custom picture you'd like to use, you can drop it into the **game/Drop Your Profile Picture Here/** folder. The dimensions should be at least 110x110 pixels, though images that are up to 363x363 pixels will display better on the profile screen. It must be a square image to prevent warping. The program takes png, jpg, and gif files. You can add as many or as few pictures as you like, and are free to delete the existing images in that folder.

1.2.7 How to update a character's cover photo

To update a character's cover photo, use

```
$ ja.set_cover_pic('Cover Photos/Jaehee/your-pic.jpg')
```

where **ja** is the character whose cover photo you'd like to change, and **'Cover Photos/Jaehee/your-pic.jpg'** is the path to the cover photo you'd like to update to. Cover photos should be 750x672 pixels.

You'll generally write this right at the beginning of a new chatroom, but you can also do it in the **after_** label if you like (See 7.2 Chatrooms with content afterwards for more information on the **after_** label).

1.2.8 How to update a character's status

To update a character's status, use

```
$ ja.set_status("This is my new status update.")
```

where **ja** is the character whose cover photo you'd like to change, and **"This is my new status update"** is whatever you'd like to update their status to.

You'll generally write this right at the beginning of a new chatroom, but you can also do it in the **after_** label if you like (See 7.2 Chatrooms with content afterwards for more information on the **after_** label).

1.3 Using the chatroom spreadsheet

Included with the program is a spreadsheet called **Script Generator.xlsx**. The easiest way to write dialogue is using this spreadsheet. The first tab is called **Chatroom Instructions** and explains how the **CHATROOM TEMPLATE** tab is used. As much as possible, the spreadsheet will check off the correct boxes for you when needed and notify you when you've made a mistake like typing a character's name wrong.

The tab **Popcorn** also has examples right from the program of how the dialogue for that chatroom was written.

In general, you should create a copy of the **CHATROOM TEMPLATE** tab, then fill it in with your desired dialogue. Don't forget that messages such as "707 has entered the chatroom" are handled with function calls - see **1.2 Useful Chatroom Functions**.

Once you're happy with your chatroom and you want to try it out, look at 7. Setting up a route to see how to get the chatroom to show up in the program.

Note: Posting CGs in chatrooms

Example files to look at: **Example Chat.rpy**

One other thing you can do in chatrooms is have the characters post images that the player can click on to view full-size. These images will also be automatically unlocked in the Album once the player has seen them. To post an image, first you need to add it to the correct album in **gallery.rpy** just beneath the python declarations. There are two versions of each variable: a persistent version, which is a blank list (e.g. persistent.ja_album), and a regular version, which is where you'll declare your images. Each album is a list of **Album** objects like the following.

```
default common_album = [ Album("CGs/common_album/cg-1.png"),
                        Album("CGs/common_album/cg-2.png"),
                        Album("CGs/common_album/cg-3.png")]
```

The only required field is the first one, which is the path to the image. It should be 750x1334 pixels large. The **thumbnail** field is the image that will display in the album after that photo is unlocked. If you provide the path to your own image, it should be 155x155 pixels. If not, the program will crop and resize the CG to fit the thumbnail. **locked_img** is the thumbnail shown in the album if the image is not yet unlocked, and **unlocked** is True when the player has unlocked the image.

Album class

Album(img, thumbnail=False, locked_img='CGs/album_unlock.png', unlocked=False)

Next, to have a character post that image, you'll write down a simplified version of the image path in their dialogue column and check off the **Image** column. For example, if the path to your CG was **CGs/common_album/cg-1.png** then you will write **common/cg-1.png** in the dialogue column. **CGs/s_album/cg-1.jpg** becomes **s/cg-1.jpg** and so on. The program relies on having folders in the **CGs** folder that are named the same thing as the **_album** variables in **gallery.rpy**, so be careful when spelling file names. Checking the Image column in the spreadsheet will ensure the image will show up and will be clickable. Images unlocked in the album will persist across playthroughs. In this program, it is also possible for the MC to post images.

1.4 Creating an "Opening" chatroom

Example files to look at: **script.rpy**

When the player starts a new game, you may wish to have an "introductory" chatroom, to introduce the characters and story before the player begins the route. This should go under the **start** label, which is found in **script.rpy**. It works a bit differently than regular chatrooms, so **it's recommended you have a good grasp of how to create regular chatrooms before you look at this code** to modify it.

You generally won't need to modify any of the code until the line

```
call new_incoming_call(Phone_Call(u, 'n/a'))
```

In the default program, this line causes Unknown (**u**) to call the player immediately after they start a new route, before they get to the chat home screen. You can also include phone calls that trigger **after** the opening chatroom, but this call is included so you can see what it might look like if you wanted a phone call at the beginning.

What follows is almost the same as any regular phone call; it begins with **call phone_begin** but ends with **call phone_end** instead of **jump call_end**. The **call** instead of **jump** means that we can continue writing the introduction instead of ending the introduction right there.

Next, you'll see the line

```
scene bg black
```

which is optional, but in this case it cleans up the transition between the phone call and the chatroom.

Now, we begin a chatroom section like we would usually:

```
call chat_begin('hack')
```

You can add any of the usual chatroom functions to this, including the hack effects, background music, and heart functions, to name a few. Fill out this chatroom exactly as you'd fill out any other chatroom, including

```
jump chat_end
```

to finish off the label.

If you want to have characters send text messages after the introductory chat, or you'd like to make phone calls available or trigger an incoming call, it's taken care of the same way as usual, except the label you'll be using is called **starter_chat**, not **start**. So incoming calls should look like

```
label starter_chat_incoming_ja:
```

and post-chatroom things (including text messages) will be taken care of in the

```
label after_starter_chat:
```

label. Note that the introductory chatroom **does not** support separate VN modes after the introductory chatroom, but you **can**, however, include calls to VN sections in the **start** label itself.

Note that you must always end the introductory chatroom with **jump chat_end** OR, if you want to go more advanced, you **must** ensure the end of your introduction sets the variables **starter_story = False**, **persistent.on_route = True**, and then you must call some form of the label **press_save_and_exit(phone)** where **phone** is True by default if you're ending a chatroom section and should be False if you're ending a VN section.

Including VN sections in the introductory chat has not currently been tested, so if you would like to include them be aware that it will require more knowledge of the program's structure than usual.

1.5 Expired chatrooms and real-time mode

Example files to look at: **Coffee Chat.rpy**, **popcorn.rpy**

In the program, you can switch between two different modes: real-time, and sequential. Sequential mode is the default, and you can choose to toggle real-time mode on from the **Others** tab in the **Settings** menu.

In sequential mode, chatrooms unlock sequentially. In other words, once you finish a chatroom, the next one will automatically unlock. Chatrooms don't expire unless you back out of them, and you can proceed through chatrooms regardless of what the current time is.

In real-time mode, chatrooms unlock based on the current time. You'll also have the option to buy the next 24 hours' worth of chatrooms in advance. If an old chatroom 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 chatroom (though you can usually call the characters back).

Each chatroom you create should have both a "regular" version and an "expired" version. The expired version is the one the player will play through if the chatroom has expired and they have not bought it back. Generally this means the player will not participate in the chatroom.

To create the expired chatroom, simply take the name of the regular chatroom and add **_expired**. So, if your chatroom is called

```
label mychat:
```

then the expired chatroom should have the label

```
label mychat_expired:
```

The rest can be filled out as any other chatroom.

Note: backing out of chatrooms vs real-time expiry

There are two different ways for chatrooms to expire: first, chatrooms expire if you're playing in real-time and miss a chatroom before the next one triggers. Second, chatrooms expire if you use the back arrow during a chat you haven't seen before.

In the first case (expiry due to real-time mode being active), the following will happen:

- You will receive a missed call from a character who was going to call you after the expired chatroom, and that phone call will be turned into an 'outgoing' call so you can call the character back to receive that conversation
- Any text messages that would have been delivered after the chatroom (or VN) will be automatically delivered to your inbox

- Any outgoing calls that were to be made available after the chatroom (or VN) will be made available

Note that phone calls will "time out" two chatrooms after they were set to appear. So, for example, say you have 3 chatrooms A, B, and C, and Bob 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'll receive a missed phone call from Bob. You can call Bob back to receive this phone call up until chatroom C becomes available, at which point the phone call will become unavailable and you won't be able to call Bob to get that conversation anymore.

In the second case, where the player backs out of an active chatroom and causes it to expire, the following will happen:

- Any incoming calls that would have been triggered after the chatroom are instead turned into outgoing calls, though the player receives no missed call notification
- Any text messages that would have been delivered after the chatroom (or VN) will be automatically delivered to your inbox
- Any outgoing calls that were to be made available after the chatroom (or VN) 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 won't receive a missed call notification for it.

1.6 Timed Menus

Example files to look at: **timed menus.rpy**

Timed menus are a feature unique to this program. If used, the answer button will appear for a specific period of time before disappearing, and the chat will continue on while the answer button is displayed at the bottom. The time the player has left to answer is shown above the answer bar as a spaceship moving towards the right side of the screen. When it reaches the right side, the answer button will disappear.

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

- **The time the player has to press the answer button changes depending on how fast they have the chatroom speed set to.** If, when writing the menu, you give the player 10 seconds to reply, the answer button will stay on the screen for 10 seconds when the player has the chatroom speed set to 5, but it will stay on the screen for 5.5 seconds if the player has the chatroom speed set to 9, and 14.5 seconds for a player with chatroom speed set to 1.

  - This is so that players who have a slower reading speed will still be able to read the same number of messages as a player with a faster reading speed before the answer button times out

- **If the player is on Max Speed, it will skip the timed menu entirely,** with no opportunity for the player to press the answer button
- **As soon as the player hits the answer button, the chat will stop until they choose an answer.** For example, if the player has 10 seconds to answer and presses the answer button after 5 seconds, the answer choices that appear will not go away even after the ten seconds are up.
- **A player who selects the answer button before the time is up will miss any messages that they might have seen if they waited to press the answer button.** For example, if the timer is 10 seconds long and the player presses the answer button after 5 seconds, they will be unable to see messages from the 5-10 second period before the timer expired, since they pressed the answer button only 5 seconds in.
- If the player replays a chatroom with timed menus, the answer button will not appear/all answers will "time out". This is due to possible replay issues in the event that a player has never seen the menu options before.

1.6.1 How to write a timed menu

Writing timed menus works very similarly to regular menus. Whenever you'd like the answer button to become available, include the line

```
call continue_answer("menu1", 8)
```

where **"menu1"** is the name of the menu to jump to when the player presses the answer button, and **8** is the number of seconds to show the answer button to a player with the chatroom speed set to 5. It may take some trial and error to determine how long you should display the answer button for/how many messages the player might see before answering. If you don't provide the number of seconds for the answer button to remain on-screen, it will default to 5 seconds.

After calling **continue_answer**, you should write more dialogue that will continue to show until the player hits the answer button. In most cases, the time it takes to display this dialogue should be the number of seconds you show the answer button for.

Next, we write the menu. To use timed menus, we need to give each menu a name so we can jump to it when/if the player hits the answer button.

```
if timed_choose:
    menu menu1:
        "Choice 1":
            m "Your dialogue here."
        "Choice 2":
            m "Your dialogue here."
else:
     s "Optional dialogue here."
```

The main things to note are that you must first write **if timed choose** before your menu, and you must name the menu whatever you called it earlier.

Including **if timed choose** ensures that if the player doesn't answer before the timer runs out, then they won't see the menu. Optionally, you can include an **else** statement like you see in the above example. Dialogue indented under the **else** statement will **only** be seen by a player who didn't click on the answer button and let the menu time out.

Unlike for regular menus, the dialogue immediately after a choice usually doesn't need the argument **(pauseVal=0)**. For ordinary menus this reduces the delay between when the player selects a choice and when it displays in the chat, but because timed menus allow the player to interrupt the flow of dialogue, it feels more natural for there to be a delay between selecting an answer and having it display in chat.

Otherwise, timed menus work almost the same as regular menus, and can be nested as well. See 1.2.1 How to let the player make a choice for more information on regular menus.

2. Emails

2.1 Writing an email chain

Example files to look at: **email test.rpy**

First, go to **email test.rpy** and scroll to the bottom, where there is a template email guest. Copy this code and follow the instructions given to create your email chain. There is a guest used in the program that is defined above the email template as well, so you can see how that guest works.

The main thing you should be careful to do is make sure you name your reply labels correctly. The labels are case-sensitive, and if you make a mistake, the program will run into an error when you try to reply to the email.

2.2 Inviting a guest

This is extremely straightforward; after you've defined your guest as per 2.1 Writing an email chain, in the chatroom or VN section that you'd like to invite your guest in, simply use the call:

```
call invite(YourGuest)
```

where **YourGuest** is the variable that you made when you defined the guest using

```
default YourGuest = Guest("test", "thumbnail.png", [...])
```

After the chatroom/VN section, you will receive the first email from that guest.

2.3 Overview on how emails work

After you define your guest and invite them to the party, they will immediately send you an initial email. From this point onwards, if you **do not** reply to the email, after every chatroom a variable called **timeout_count** will decrease by one. By default, it starts at 25. This number resets back to 25 as soon as you reply to the email. However, if the timer reaches 0 (aka you've gone through 25 chatrooms/VN sections since you first received the email), the email will be considered **timed out** and can no longer be replied to. If you'd like to change this number, take a look at the **Email** class under **email.rpy**.

If you **do** reply to the email, the program will calculate when it should return the guest's reply to you. For example, if there are 30 chatrooms left to be played through, and this is the first email reply in the chain (there are up to three total replies in a successful email chain), the program will determine that the maximum number of chatrooms it can wait before delivering the reply is 30/3 or 10 chatrooms. It will then generate a number between 10 and 10-7=3, and that number will be the number of chatrooms the program will wait before delivering the guest's reply to your email inbox. This number is stored in the variable **deliver_reply** and is decreased by 1 after every completed chatroom or VN.

For testing purposes, there is also a variable called **test** in the function **set_reply** found in the **email.rpy** file. If **test = True**, then instead of generating a random number based on the number of remaining chatrooms, the program will instead just generate a number between 5 and 10. This can be useful if you want to test out emails, but have a very small number of chatrooms created thus far.

3. Phone Calls

3.1 How to trigger an incoming call after a chatroom

Example files to look at: **Coffee Chat.rpy**

To trigger an incoming call, there's a specific naming convention for the label. If your chatroom is called

```
label my_chatroom:
```

then to trigger the incoming call, you need to create a label like

```
label my_chatroom_incoming_ja:
```

where **ja** is the variable of the character who's calling you (see section 1.2.2 How to show a heart icon or **character definitions.rpy** for a list of the existing characters).

Note that you can only have **one** incoming call after a chatroom (so if you define two labels like **label my_chatroom_incoming_ja** and **label my_chatroom_incoming_ju**, then only one of them will show up as an incoming call).

You can then use that label to write your phone call as outlined below. After the chatroom is finished, you will get an incoming call from that character.

3.2 How to write a phone call

Example files to look at: **Coffee Chat.rpy**

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

1. Create a label + the correct suffix for the phone call
    a. **my_chatroom_phone_ja** to make an outgoing call to the character **ja** available after the **my_chatroom** chatroom
    b. **my_chatroom_incoming_ja** to trigger an incoming call from the character **ja** after

the **my_chatroom** chatroom

2. At the beginning of your new label, write **call phone_begin**
3. Write the phone call dialogue using the character's "phone" version e.g. **ja_phone**
4. End the phone call with **jump phone_end**

Aside from incoming calls (see above), all outgoing calls follow the same naming convention. If your chatroom is called

```
label my_chatroom:
```

then any outgoing calls you want to make available will be called

```
label my_chatroom_phone_ja:
```

where **ja** is the variable of the character whom you're calling (see section [1.2.2 How to show a heart icon]() or **character definitions.rpy** for a list of the existing characters).

If the program finds the correct label, after the player is finished with the chatroom, these calls will be made available (aka the user can go to the phone menu and call the character to go through this phone call). You can make one phone call available for every character, **regardless** of whether or not that character also has an incoming call. In the case that the player misses the incoming call from that character, **both** the incoming **and** the outgoing calls will be made available so you can call the character back twice and get both conversations.

Underneath your phone call label, you should start off the conversation by typing

```
call phone_begin
```

Then you can use the "phone" version of the character to write dialogue in the same way as you would for a VN section. In general, all characters' "phone" versions are their original variable + _phone. E.g. ja (Jaehee) becomes ja_phone and m (MC) becomes m_phone

All dialogue will generally look the same between characters other than MC, but having different characters defined also lets the player switch off voice for certain characters if they don't wish to hear a particular character's voice acting. (Currently, there is very little voice acting in the program; for an example of how to add voice acting to your phone calls, see Zen's incoming phone call in **Coffee Chat.rpy**).

You may also find Ren'Py's "monologue mode" helpful, since unlike in VN mode you won't be switching between expressions or different speakers very often. See **tutorial_chat_phone_y** in the file **Coffee Chat.rpy** for an example of this.

At the end of your phone call label, type

```
jump phone_end
```

to finish up the phone call.

3.3 How to change a character's voicemail

Example files to look at: **Coffee Chat.rpy, phone screen.rpy**

Voicemail is part of a character's definition. 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.

To update a character's voicemail, simply type

```
ja.update_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 call is.

In general, it's a good idea to define new voicemails in **phone screen.rpy** at the bottom of the file. They are defined the same way as a regular phone call, although there are no restrictions on what the label should be named (but it's recommended you pick a descriptive name for it, probably including the word 'voicemail' to make it easier to understand).

If you'd like to write a 'generic' voicemail message, there is a character named **vmail_phone** that you can use to write the dialogue. Otherwise, you can use the characters' regular phone versions (e.g. **ja_phone**) to write the dialogue for their voicemail messages.

4. Text Messages

4.1 Regular text messages

There are two kinds of text messaging styles available in the 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", is unique to this program and plays out similar to a chatroom. The player receives a text message, and upon entering the conversation, cannot leave until the conversation is either over or the player ends the conversation themselves. For more on regular text messages, keep reading. For more on real-time text conversations, see 4.2 Real-time text conversations.

4.1.1 How to write text messages

Example files to look at: **Coffee Chat.rpy, text_msg_test.rpy**

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

1. Create a label using the prefix **after_** + the name of the chatroom you want to send the text messages after e.g. **label after_my_chatroom**
2. Fill out your dialogue using the **TEXT TEMPLATE** tab of **Script Generator.xlsx**
3. Copy the **What should be filled into the program** column into your new label
    a. Note that you can have as many characters text the MC after the chatroom as you like, all under the same label
4. If you want the player to be able to reply, write **$ add_reply_label(s, 'my_reply')** where **s** is the name of the character whose conversation you're replying to and **'my reply'** is the name of the label to jump to for the reply
5. End the **after_my_chatroom** label with the line **return**
6. Create a label for the reply (e.g. **label my_reply**) and add a menu + any text messages for the MC/other character to send
    a. (Optional) Award the player a heart point for a liked response with the line **$ add_heart(s)**
    b. (Optional) Award the player a heart point for a liked response with a character *other* than the person whom the MC is texting with the line **$ add_heart(s, y)** (gives the player a heart point from Yoosung in a text conversation with Seven)
7. Finish the reply label with **jump text_end**

To have characters text you after a chatroom, the program looks for a label with a specific naming convention. For example, if your chatroom is called

```
label my_chatroom:
```

then you need to create a label called

```
label after_my_chatroom:
```

Next, go to **Script Generator.xlsx**. There is a tab called **TEXT TEMPLATE**, and another called **Example Text**. The easiest way to write text messages will be to use this. Much like the **CHATROOM TEMPLATE**, you'll fill in the name of the speaking character and dialogue in the correct columns. You'll also notice that there are columns to check off different fonts like there are for chatrooms - this is a feature unique to this program. The default font is the first one, **Sans Serif 1**.

There is also an **Image?** column. In the same way as you show images and emojis in chatroom sections, you can show them in text message conversations as well. See Posting CGs in chatrooms to understand how to fill out the spreadsheet, and see **text_msg_test.rpy** for examples of both CGs and emojis being posted in a text message. **Both** MC and the other characters can post either CGs or emojis.

Filling in this spreadsheet will be almost identical to writing a chatroom, so you can look at the tab **Chatroom Instructions** in the spreadsheet, as well as **1.3 Using the chatroom spreadsheet**. The main difference is in the **Sender** column.

In general, this column will be automatically filled out for you. This column indicates whom the player is in a text message conversation with. Usually this will be the same as the **Name of Character** column, unless the MC is the character sending the message. In this case, the spreadsheet will assume the name of the character whose dialogue is in the row above the MC's dialogue is the sender.

In other words, this just means that for Seven and MC to send text messages to each other, the **Sender** column will always be **s**. You usually won't have to worry about this column anyway, as the spreadsheet should fill it out correctly for you.

After you've written your dialogue, you should copy the contents of the column **What should be filled into the program** and paste it into your label. Unlike phone calls, you can add every character's text message under this one label. At the end of your **after_** label, write

```
return
```

to end the label.

4.1.2 Replying to text messages

Example files to look at: **Coffee Chat.rpy**, **text_msg_test.rpy**

If you want the user to be able to reply to the text messages they receive, in the **after_** label you created (see 4.1.1 How to write text messages), you need to also write

```
$ add_reply_label(ja, 'my_text_reply1')
```

where **ja** is the variable of the character whose text message you're replying to (see section 1.2.2 How to show a heart icon or **character definitions.rpy** for a list of the existing characters), and **my_text_reply1** is the name of the label that contains the menu of replies to that text message. There are no naming restrictions on what to call this label, but I recommend you come up with a naming convention of your own so that you don't accidentally create several labels with the same name.

Create a new label for the reply, as you named it above:

```
label my_text_reply1:
```

Then, create a menu and add your text messages below as seen in **text_msg_test.rpy**.

When you are done adding your text messages, at the same indentation level as **menu:**, write

```
jump text_end
```

Note that if you'd like the player to be able to continue to reply to the text message, you can include another **$ add_reply_label(ja, 'my_text_reply2')** within the first reply label.

4.1.3 Giving heart points during text messages

Example files to look at: **Coffee Chat.rpy, text_msg_test.rpy**

add_heart

add_heart(textmsg_character, other_heart_character=False)

Similar to chatrooms, you can award the player heart points for picking certain responses during a text message conversation. In general, after the reply that triggers the heart point, write

```
$ add_heart(ja)
```

where **ja** is the variable of the character whose text message you're replying to (see section [1.2.2 How to show a heart icon](#) or **character definitions.rpy** for a list of the existing characters).

This will tell the program to award the player a heart point when they next open the message. In general, this heart point will belong to the person the player is in a text conversation with. However, if you'd like the player to receive a heart point from a different character, you need to pass an additional argument

```
$ add_heart(ja, ju)
```

where **ju** is the variable of the character whose heart point you'd like to award the player. See the text messages in **Coffee Chat.rpy** for an example of this.

You can only give the player one heart point per text message reply (i.e. the player can't receive heart points from more than one person at a time).

4.1.4 How text messages are delivered

When you write text messages using the **$ add_text** function, those conversations are added to a queue which is slowly delivered to the player after the chatroom/VN is complete. So, if you wrote up text messages to be sent from Character A, B, and C, the player will receive Character A's message immediately upon returning to the 'home' screen, and will receive Character B and C's messages either by waiting around on the home screen or by moving through the menu screens.

Similarly, if the player has replied to Character A's message, Character A's response will be put into the queue behind any other undelivered messages and delivered as the player moves around the menu screens or as they wait.

All text messages are immediately delivered as soon as the player clicks on a Day to begin a new chatroom.

In this program, there is no time limit on when you can or can't reply to messages; however, if a character sends you a new text message and you haven't yet replied to a previous message that they sent to you, you will no longer be able to reply to the older text message.

4.2 Real-time text conversations

This style of text messaging is similar to a chatroom with a single character. In order to use real-time texting, you first need to check the **Real-Time Texts** option in the **Others** tab on the settings menu. This will turn **all** of your text message conversations into real-time conversations. There currently is no way to mix-and-match the two texting styles.

4.2.1 Beginning a text conversation after a chatroom

Example files to look at: **text_msg_test.rpy**

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

1. Create a label using the prefix **after_** + the name of the chatroom you want to send the text messages after e.g. **label after_my_chatroom**
2. Write a conditional statement **if persistent.instant_texting:** to nest your messages under
3. Begin with the line **call inst_text_begin(r)** where **r** is the name of the character who is sending the text message
4. Fill out your dialogue using the **CHATROOM TEMPLATE** tab of the Script Generator spreadsheet (*not* the **TEXT TEMPLATE** tab)
5. Copy and paste the dialogue from the **What should be filled into the program** tab and make adjustments like you would in a chatroom (See 1.3 Using the chatroom spreadsheet)
    a. Note that you **cannot** include menus, display heart icons, etc. in the initial message. Any dialogue you include here will be already shown to the player when they enter the conversation and will show up in a notification.
6. If you want the player to be able to continue the conversation, write **$ r.update_text('my_reply')** where **r** is the name of the character whose conversation you're replying to and **'my reply'** is the name of the label to jump to for the reply
7. End the initial conversation with **call inst_text_end**

8. End the **after_my_chatroom** label with the line **return**
9. See 4.2.2 Writing a text conversation for more on how to let the player continue the conversation

To have characters begin a text conversation with you after a chatroom, much like regular text messages, you need to have a label that follows a specific naming convention. For example, if your chatroom is called

```
label my_chatroom:
```

then you need to create a label called

```
label after_my_chatroom:
```

Now you need to include a conditional statement and nest your next lines of code under this block. It will look like this:

```
if persistent.instant_texting:
    call inst_text_begin(r)
    # Your dialogue here
```

where **r** is the name of the character who is sending the text message and **# Your dialogue here** is filled out as explained below.

Next, go to **Script Generator.xlsx**. There is a tab called **CHATROOM TEMPLATE** which is used to write both chatroom sections as well as real-time text conversations. The dialogue that you write in the **after_** label is the dialogue that will show up already-typed in the text message conversation, and will be previewed in a notification popup after the chatroom. Therefore you should keep the dialogue here to only a couple of lines at most.

Any dialogue you write to begin the text conversation cannot have menus, banners, heart icons, etc. It can, however, include emojis and images. See 1.3 Using the chatroom spreadsheet for more information on filling out the spreadsheet.

Fill out your dialogue underneath the conditional statement (where the **# Your dialogue here** line is). If you'd like the conversation to continue past these messages, you need to include the line

```
$ r.update_text('my_reply')
```

where **r** is the name of the character whose conversation you're replying to and **'my reply'** is the name of the label to jump to for the reply

Finally, end the initial conversation with the line

```
call inst_text_end
```

If you would like to initiate more than one text message conversation, you can repeat this process, beginning with writing **call inst_text_begin(v)** where **v** is the new character you'd like to write a conversation for.

Finally, end the **after_my_chatroom** label with

```
return
```

This should be at the same indentation level as your initial if statement - that is, it should only be one level indented under your **after_** label.

If you'd like the player to be able to hold a conversation with the character after they read their initial text message, see the next section 4.2.2 Writing a text conversation.

4.2.2. Writing a text conversation

Example files to look at: **text_msg_test.rpy**

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

1. Create a label pertaining to the label you previously defined in this initial text message setup (e.g. if you wrote **$ r.update_text('my_reply')** now you need a **label my_reply:**)
2. Begin the conversation with **call text_begin(r)** where **r** is the name of the character who is sending the text message
3. Fill out your dialogue using the **CHATROOM TEMPLATE** tab of the Script Generator spreadsheet (*not* the **TEXT TEMPLATE** tab)
4. Copy and paste the dialogue from the **What should be filled into the program** tab and make adjustments like you would in a chatroom (See 1.2 Useful Chatroom Functions

and 1.3 Using the chatroom spreadsheet)

   a. Note that unlike in the initial text message, here you can use functions like banners, heart icons, and menus. You should also preface menus with **call answer** as usual

5. End the text conversation with **jump text_end**

In the previous step, to initiate a text message conversation, you wrote a line like **$ r.update_text('my_reply')** where **r** is the character the player is having a conversation with. Now you need to create that label, so:

```
label my_reply:
```

There are no restrictions on what you can call this label, but I recommend you come up with a naming convention of your own so that you don't accidentally create several labels with the same name.

Now you should begin the conversation with

```
call text_begin(r)
```

where r is the name of the character who's sending the text message.

You can fill out your dialogue using the **CHATROOM TEMPLATE** tab of the Script Generator spreadsheet (*not* the **TEXT TEMPLATE** tab) in much the same way as you would write a chatroom. You can also refer to 1.2 Useful Chatroom Functions and 1.3 Using the chatroom spreadsheet for more information on the functions etc. available to you. Note that in this conversation, you're free to use functions normally available in chatrooms, such as banners, heart icons, and menus.

Unlike in regular text messaging, in a real-time text conversation you can include multiple menus and replies just as you would a chatroom conversation. Note, however, that there is no "expired" equivalent for real-time text conversations, so if the player chooses to back out of the conversation, they may not view it in its entirety and you will have no other way of conveying this information. Therefore I recommend you use real-time texting for non plot-essential information, or at least ensure that you have another way of explaining this information to the player outside of the real-time text conversation.

Once you've written up your dialogue and formatted it correctly, end the conversation with

```
jump text_end
```

4.2.3 Backing out of real-time text conversations

Unlike regular text messages, real-time text conversations function more like chatrooms. Thus, if you click the "back" button during a real-time conversation, you will be asked if you'd like to end the conversation. **Backing out of a real-time conversation means that that conversation will no longer be available to continue**, though you will retain any heart points or CGs you collected during the conversation. There is no option to "replay" text message conversations, but unlike chatrooms they will not 'expire' until another text message conversation overwrites them.

5. Visual Novel Mode

5.1 Setting up a VN mode section

Example files to look at: **VN tutorial.rpy, popcorn.rpy**

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

1. Create a label for your VN (e.g. **label my_vn**)
2. In your VN label, write **call vn_begin**
3. (Optional) Add music to your VN with **play music your_music loop**
4. Set up the background of your VN with **scene your_bg**
    a. (Optional) Use transitions like with fade (e.g. **scene your_bg with fade**)
    b. (Optional) Write **pause** after your scene statement to give the player a moment to look at the background
5. Fill in your VN mode with dialogue, characters, etc
6. Finish the VN mode with **jump vn_end**

First, much like creating a chatroom, you must create a label for your VN. This can be anything you like, though it's a good idea to call it something similar to whatever you called the chatroom before it. You may want to include the VN in the same file as the chatroom preceding it, though you can also put it in its own separate **.rpy** file.

```
label my_vn:
```

Now we'll begin the VN.

```
call vn_begin
```

This sets some important variables before the VN. Next, you can set the background for your VN. It begins as black. Use

```
scene bg rika_apartment with fade
```

where **bg rika_apartment** is a pre-defined variable you can find in **VN Mode.rpy** under the header **Backgrounds**, or you can define an image yourself. Backgrounds should be 750x1334 pixels. **with fade** indicates that the background should fade in from black.

If you'd like the user to have a moment to look at the background before you move on, you can write

```
pause
```

beneath your **scene** statement.

When you write a menu, before the options you should include the line **extend ''** e.g.

```
menu:
    extend ''
    "Your choice here.":
        m_vn "Your choice here."
```

This will show the previous line of dialogue under the choice menu so the player can read it. See the example files mentioned above for an example of this.

You can also add heart points in the same way as you add them in chatrooms; see [1.2.2 How to show a heart icon].

Finally, to end your VN section, write

```
jump vn_end
```

to show the player the Sign screen and return them to the main menu.

5.2 Adding music and SFX to VNs

Example files to look at: **VN tutorial.rpy, popcorn.rpy**

Much like chatrooms, music can be played via

```
play music mystic_chat loop
```

Where **mystic_chat** can be replaced by the name of whatever music you want. There are several files already pre-defined in **variables.rpy** under the heading **BACKGROUND MUSIC DEFINITIONS**. You should usually include **loop** after the music title so the song will repeat once it ends.

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. Some are already defined for you in **VN Mode.rpy**.

5.3 Moving characters around

Example files to look at: **VN tutorial.rpy, popcorn.rpy**

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 Mode.rpy** under **Transforms/VN Positions**.

vn_center is a unique position because it moves the character closer to the screen as well as placing them in the center. It's often used to imply the character is talking directly to you. However, if you wish to move a character currently in **vn_center** position to another position, you need to **hide** the character first. See **VN tutorial.rpy** for an example of how to do this.

Now that we know the different positions, we simply need to write

```
show jumin front at vn_right
```

where **jumin front** is the name/position of the character we'd like to show, and **vn_right** is the position we'd like to show them in. We can also add transitions like so:

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

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

5.4 Changing characters' outfits and expressions

Example files to look at: **VN tutorial.rpy, popcorn.rpy, character definitions.rpy**

To show a character, you should look at **character definitions.rpy** and find the character you'd like to show under the **Character VN Expressions Cheat Sheet** header. Any major character who does **not** have a front and a side position (Jaehee, Rika, Saeran, Yoosung), will have **vn** after their name e.g. **jaehee vn**. Characters with both side and front poses will have that appended to their name e.g. **zen side** or **zen front**.

If you'd like the character to wear their glasses for an expression, the first time you show them (or when you show them again after they've been hidden), you need to add **glasses** to their show statement, e.g.

```
show jaehee vn glasses happy
```

where **jaehee vn** is the name (and possibly position) of the character, **glasses** indicates you'd like them to be wearing their glasses, and **happy** is the expression. A similar expression for a character who has a side position is:

```
show v side angry glasses
```

where **v side** is the character + position you're showing, **angry** is the expression, and **glasses** indicates he should be wearing his sunglasses.

To change the character's outfit, the first time you show them (or when you show them again after they've been hidden) you need to add the appropriate outfit to their show statement e.g.

```
show saeran vn suit thinking
```

where **saeran vn** is the name (+ position, if applicable) of the character you're showing, **suit** is the keyword of the outfit the character is wearing, and **thinking** is the expression you want.

Note that attributes can be listed in any order after the character, so **show saeran vn thinking suit** is equally correct. Characters with other accessories (glasses, hoods, masks) can include those keywords as well, e.g.

```
show rika vn dress mask worried at vn_right with easeinright
```

This will show **Rika**, wearing her **dress** outfit, with the **mask** accessory, and a **worried** expression, at the position **vn_right** after being "eased in" from the right side of the screen.

You can play through the Example VN section on Tutorial Day to look at all the characters' available outfits and expressions. You can also refer to **character definitions.rpy** and find the character you'd like to show under the **Character VN Expressions Cheat Sheet** header, as mentioned above.

6. Creating new characters

6.1 Adding a new character to chatrooms

First, go to **character definitions.rpy**.

Under the header **Chatroom Characters**, you'll see several characters already defined. Copy one of the existing characters and replace the variables with your own variables e.g.

```
default b = Chat("Bob", 'b', 'Profile Pics/Bob/bob1.png', 'Profile
Pics/b_chat.png', '#f995f1', "Cover Photos/bob_cover.png", "Bob's
Status", '#ffddfc','#d856cd')
```

This defines a character whose name in the chatroom will show up as "Bob". His "file_id" is b, so if we want to give Bob any speech bubbles, the program will look for them under **game/images/Bubble/b-Bubble.png** or **game/images/Bubble/b-Glow.png**. You can also define special speech bubbles for Bob, which the program will look for under **game/images/Bubble/Special/b_cloud_l.png** etc. Alternatively, you can use the arguments **bubble_color** and **glow_color** (described below) to automatically colourize Bob's regular and glowing speech bubbles. Special speech bubbles will still need to be added as described above.

Defining a Chat Character

Chat(name, file_id, prof_pic, participant_pic, heart_color, cover_pic, status, bubble_color=False, glow_color=False, emote_list=False, voicemail=False)

'Profile Pics/Bob/bob1.png' tells the program where to look for Bob's profile picture. This should be the path to an image you've included. Similarly, **'Profile Pics/b_chat.png'** is Bob's "participant picture", or the picture that shows up on the Day timeline to show that Bob was present in a chatroom. You can take a look at the other characters' pictures to get an idea of what they look like.

Next, **'#f995f1'** is Bob's heart colour. The program will automatically generate a heart with that colour and award the player a point for Bob whenever we call the heart icon, like **call heart_icon(b)**.

"Cover Photos/bob_cover.png" tells the program where to look for Bob's cover photo. This should be the path to an image you've included. You may want to create a separate folder for the character's cover photos.

"Bob's Status" can be anything you like - it's Bob's initial status when you look at his profile page.

'#ffddfc' is an optional argument which will provide the background colour of Bob's regular speech bubble. It is defined in hexadecimal. If you don't define a colour here, you need to have an image in **game/images/Bubble/** called **b-Bubble.png** for the program to use.

'#d856cd' is another optional argument that provides the glowing colour on Bob's glowing speech bubble. It is defined in hexadecimal. If you don't define a colour here, you need to have an image in **game/images/Bubble/** called **b-Glow.png** for the program to use.

There is another optional variable after Bob's glow_color, called "emoji_list". This is used for chatroom creation, though it isn't fully implemented yet. You can see the variables the other characters' definitions refer to in **emojis.rpy**, though you can generally just ignore this like we've done above or write "False" for this field.

There is also an optional final variable, called "voicemail". If you like, you can set this up as a **Phone_Call** object, but the program will assign your character a default voicemail if you don't. You can always change their voicemail later; see 3.3 How to change a character's voicemail.

Finally, after you've created your character, you should also add them to the **character_list**, which can be found under the definitions for the Chatroom Characters. Just put a comma after the last character and add your character's variable, in this case, **b**. This means that Bob's profile will now show up alongside the other main characters and can be clicked on in the main hub screen.

Bob's dialogue can now be written like any other character in a chatroom, e.g.

```
b "How's it going?"
```

will now show Bob writing a message in the chatroom.

6.2 Adding a new VN mode character

First, go to **character definitions.rpy**.

Under the header **Visual Novel Mode**, you'll see several characters already defined. Copy the **narrator**'s definition and replace **None** with the character's name, in quotes. You can see the definitions of **sarah_vn** and **chief_vn** for examples of this.

If you'd like to change the background of the character's dialogue window, look for **window_background="VN Mode/Chat Bubbles/vnmode_9.png"** in your character's definition and replace **"VN Mode/Chat Bubbles/vnmode_9.png"** with the background you'd like them to use.

You can also replace **who_color="#fff5ca"** with a different colour. This is the colour of the character's name above their dialogue when they speak.

Note on voiced characters:

When creating a VN or a phone call character, you'll notice they also have a **voice_tag** variable. In most cases you can leave this as "other_voice", but if you'd like to include voiced lines for the character then you should allow the player to toggle your character's voice on or off. If our character's name is Bob like we've used above, we can include the line

```
voice_tag="bob_voice"
```

and then in **menu screen.rpy** under the **preferences()** screen we'll include Bob under Rika's definition, like so:

```
text "Rika" style "settings_style"
text "Bob" style "settings_style"
```

We'll also include a voice button for him:

```
use voice_buttons('rika')
use voice_buttons('bob')
```

This will include a toggle for Bob's voice so the player can choose whether to have his lines muted or not.

The above instructions will allow you to create a speaking character for VN mode. However, if you'd also like to include images of a new character, you need to define a **layeredimage** for them. Below the definition for the VN characters are all of the existing characters' layered images. Ren'py also has documentation on the layered image feature to help you.

I recommend that you keep any accessories that a character has separate from their expressions/outfits/etc. if possible. Many of the existing assets have two versions of an expression - one with glasses, and one without, for example - but the easier way to manage this is to simply have all of your expressions without the glasses, and then have a glasses attribute that can be optionally toggled on and off. The only character with an example of this is **layeredimage rika vn** where you can see her definition has been split into body, face, and head.

If you'd also like to define a "VN Mode image" for your new character (aka the VN Mode image that shows up beneath a chatroom which you select to enter VN mode), see 7.1 Setting up sequential chatrooms.

6.3 Allowing a new character to make phone calls

First, go to **character definitions.rpy**.

Under the header **Phone Call Characters** you'll see several characters already defined. Copy the last one, **vmail_phone**, and replace the name (**'Voicemail'**) with the name of your character. Rename the variable for your character - for example, if you want to create a phone call character for Bob, the character we defined in 6.1 Adding a new character to chatrooms, then you should call the variable **b_phone** e.g.

```
define b_phone = Character('Bob', what_font= "00 fonts/NanumGothic
(Sans Serif Font 1)/NanumGothic-Regular.ttf", what_color="#fff",
what_xalign=0.5, what_yalign=0.5, what_text_align=0.5,
voice_tag="other_voice")
```

You generally won't have to touch the other values, as they are the same for every character except for the MC. However, if you have voiced lines for your new character, you can replace **voice_tag="other voice"** with your own variable. See Note on voiced characters for more.

Now you can write phone calls using this new character, e.g.

```
b_phone "How are you, [name]?"
```

You also need to go to **phone screen.rpy** and scroll down to the **deliver_calls(lbl)** function. Here, you need to copy one of the **if** statements for **both** the incoming and outgoing portions and add your new character to them, e.g.

```
if renpy.has_label(lbl + '_phone_b'):

            available_calls.append(Phone_Call(b, lbl + '_phone_b',
'outgoing'))
```

This will allow you to write labels like **label mychat_phone_b:** which will let the player phone Bob after the chatroom **mychat**.

6.4 Allowing a new character to send text messages

First, go to **character definitions.rpy**.

Under the header **Text Messages** you'll see two lists, **text_messages** and **text_queue**. You must add your new Chat character defined back in 6.1 Adding a new character to chatrooms to the end of these lists. Copy the last entry, add a comma after it, then replace the Chat variable with the one for your character in both lists e.g.

```
default text_messages = [Text_Message(u, []),
                        Text_Message(sa, []),
                        Text_Message(b, [])
                        ]
default text_queue = [Text_Message(u, []),
                        Text_Message(sa, []),
                        Text_Message(b, [])
                        ]
```

****Other characters have been left out of the list for brevity; do not remove them from the list**

Now we can see that **b** (Bob) has been added to the end of both lists. This means that when you write text messages like

```
$ addtext (b, "Good morning!", b)
```

they will be delivered correctly like all the other text messages.

6.5 Adding or removing CG albums

First, go to **gallery.rpy**

You need to define both a persistent album and a regular album for your new character. So, if we're defining albums for Bob, it might look like this:

```
default persistent.b_album = []
```

and then the regular album:

```
default bob_album = []
```

Now we'll go to **script.rpy** and add another line under the **define_variables** label. You'll see several similar lines. This code will allow Bob's album to update when we add new images to it.

```
merge_albums(persistent.b_album, b_album)
```

Back to **gallery.rpy**, we can add Bob's album to the various album screens. In the screen **photo_album**, you'll see how the other characters' albums are defined. The screen can display a maximum of three columns, but you can add more rows to accommodate more characters. Not every row has to have three albums, so you can experiment with the placement.

Once you've figured out where you'd like to put Bob's album, we'll add him with the line

```
use char_album('cg_label_other', 'Bob', persistent.b_album,
'u_album_cover')
```

where **cg_label_other** is an image defined in **variables.rpy** under the **CGs** header (it displays the name of the character under their album image), **Bob** is the name of our character, **persistent.b_album** is the persistent album we defined earlier for Bob, and **u_album_cover** is also an image defined in **variables.rpy** under the **CGs** header (it's the image that will display on Bob's album button). You're encouraged to define your own images for the first and last variables and use those instead.

Bob's album should now show up when you click the Album button from the chat home screen. To add CGs to display in Bob's album, we need to create another folder in the **images/CGs** folder. Call the new folder **b_album**, the same name as we used for Bob's album variables. You can then add images to this album and post them as described in Posting CGs in chatrooms.

6.6 Giving a new character spaceship thoughts

To allow a new character to have "spaceship thoughts" (See 8.3 Spaceship thoughts for more information), you simply need to add another image to **images/Phone UI/Main Menu/Original Story/Spaceship**. In **variables.rpy** under the header **Spaceship thought images**, you can find the existing images defined. If you've named your character 'Bob' and his file_id is 'b', you need to define a new image here like so:

```
image b_spacethought = "Phone UI/Main Menu/Original
Story/Spaceship/b_spacethought.png"
```

This image should be 651x374 pixels, and will be shown behind the thought the character has when the spaceship icon is clicked.

You can then give your character a **SpaceThought** in the list like everyone else e.g.

```
SpaceThought(b, "Your thought here.")
```

7. Setting up a route

7.1 Setting up sequential chatrooms

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

1. Fill out the **List of Chatrooms** tab in the **Script Generator.xlsx** spreadsheet with your chatrooms
2. Find the **chat_archive** variable in **route_setup.rpy** and replace its **Chat_History** objects with the ones you made in the spreadsheet (Optional: keep a copy of the original **chat_archive** in case you want to go through those chatrooms again later, or keep a save file at the beginning of Tutorial Day)
    a. It's a list, so be sure to add a comma after each item
    b. You can rename the days whatever you like and add more or fewer days as needed
    c. You can also add a field after the list of **Chat_History** objects to make a character's image appear on the Timeline screen
3. Select **Start Over** from the settings screen to test out your new route

Archive

Archive(day, archive_list=[], route='day_common2')

First, go to **route_setup.rpy** and find the variable **chat_archive** just below the python definitions. This is where you'll declare your chatrooms to set up a route.

First, you'll notice that **chat_archive** uses a list of **Archive** objects. An **Archive** object has three fields: first, the **day**, which is generally equal to '1st', '2nd', etc. You can, however, name these whatever you like - for example, there is a 'Tutorial' day included with the program. You can also create as many days as you like - more than 11 days is fine. The last day should be titled **'Final'** if you'd like the special icon above that day.

The second field in an **Archive** object is a list, called **archive_list**. This list contains **Chat_History** objects. We'll get to that in a second.

The third and final field of an **Archive** object is called **route**, which refers to the character whose route you're on. This is so the program can display the right image on the day select screen. If you look in **variables.rpy**, under the header **Chat Select Screen** you'll find several images already defined. These are:

- **day_common1** (the golden default 'route')
- **day_common2** (an alternate blue version of the default route)
- **day_ja** (Jaehee's route)
- **day_ju** (Jumin's route)
- **day_r** (Ray's route)
- **day_s** (Seven's route)
- **day_v** (V's route)
- **day_y** (Yoosung's route)
- **day_z** (Zen's route)

You won't need to write anything more if you want the route to be the 'common' route. Otherwise, this third field should be **'day_ja'** or whoever's route you'd like the day to show up as.

Now, we'll look at **Chat_History** objects. You need a list of these as the second argument in the **Archive** object. Each **Chat_History** object represents a chatroom in your route. The easiest way to set them up is by using the **List of Chatrooms** tab on the **Script Generator.xlsx** spreadsheet.

The column headers are fairly self-explanatory. **Save Image** helps the program decide which icon it should use for a save file. Generally it will be the name of the route character, in lowercase. You can find the possible images under the **Save & Load Images** header in **variables.rpy**. You don't need **save_** in front of the name, so if you want the **save_deep** image (indicating deep route), then you should write **deep** under the **Save Image** column in the spreadsheet. **auto** is just the generic save image. Note that this image is independent of the **route** in the **Archive** object - that one uses the variable to set up the day list. This **Save Image** simply sets the image for a save file created during that chatroom.

Chat_History

Chat_History(title, save_img, chatroom_label, trigger_time, participants=[], vn_obj=False, plot_branch=False)

Trigger Time = when the chatroom should show up. The program has two different modes - real-time and sequential. You can switch between the two modes on the Settings screen. If you're playing in sequential mode (the default), trigger time just indicates the "time" it is when the chatroom begins, but it will be triggered sequentially when the previous chatroom is finished regardless. **It's 24-hour and requires a leading zero for hours less than 10.** So, 12:44am should be written as 00:44, 9:10am should be written as 09:10, and 8:33pm should be written as 20:33. In real-time mode this will be the actual time when the chatroom is triggered, according to your system clock. There's more explanation on real-time mode and expired chatrooms in 1.5 Expired chatrooms.

Participant List refers to the people who are already present in the chatroom before you enter it. If no one's in it, you can leave this blank. If more than one character is in the chatroom, separate them with commas e.g. **y, z**

VN_Mode

VN_Mode(vn_label, who=None, party=False)

Finally, if there is a VN that comes after this chatroom, you need to fill in the 'optional' columns. **VN Character** can be left blank if you'd like the "generic" VN mode icon. Otherwise, if you want a character's specific VN icon to be used, fill in this column with their Chat variable (so, **ja/ju/r/ri/s/sa/u/v/y/z**). If you defined your own character, you can give them a VN mode image as well. The existing images can be found in **variables.rpy** under the header **Chat Select Screen**. They are formatted as **vn_ja, vn_ju** etc. So, if we wanted Bob from 6.1 Adding a new character to chatrooms to have his own VN image, we'd call it **vn_b** and define the image as shown in the other examples. Then you can fill in the **VN Character** column in the spreadsheet with **b**.

Note: there are also variables **ja_vn**, which is NOT the same as **vn_ja**. **vn_ja** is an image used in the Day Select Timeline screen. **ja_vn** is a **Character** object which allows a character to speak during VN mode sections.

The **party** variable is used if you fill in the "VN Character" column with "party". It uses the party VN icon instead of the regular icons associated with the various characters. See 7.4 Creating the party for more information on this.

When you're done filling out all the columns you need, simply copy the cell under **What to fill in to the program** into your list of **Chat_History** objects.

Essentially, an **Archive** object will look like this:

```
Archive('1st', [Chat_History(...), Chat_History(...)], 'day_ja')
```

where **Chat_History(...)** is an appropriately filled-out **Chat_History** object as described above. You can leave the last argument (in the above example, **'day_ja'**) out if that particular day is not on any character's route.

Your full **chat_archive** variable will then look something like this:

```
default chat_archive = [Archive('1st', [Chat_History(...),
                              Chat_History(...),
                              Chat_History(...)],
                                'day_ja'),
                        Archive('2nd', [Chat_History(...),
                              Chat_History(...),
                              Chat_History(...)],
                                'day_ja'),
                  Archive('Final', [Chat_History(...)],
                                 'day_ja')]
```

Again, **Chat_History(...)** should be an appropriately filled-out **Chat_History** object as described above. I've also added some colour-coding to the brackets in the above example so you can differentiate which opening bracket pairs with which closing bracket. You can add as many or as few **Chat_History** objects in your **Archive**'s **archive_list** as you like and, as always, the **'day_ja'** part is optional.

7.2 Chatrooms with content afterwards

Example files to look at: **Coffee Chat.rpy**

If there's anything you'd like to do after a chatroom, or any additional content you'd like to make available to the player (e.g. text messages), you need to create an "after chatroom" label. This label will be **after_** + the name of your chatroom label.

For example, if your chatroom label is called

```
label my_chatroom:
```

then the "after label" should be called

```
label after_my_chatroom:
```

You can put many things here. If you'd like the player to receive text messages after the chatroom, see 4. Text Messages for more information. You can also change a character's voicemail here: see 3.3 How to change a character's voicemail.

7.3 Plot Branches

Example files to look at: **route_setup.rpy, plot branch.rpy**

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

1. Use the **List of Chatrooms** tab in **Script Generator.xlsx**. Fill in your chatroom as normal, and be sure to check off the **Plot branch after this?** column
    a. If you want a VN mode to only be available **after** the player has gone through the plot branch, make two versions of your chatroom. One will **not** have the VN mode columns filled out, and will have the plot branch column checked. The other will **not** have the plot branch checked, and will have the VN mode information filled out
2. Create another list of **Archives** (like the **chat_archive** variable in **route_setup.rpy**). If your plot branch is on the '5th' day, start your list of Archives on the '5th' day
3. After the chatroom label with your plot branch, create a label with the same name + the suffix **_branch** e.g. label mychat_branch
4. Put whatever criteria you're testing for the plot branch in this label. To set the pllayer on a route, use the function **$ merge_routes(your_route_variable)**
    a. If you have a VN mode that's only available after going through the plot branch, write **True** as the second field e.g. **$ merge_routes(your_route_variable, True)**

If you get to a point where you'd like a "plot branch" to occur (aka the plot will continue differently depending on the player's choices up to that point), you first need to set up another list of Archive objects. See 7.1 Setting up Sequential Chatroom first for information on how to do so. The only difference is that you only need to create an **Archive** object for a day you have new content for. In other words, if you want to check on the 4th day if a player will move on to a route, and then begin the new content on the 5th day, you only need to create **Archive** objects from the 5th day onwards. If you have a plot branch occur midday on the 4th day and want to continue to have different chatrooms into the evening depending on the player's choices, then you should create **Archive** objects starting from the 4th day. You can look at the variables **tutorial_bad_end** and **tutorial_good_end** in **route_setup.rpy** for some examples of this. The variable **seven_route** also shows roughly how a route might look so that all of the icons for days 5-11 show Seven's image. You would fill in your chatrooms in the second field, currently **[]**.

After the chatroom where you'd like to have a plot branch, after the **vn_obj** field we'll have a field equal to **True** to indicate to the program that there's a plot branch after this chatroom. If you're using the **Script Generator.xlsx** spreadsheet, you'll notice on the **List of Chatrooms** tab that there's a column called **Plot branch after this?**. Checking this off will fill out the **Chat_History** object correctly for you.

Note: Plot branches and VNs

Case 1: If you'd like a VN mode section to only show up **after** the player has gone past the plot branch, then in your new **Archive** list, the first chatroom will need to be a 'duplicate' of the chatroom before the branch, plus the **vn_obj** you want. The **Chat_History** object in your current **chat_archive** variable should **not** have the **vn_obj**, and instead that field should be marked as **False**. You can see an example of this in **tutorial_good_end** found in **route_setup.rpy**. Note that the final **Chat_History** object on 'Tutorial' day is

```
Chat_History('Plot Branches', 'auto', 'plot_branch_tutorial', '10:44',
[], False, True)
```

where the bolded **False** is the field **vn_obj** and the field marked **True** after it is the **plot_branch** field.
Technically the actual names and fields of the new route's **Chat_History** object won't matter aside from the **vn_obj** field, but it's a good idea to keep the naming consistent so it's easier to understand later.

Case 2: If, instead, you'd like the VN mode section to be playable **before** the player goes through the plot branch, then you can declare the chatroom as usual, being sure to check the **Plot branch after this?** column if you're using the **Script Generator.xlsx** spreadsheet. There's no need to create any duplicate **Chat_History** objects in your new route.

Case 3: If there are **no VN objects** before or after your plot branch, then be sure that the **vn_obj** field in your **Chat_History** object is marked as **False**. See the code above for an example of this. In this case, you **will not** need to create a duplicate **Chat_History** object in your new route.

The final thing you need to do for your plot branch is to create a label which tells the program which route to put the player on. This label follows a specific naming scheme, with the suffix **_branch** following the title of the chatroom before your plot branch. For example, if the chatroom before your plot branch is titled

```
label mychat:
```

then you need to create a label titled

```
label mychat_branch:
```

You can see in **plot branch.rpy** under the label **plot_branch_tutorial_branch** there are several examples of functions you might want to use to calculate which route the player gets on. For example, checking if they have a minimum number of hearts with a particular character, or how many guests they've invited. To set the player on a route, use the function

```
$ merge_routes(tutorial_good_end, True)
```

where **tutorial_good_end** is the name of the route variable you defined (should be a list of **Archive** objects), and **True** is an optional variable that lets the program know whether or not there should be a VN mode immediately after this chatroom. See the above note on Plot branches and VNs - you will only need to include this field for **Case 1**. Otherwise, **$ merge_routes(tutorial_good_end)** is sufficient for cases 2 and 3.

7.4 Creating the party

Example files to look at: **plot branch.rpy**

The "Party" at the end of a route functions almost the exact same as any other VN section. The only real difference is in the icon itself.

You can tell the program to display the party VN button when you're setting up your route. The **VN_Mode** object has a special third field, **party**, which we set to **True** to get the correct icon. If you're using the **Script Generator.xlsx** spreadsheet, you can fill out all of your columns as normal, but under the **VN Character** column, write **party**. The **Chat_History** object will look like the following:

```
Chat_History("Onwards!", 'auto', 'tutorial_good_end', '13:26', [u],
VN_Mode('good_end_party', None, True))
```

When the player clicks on the Party icon, they will be taken to the label you defined in the **VN_Mode** object (so, in this case, **'good_end_party'**).

Note that if you want to do any final "checks" before starting the party, you should put them at the beginning of this label. For example, just before the party, you can check how many guests the player has successfully invited:

```
label my_party:
    if attending_guests() >= 10:
        jump my_party_good_end
    else:
        jump my_party_bad_end

label my_party_good_end:
    # (Your party VN here)

label my_party_bad_end:
    # (Your party VN here)
```

This is very similar to plot branches; you can see 7.3 Plot Branches for more information on the functions you might use to check the player's progress. However, since the party is the final part of the game, you do not need to merge routes.

At the end of the party, if you'd like to end the game, you can write

```
jump restart_game
```

This works the same as hitting "Start Over" from the settings screen, and will reset the game to its initial state so the player can start a new game. They can still load their save files to go back and make different choices.

You may also want to show them an image saying "Good End" or "Bad End" etc before resetting the game. If you are ending the game directly from a chatroom, you can write

```
call chat_end_route('bad')
```

where 'bad' is either 'good', 'normal', or 'bad', and shows the image corresponding to that ending. Then you should write **jump restart_game** as mentioned above.

Otherwise, if you're ending the route after a VN (including a party VN), you can write

```
scene bg good_end
pause
jump restart_game
```

where **good_end** is one of **good_end**, **bad_end**, or **normal_end**, depending on the ending screen you'd like to show. The pause statement simply gives the player time to look at the image before the game restarts.

7.5 Customizing the route select screen

Example files to look at: **menu screen.rpy**

When you begin a new game, after pressing "Original Story" you'll be taken to the "Mode Select" screen where there is a large button that says "Start Game". This screen is defined in **menu screen.rpy** called **screen route_select_screen**.

You can customize this screen however you like beyond the setup lines, which should be as follows:

```
screen route_select_screen:
    tag menu
    use starry_night
    use menu_header("Mode Select", Show('main_menu', Dissolve(0.5)))
```

The only thing you will definitely need is a button that will take the character to the beginning of the game or route. If you're using a common route, it's sufficient to have the button action be

```
action Start()
```

This will take you to the **start** label found in **script.rpy**.

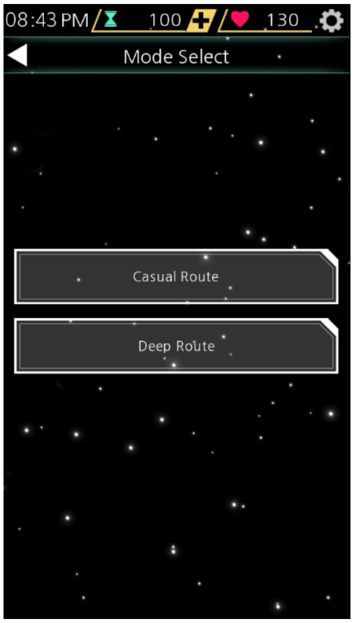
Otherwise, if you'd like to have multiple routes, you need to instead tell each button to jump to a specific label. For example, if we wanted the player to be able to choose between a "Casual" and "Deep" route, the route_select_screen might look as follows:

```
screen route_select_screen:
    tag menu
    use starry_night
    use menu_header("Mode Select", Show('main_menu', Dissolve(0.5)))

    vbox:
        maximum(700, 350)
        align (0.5, 0.5)
        spacing 30
        button:
            xysize (700, 120)
            focus_mask True
            background "right_corner_menu"
            hover_background "right_corner_menu_selected"
            action Jump('casual_route_start')
            text "Casual Route" style "menu_text_small" align(0.5,0.5)
        button:
            xysize (700, 120)
            focus_mask True
            background "right_corner_menu"
            hover_background "right_corner_menu_selected"
            action Jump('deep_route_start')
            text "Deep Route" style "menu_text_small" align(0.5,0.5)
```

You can of course change the button backgrounds, text, spacing, etc. This will create a very simple route select screen that looks like the image on the right. For more information on customizing screens, you can read the Ren'Py documentation on Screen Language.

Since you have two separate routes, you need to define them as lists of Archive objects separate from the **chat_archive** variable. You can read more on this in 7. Setting up a route. We'll assume that you've called these variables **casual_route** and **deep_route**.



Next, the labels that you jump to (in this case **casual_route_start** and **deep_route_start**) will need to have a few lines of code at the beginning to start the route off properly.

If you would like to have an introductory chatroom, you can follow the example in **script.rpy** under the **start** label. There's more information on that in 1.4 Creating an "Opening" chatroom. The most important lines to include are:

```
$ chat_archive = casual_route

call define_variables

$ current_chatroom = Chat_History('Starter Chat', 'auto',
'starter_chat', '00:00')

$ starter_story = True
```

The first line will set up casual_route as the route to go through. You should replace **'starter chat'** in the third line with your own label name if you'd like to include an after_ label with content such as text messages. So, for example, the third line might look as follows:

```
$ current_chatroom = Chat_History('Casual Route Starter Chat', 'auto',
'casual_route_starter', '00:00')
```

This will allow you to make a label called **after_casual_route_starter** which the program will jump to after the chatroom is completed, and where you can put text messages, change voicemails or give the characters status updates, and more.

You also need a **jump chat_end** at the end of your introductory chatroom.

If you do **not** want an introductory chatroom, and instead would like the player to begin in the chat hub where they can begin playing after selecting a chatroom from the day list, you still need to include some code after the label you told the program to jump to. For example, if we want our casual story route to jump right to the chat hub when selected, **casual_route_start** should look like the following:

```
label casual_route_start:
    $ chat_archive = casual_route
    call define_variables
    $ persistent.first_boot = False
    $ persistent.on_route = True
    $ next_chatroom()
    call screen chat_home
```

This should make it possible for the player to choose between multiple different routes when beginning a new game.

7.6 Unlocking routes when certain conditions are met

If you'd like to keep certain features "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've gone through Character A's route first), you need to set up your own persistent variable to keep track of whether the player has fulfilled that condition or not. You can define these variables just about anywhere you like, but I recommend you do so in **variables.rpy** under the **Short forms/Startup Variables** heading. To define this variable, we need to use Ren'Py's special **persistent** object so that the program will remember whether or not the player has fulfilled these conditions even if they start a new game.

For example's sake, we're going to have a variable check whether or the player has successfully gotten the Good End in Tutorial Day. First, we define our variable:

```
default persistent.tutorial_good_end_complete = False
```

tutorial_good_end_complete is the name of our field in the persistent object; you need to preface it with **persistent.** to get it to save properly across different playthroughs. We initialize the variable to **False**, and we will set it to **True** once the player has successfully gone through the Good End.

A player who has gone through the Tutorial Good End will have finished the party VN called **good_end_party** found at the bottom of **plot branch.rpy**. The last lines of the label are as follows:

```
    scene bg good_end
    pause
    jump restart_game
```

jump restart_game will reset the game and take the player back to the main menu, so just before that line we'll include a line to toggle our variable.

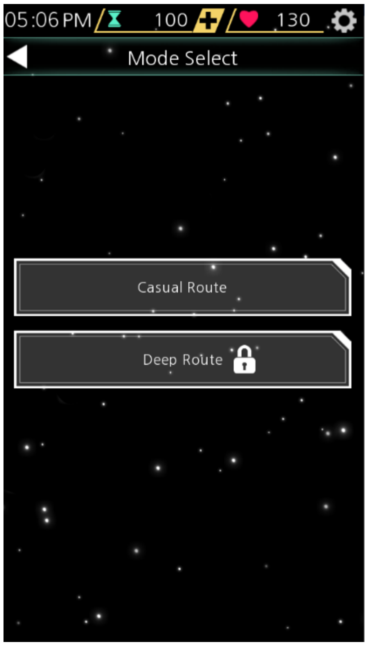
```
    scene bg good_end
    pause
    $ persistent.tutorial_good_end_complete = True
    jump restart_game
```

If your route ends with a chatroom rather than a VN, putting this line just before **jump restart_game** will be sufficient to let the program know the player has completed the route. Of course, you're not limited to putting your condition check right at the end of a route - you could also put the **$ persistent.your_variable_here = True** line after a menu option, or after a particular phone call, for example. Note that if you don't need the program to remember a variable across playthroughs (for example, if you want the program to remember that the player told Bob "I own a cat" so that you can later have him mention the cat while you're on his route), you **do not** need **persistent.** in front of your variable declaration and can simply write **default owns_cat = False** and set **$ owns_cat = True** after the line where the player declares they own a cat. This variable will be reset to its default value of False whenever the player begins a new game, unlike persistent variables which aren't reset when the player begins a new game.

Now, if we want to only allow the player to access a route if a certain condition is fulfilled, we'll need to add some extra code to the route select screen. For this example, we're going to assume you have a **route_select_screen** set up as described in 7.5 Customizing the route select screen, with a Casual and a Deep route.

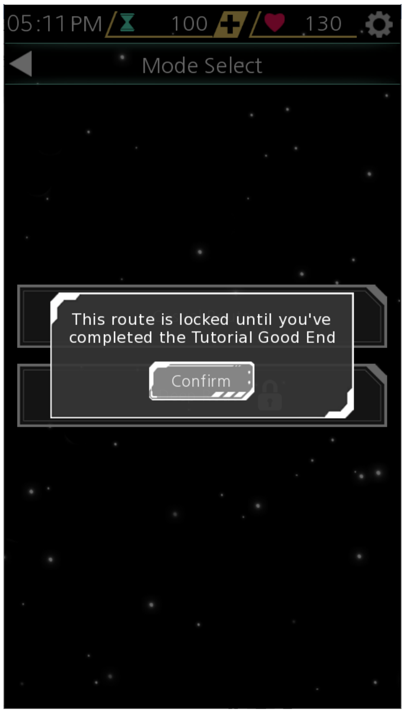
In this example, we'll make it so that Deep Route shows as "locked" until after the player has completed the Good End of Tutorial Day. Everything from the **route_select_screen** will remain the same except for the button for Deep Route:

```
button:
    xysize (700, 120)
    focus_mask True
    background "right_corner_menu"
    if persistent.tutorial_good_end_complete:
        add 'plot_lock' align (0.7, 0.5)
        action Show("confirm", message="This route is locked until
you've completed the Tutorial Good End",
                    yes_action=Hide('confirm'))
    else:
        hover_background "right_corner_menu_selected"
        action Jump('deep_route_start')
    text "Deep Route" style "menu_text_small" align(0.5,0.5)
```



This creates a screen that looks like the image on the left. If **persistent.tutorial_good_end_complete** is **False**, they will see the lock symbol on the Deep Route button, the button won't light up when hovered over with the mouse, and if they click on Deep Route it will give them a message that says they can't proceed until they've completed the Tutorial Good End, as seen in the image below.

After the player has completed the Tutorial Good End, however, the lock symbol will disappear, the button will light up when hovered over, and clicking on Deep Route will take the player to the start of that route. You could also keep the Deep Route button the same as it is in the original **route_select_screen** and instead put the statement **if persistent.tutorial_good_end_complete:** above the Deep Route button (remembering to indent all of the code one step to the right after the **if** statement). If you do this, then the Deep Route button will not even be shown to a player who hasn't unlocked it yet. Keep in mind, however, that it's usually a better idea to show the player this content exists so they have a reason to come back and play more of the game.



This route is locked until you've completed the Tutorial Good End

8. Miscellaneous

8.1 Pronoun integration

Example files to look at: **popcorn.rpy**

This program allows the player to change their pronouns whenever they desire during the game. This means that any references to the player's gender or use of pronouns to refer to the player will need to be taken care of via variables. In **variables.rpy** under the **Short forms/Startup Variables** heading you will find many default variables already defined.

If you'd like to define more variables, you need to first define the variable here, and also under the **set_pronouns()** function in **menu screen.rpy**.

For example, if we'd like to add a variable that is "has" if the player is using he or she pronouns, but "have" if the player is using they pronouns, it would look like the following.

In **variables.rpy** under the **Short forms/Startup Variables** heading, we add

```
default has_have = "have"
```

Then in **menu screen.rpy**, under the function **set_pronouns()** we add

```
has_have = "has"
```

under both **if persistent.pronoun == "female":** and **elif persistent.pronoun == "male":**, and add

```
has_have = "have"
```

under **elif persistent.pronoun == "non binary":**.

This will set the variable appropriately regardless of what pronouns the player chooses. You also need to add the **has_have** variable to the **global** variables defined at the top. You can write

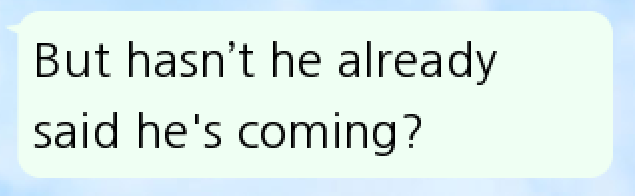
```
global has_have
```

or simply add it on to the end of one of the existing lists after a comma.

To use our new variable, we can write dialogue such as

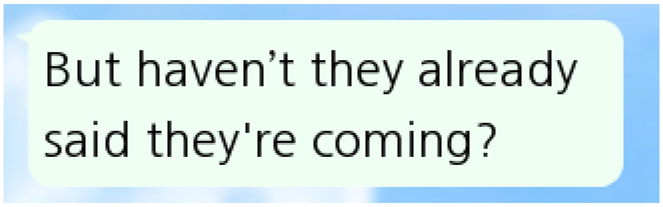
```
y "But [has_have]n't [they] already said [they_re] coming?"
```

If the player has he/him pronouns, in-game this will display as



But hasn’t he already said he's coming?

However, if the player has they/them pronouns, this displays as



But haven’t they already said they're coming?

Variables are capitalization-sensitive, so you can also create "capitalization" versions of variables (e.g. you could have a separate **Has_Have** variable that is either **Has** or **Have** depending on preferred pronouns). There's no limit to how many of these variables you can make, so feel free to create as many as you need to write your script more easily.

8.2 Custom Emojis

Example files to look at: emojis.rpy

If you'd like to add your own emojis to the game, you need to add a few lines to the **emoji_lookup** dictionary found in **emojis.rpy**. Emojis are saved as two separate .png files, and are found in the **/Gifs** folder under **/images**. At the bottom of **emojis.rpy**, you can see how to define an emoji.

```
image jaehee angry:
    "Gifs/Jaehee/emo_jaehee_angry1.png"
    0.5
    "Gifs/Jaehee/emo_jaehee_angry2.png"
    0.5
    repeat
```

Most emojis have the name of the character (jaehee), a name for the expression (angry), and then ATL is used to animate the image. The first line should be the path to the first frame of your emoji, followed by **0.5** to tell the program to wait half a second before showing the next image. The next line is the path to the second frame of your emoji, followed by another **0.5**, and then the line **repeat** which tells the program to continue to animate the emoji while it's on the screen. You can, however, add more frames to your .gif animation and reduce the time for the pause accordingly.

Next, go to the **emoji_lookup** dictionary. You need to insert your own entry in order for the emoji to play sound when it's used in a chatroom.

```
'{image=jaehee angry}': 'sfx/Emotes/Jaehee/jaehee_angry.wav'
```

You can see that on the left side we used the same name as we defined for the image earlier, **jaehee angry**. You should replace **jaehee angry** with whatever you named your emoji previously.

The right side is the path to the sound effect you want to play when this emoji is shown. Currently all the sound effects are organized in the **sfx/Emotes** folder under their respective characters.

Then, to show your emoji in the chatroom, you just need to type

```
ja "{image=jaehee angry}" (img=True)
```

and it will display the emoji accompanied by the right voice clip. Note that if you use the **Script Generator.xlsx** spreadsheet to write chatroom dialogue, it will usually check off the "image" column automatically for you if it detects **{image=** in the Dialogue column. However, if the emoji is displaying oddly in-game, you may want to ensure this is checked off and that **(img=True)** accompanies the line in your script.

You may also notice that there are a few variables defined under the emoji dictionary; these are used in the **chatroom generator**, which is not yet fully implemented. Therefore you can just ignore these variables for now.

8.3 Spaceship thoughts

SpaceThought

SpaceThought(char, thought)

Example files to look at: **popcorn.rpy**

When the floating spaceship isn't giving out chips, you can click it to view a random thought from one of the characters. You'll find the functions and screens used to handle this in **spaceship.rpy**. The variable **space_thoughts** can be modified to change the spaceship thoughts the player sees upon starting up the game. You can have as many or as few **SpaceThought** objects in the list as you like - even multiple thoughts for one character. A **SpaceThought** only has two fields - the first is the character variable of the character whose thought it is, and the second is the thought itself.

To change the spaceship thoughts, you need to add a line in the **after_** label of the chatroom you want the spaceship thoughts to change after viewing. It will look like this:

```
$ space_thoughts.new_choices( [

            SpaceThought(ja, "What I wouldn't give to go home and
watch one of Zen's DVDs...")

        ] )
```

You can add as many or as few **SpaceThought** objects as you wish in the list, so long as you make sure to separate them with commas. This will clear the previous list of thoughts and replace it with your new list after the player has gone through the chatroom (and the VN mode as well, if there is one).

8.4 Adding new ringtones/email tones/text tones

First, go to **settings screens.rpy**. There are three dictionaries defined at the beginning of the file, along with three lists, one of each for email, text, and ringtones. As the method is the same for all three, only adding a text tone is described below.

First, add a new entry to the **text_tone_dict**. Dictionary entries are a key-value pair, separated by a colon. In this case, the "key" is the name of the tone we want to show the player, and the "value" is the path to the file with the correct sound effect. Note that no two entries can have the same key, so you should preface entries with a category title if you're adding multiple text tones for the same character e.g.

```
default text_tone_dict = {
      'ZEN': 'sfx/Ringtones etc/text_basic_z.wav',
     'Bonus ZEN': 'sfx/Ringtones etc/text_bonus_z.wav' }
```

Next, you need to add your ringtone to the **text_tone_list**. You can either add on to an existing list entry to add more items in the same category, or we can create a new category like so:

```
default text_tone_list = [ ["Basic", ['Default', 'Jumin Han', 'Jaehee
Kang', '707', 'Yoosung★', 'ZEN' ]],
                    ["Bonus", ['Bonus ZEN']]
                          ]
```

This will create a new category called "Bonus" which is shown when the player selects a new text tone. Note that the entries need to match up with the key you entered in the **text_tone_dict** dictionary. And that's all! Your ringtones should show up and be selectable in the Settings menu.

8.5 "Hacked" mode

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 when selecting a chatroom 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
```

will activate it. Similarly, **$ hacked_effect = False** will get rid of these effects.

While you're testing a route, there's also an option to toggle the hacked_effect on/off in the Others tab of the Settings menu.

8.5.1 The tear screen

Additionally, 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 on the chatroom timeline screen, but can also be shown in the middle of any ordinary chatroom/phone call/VN mode/etc. You call it like so:

```
show screen tear(10, 0.1, 0.1, 0, 40, 0.3, None)
```

where **10** is the number of pieces to tear, **0.1** is the value for both **offtimeMult** and **ontimeMult** respectively - this controls how much the pieces bounce back and forth - **0** is the minimum offset for the pieces (this can be a negative number) and **40** is the maximum offset for the pieces (offset just means how far the pieces will move away from their origin point, in pixels). Finally, there are two more optional arguments at the end. **0.3**, in this case, is the value for the timer (in seconds). If you provide a number, such as 1, then the screen tear will automatically disappear after 1 second. If you don't provide a number or this field is False, then you will have to hide the tear screen yourself. Finally, **None** is another optional argument, **srf**, that is either equal to None or is a special kind of render object written like so:

```
show screen tear(number=50, offtimeMult=0.4, ontimeMult=0.2,
offsetMin=-100, offsetMax=100, w_timer=0.2,
srf=renpy.render(Image('Phone UI/Day Select/vn_party.png'),
750,1334,0,0))
```

The field names have been included for clarity. Here we can see that **srf** is equal to **renpy.render(Image('Phone UI/Day Select/vn_party.png'), 750,1334,0,0))**. To use your own image, simply replace **'Phone UI/Day Select/vn_party.png'** with the path to your own file. If you provide an image file, the tear screen will tear that image into pieces to display over the screen. If you don't provide one, it will instead take a screenshot of the current screen and use that image. You may, however, find it easier to display an image on the screen yourself, and then call the tear screen without a **srf** argument, like so:

```
show screen display_img([ ['vn_party', 200, 400] ])
pause 0.0001
show screen tear(40, 0.4, 0.2, -100, 100, 0.3)
pause 0.3
hide screen display_img
```

Here we use the special screen **display_img** to put our images on the screen. The **display_img** screen takes a list 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 an image, among other things), then the xpos, and the ypos. So in this case, the program displays the image **'vn_party'** which is defined in **variables.rpy** 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. We call the tear screen as normal without a **srf** argument. Because we are showing the tear screen for 0.3 seconds, we also pause the game for 0.3 seconds to give the "glitch" time to be viewed without advancing the program. Finally, we hide our **display_img** screen. Because we gave the tear screen a **w_timer** argument (0.3) we don't have to manually hide that screen.

8.5.2 The hack_rectangle screen

The **hack_rectangle** screen will also help create a "hacked" effect. It takes one field, **w_timer**, which indicates how long (in seconds) the screen will be shown to the player. If no value is given or w_timer is False, it will not hide the screen automatically and you will need to hide it yourself.

This screen will show several random rectangles on the screen. It uses the previously defined images **m_rectstatic**, **m_rectstatic2**, and **m_rectstatic3**, though you can modify it to suit your own needs as well. The screen is found in **glitch effects.rpy**. This screen works well when paired with the tear screen. For example:

```
show screen hack_rectangle(0.2)

pause 0.01

show screen tear(number=10, offtimeMult=0.4, ontimeMult=0.2,
offsetMin=-10, offsetMax=30, w_timer=0.19)

pause 0.19
```

Again, pausing for 0.01 seconds after showing the hack_rectangle screen gives the program enough time to display the rectangles so they can be used in the screenshot by the tear screen.

8.5.3 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 main screen. It is used on the chatroom select screen when **hacked_effect** is True. To call it, simply show the screen:

```
show screen white_squares(0.16)
```

where **0.16** is the amount of time to show the screen before hiding it. If no number is given, the screen will not hide itself and will continue to show the white static until manually hidden using **hide screen white_squares**.

8.5.4 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. It has an optional argument, **length**, which indicates how long the colour-inverted screenshot should be shown before it is hidden. If no arguments are given or length is False, you will need to manually hide the screen yourself. This screen works well with the other screens that were previously mentioned. For example:

```
show screen hack_rectangle(0.2)
pause 0.01
show screen invert(0.19)
pause 0.01
show screen tear(number=10, offtimeMult=0.4, ontimeMult=0.2,
offsetMin=-10, offsetMax=30, w_timer=0.18)
pause 0.01
show screen white_squares(0.16)
pause 0.17
```

As with the previous examples where we "combined" showing multiple screens, you must be sure to include a short pause after showing a screen which you would like to show up in a screenshot (so, screens that are shown before showing the **invert** or **tear** screens). You should also include a pause after showing all of your screens to prevent the game from continuing while the effects are shown.