Translating Ren'Py games: A guide, by moskys

moskys#0576

Hello!

Call me moskys. As you surely know if you have come to this guide, Ren'Py is a free and open source game engine (used for creating visual novel games, generally) that has become one of the most popular software for amateur creators to develop their projects. Based on Python programming language, its simplicity and flexibility is a great advantage, since it allows to achieve more than acceptable results without prior or just very minimal programming knowledge.

And among the many functionalities included in Ren'Py there's the option to facilitate the translation of the games created with this engine, although it does not perform the translation itself. So after more than three years playing and manually translating into Spanish (and for PC) several games of different complexity, in December 2020 I decided to write the first edition of this guide to explain how the translation process works to anyone interested in translating (or anyone curious about what I do in my spare time).

Now, a year later, and with more experience, I'm releasing this expanded (and improved, I hope) guide. And while it's intended for unofficial translators, the method described is obviously also applicable to those developers who want to launch their game in more than one language.

Note that I'm only explaining how to get the translation scripts ready to be translated and then how to fix some issues you may encounter; if you're looking for some tool to effectively translate those scripts you'll have to look elsewhere. This guide will allow you to understand how this process works and how you can improve your translations regardless if they are manual or machine ones, but you won't learn how to actually translate.

(Please forgive my English; I swear I'm way better translating from English into Spanish)

ESSENTIAL (AND FREE!) TOOLS – versions as of 2021-Dec-31

- **Ren'Py SDK 7.4.11 ->** https://www.renpy.org/latest.html
- **UnRen v.0.11 ->** https://f95zone.to/threads/mod-update-of-the-tool-unren-v-0-11v2-old-and-new-unren-windowed.92717/ *(link to NSFW forum)*
- **Text editors (suggested software, choose your preferred one):**
  - **Notepad++ 8.1.9.3 ->** https://notepad-plus-plus.org/downloads/
  - **Atom 1.58.0 ->** https://atom.io/ (it's hihgly recommended to download the Ren'Py's optimized version you can get from the Ren'Py SDK itself)

TABLE OF CONTENTS

0-. THE ART OF TRANSLATING IN 10 STEPS

Fiiiine, I know. This guide is too long, it contains an awful lot of written info and you're likely in a hurry to start translating and probably think all of this is just a bit too much. So, in absence of a fancy videotutorial, here's a brief schematic with the main steps to take in order to translate a Ren'Py game:

1. Check the "game" folder for scripts with the extension .rpy

2. If there's none, use unRen to extract them from the .rpa file and/or decompiling them from their .rpyc files

3. Are you translating a game's update? Paste now the old version's translation

4. Check the scripts for strings that can't be extracted by Ren'Py SDK

5. Launch Ren'Py SDK and generate the translation scripts:
    1) Select the game you want to translate (if it's not listed under the "Projects" section, look for its container folder by going to preferences > Project's folder)
    2) Click on Generate Translation
    3) In "Language", write the name of the language you want to translate the game into (and don't forget what you wrote there)
    4) Check (or uncheck, as you prefer) the "Generate empty strings" box
    5) Click on Generate Translations

6. Add the switch languages option by editing the game's Preferences screen (or creating a new one)

7. Translate every .rpy script within the "game/tl/*name-of-your-language-in-5.3)"* subfolder

8. Play the translated game and check if there's any errors:
    1) The game can't be launched or you're getting exceptions? Check errors.txt and traceback.txt
    2) What if some text don't fit in the textbox?
    3) Are there some font characters missing? Change the style
    4) What could we do with those pictures that contain some text on them?
    5) Some choices just don't seem right, although they are translated? We may have homonyms
    6) Do you notice some grammar concord issues?
    7) Some words are still being displayed in the original language? Repeat step 4, although they may be interpolated variables o concatenated strings
    8) Are you still unable to get everything translated into your language? Try a last resource

9. Do you want to share your translation? Build a patch

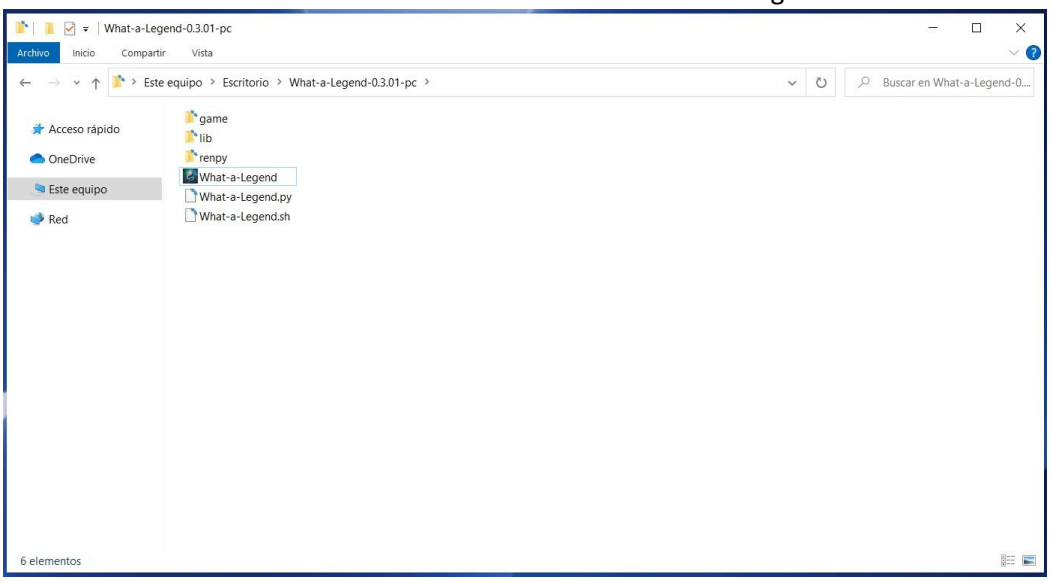
10. Did you create a patch? Be nice and let the game's dev know about it!

|Index|

1.- INTRODUCTION

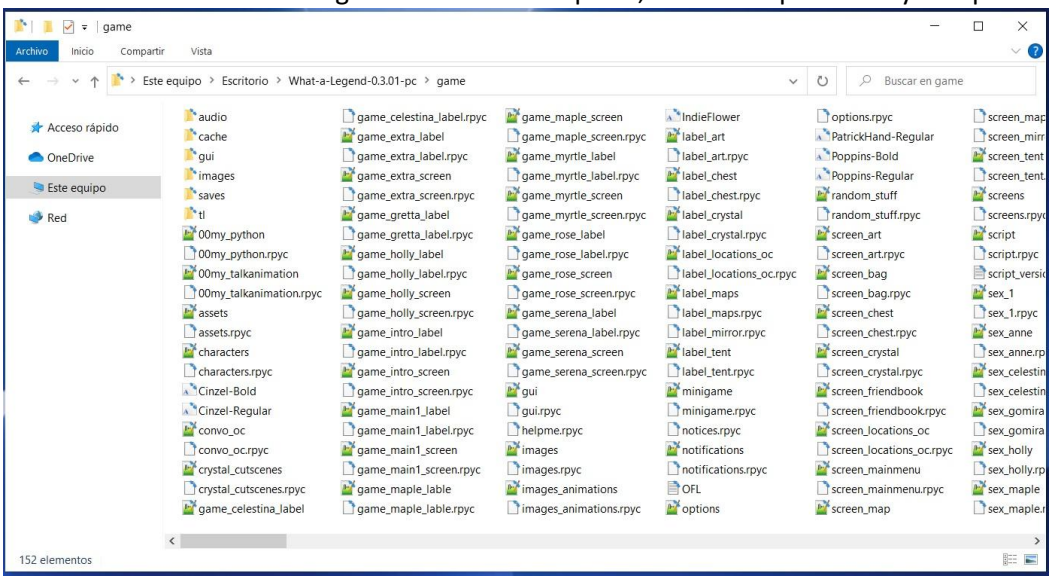
1.1.- Under the hood of a Ren'Py game

It is obvious, but in order to translate a game, first you must have a game. So look for any Ren'Py game that you have downloaded and take a look at its root folder. You'll see something like this:



Usually, we just run the executable file and play. But now let's look at the tree subfolders we see there, which is where the magic happens. Broadly speaking, in the "renpy" folder we can find the files containing the preprogrammed functions, and in the "lib" folder we have all the technical files that make the games run on the various operating systems. These two folders allow us to play the game without downloading any other specific software for it; we could say those folders turn Ren'Py games into self-executing programs.

The game itself is located within the "game" folder. Don't panic, this one is particularly complex:



There are several subfolders that we can overlook for now to focus on the individual files. When packaging their games, Ren'Py recommends creators to leave all the files in plain sight, as "What a Legend!" devs do. This allows us to see a whole series of apparently cloned files, some with a .rpy extension and others with a .rpyc extension. The number of files only depends on how the game's creator organizes their own work, as Ren'Py could perfectly run a game coded in a single file. Normally, though, there are just a few files (way less than in this example), but you can also find games that have their files stored in several subfolders within the game folder.

|Index|

1.2.- Three basic concepts and two Commandments

These files in the "game" folder are the **scripts**. It's in those scripts where developers include all the game's coding and texts. To do this, they just write in a text editor what they need to write and save that file with a Ren'Py-specific extension named .rpy.

So, using a basic program like Windows Notepad (or, preferably, a better one, like Atom or Notepad++, just to name two free and simple ones used in software programming), we can open those .rpy files and see what creating a Ren'Py game is all about. Still using "What a Legend!" as an example, if we open the file named "convo_oc.rpy", we can see this in its first lines:

```
1  # ==================== OLD Capital Base Conversations ==========================================
2  # Being stopped at the gate of the old capital ================================================
3  label convo_gate:
4      call hide_ui from _call_hide_ui_104
5      call silent from _call_silent_42
6
7      scene scene_oc_bridge_gate_talk
8      if current_hour == "Night" or current_hour == "Evening":
9          show kevin at g_cright:
10             xalign 0.6
11         show pov at m_left
12         with quickfade
13         show pov ewide bup mdislike hfshock hbshock
14         show kevin hbstop eangry
15         kevin "STOP!" with vpunch
16         show kevin hbpoint edoubt
17         show pov -mdislike hfneu1 hbneu1
18         kevin "Did you manage to get a passage permit?"
19         show pov mno bdoubt eneub hfhead
20         show kevin eneu hbneu
21         pov "Umm..."
22         show pov eneu bsad msad
23         show kevin esad hfspearmove
24         kevin "I'm sorry, buddy..."
25         show pov mcry eflat bneu hfneu1
26         kevin "...but no permit, no passage. That is the law."
```

Anything to the right of a # symbol is kind of 'invisible' for Ren'Py when running the game. So devs use that symbol to note down comments that will not appear in-game, but that allow them to navigate themselves through their code. Later we will see how useful this symbol can be for translators. In line 3 we can see a word, **label**, which is going to be of importance. Those labels are portions of the game's script. The size of those portions depends on each game developer, but they are generally a specific scene or a particular part of a scene, that will be activated and seen during the player's playthrough. Due to Python language conventions, everything happening in a label is coded with an indentation, and within a label there may be other indented blocks, always after a line finished in a colon : .

> **Ren'Py's First Commandment:** you shall respect indentations and shall NEVER mark them with the Tab key, but with the space bar (4 spaces, generally). If Ren'Py detects a Tab or an indentation mismatch somewhere in the scripts (translation scripts included) the game will not even launch.

If we scroll down the script, within this very same "convo_gate" label we'll see several functions to show and hide images and to determine which part of the conversation will be displayed based on the game's variables triggered by players during their playthrough. Also, we can see some quoted sentences: these are the dialogue lines that will be shown on screen. Those quoted sentences are the ones we will have to translate, and they are called **strings**, or text strings.

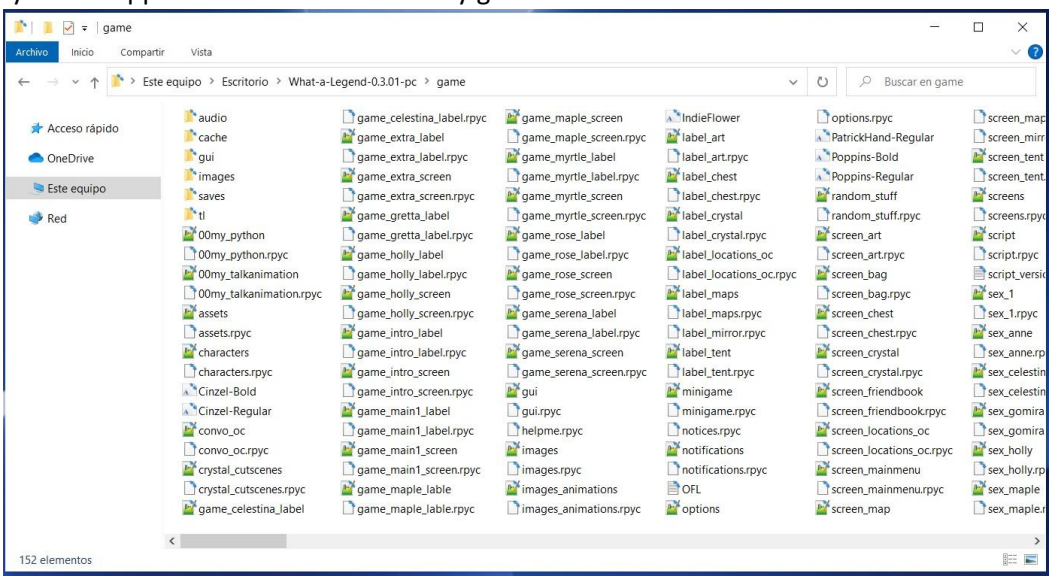
> **Ren'Py's Second Commandment:** you shall open strings with double quotes " and end them the same way ". A string without opening (or closing) quotes, or with some loose quotes in it, will prevent the game from launching.

When writing a text string, if we want to include a double quote that should be displayed on screen, we'll have to write those quotes preceded by a slash \" This way Ren'Py will identify them as part of the text.

|Index|

1.3.- How (I think) Ren'Py works? The Third Commandment

Assuming that I am not an IT expert and it is very likely that I will say something stupid, I will try to explain very quickly what happens when we run a Ren'Py game.



Do you remember the list of cloned files within the "game" folder? Well, they are not exactly clones: files with a .rpy extension are plain text files where game's dev write and code their game, but Ren'Py actually runs the .rpyc files, which are a compilation of their homonym .rpy's. When the game's dev launches the game for the first time in their computer, Ren'Py uses its algorithms to assign an identification to every coding element present in the .rpy files, according to their location within the script and its nature (text strings, variables, functions and so on), and those identifications are built in a compilation and saved as a .rpyc file with the same name than the original .rpy file. In order to create that initial compilation, an AST (abstract syntax tree) is generated. This AST contains those identifications we just mentioned and link them all together, and it's then used by Ren'Py as a guideline to create further compilations of that game project.

Every time the game is launched, Ren'Py looks for .rpy files within the "game" folder and, if they exist (they might not exist, as they aren't needed to run the game), a compilation process is started to build a .rpyc file. But, if there's already a .rpyc file with that name, Ren'Py will search the AST references in it, in order to update that file with the new content that has been added to the .rpy file since the last compilation were made. As a simplification, let's imagine that .rpy file's line number 3 is initially compiled in such a way that is assigned the number "3" by the game's AST, so it's stored in the .rpyc as just "3". If we edit the .rpy file so that this very same line is now the line 7, Ren'Py would be ready to compile it as "7", but then it will detect that this line is already present in the .rpyc file as "3" and won't change that. This way, the existing content that is still in use by the game is always kept as it was originally compiled, and the new content is compiled respecting those existing references, which have their roots in the very first compilation ever made.

Why am I telling you all this stuff? Because, if we delete the existing .rpyc files, Ren'Py will create brand new ones, but Ren'Py won't have the original AST references to use them as a guide, so it will take the current version of the .rpy files and will make a kind of "first compilation". And that means that the new links and references included in the new .rpyc won't be the same as before: taking our previous, absurd example, .rpy's line number 7 that in the old .rpyc file was still compiled as "3", as it came from a previous version, when creating a new .rpyc file it could perfectly be compiled now as "7", as Ren'Py will not find any restrictions nor old references to keep. The game will be launched perfectly and player's new games won't be affected, but some functions that use the AST references, such as loading old saves created while running the .rpyc's older versions, might be broken. And sometimes, translations could also be affected.

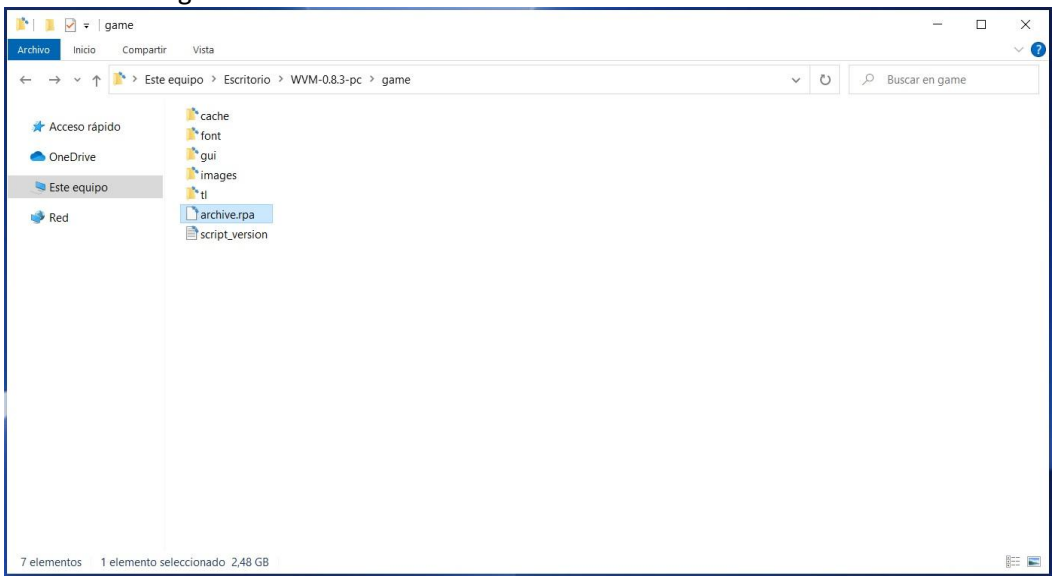
> **Ren'Py's Third Commandment:** you shall not delete the .rpyc files from the original game, as this can originate unintended problems to players.

|Index|

1.4.-UnRen and .rpa files

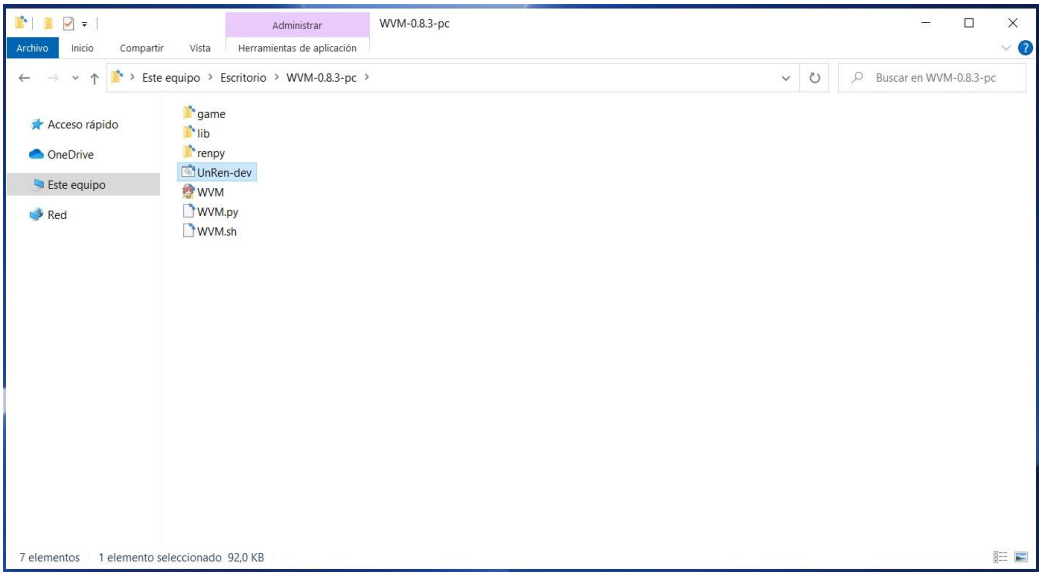
It's possible that, when you opened the "game" folder, you may have not seen anything of all that I mentioned in section 1.1. That's because, when it comes to build the game for its release, Ren'Py offers developers several options. The recommended one is to include all the individual scripts both in .rpy and .rpyc format, as we have seen in the "What a Legend!" example, but that's just a recommendation. As Ren'Py games only need their .rpyc scripts to work, some developers only include those .rpyc scripts in their builds. That way the game has a slightly smaller size and, maybe more importantly, players have no direct access to the game's code, as .rpyc files don't contain any comprehensible nor editable text.

And there's yet another possibility (a quite common one, I'd say), as we may find out that in the "game" folder there are no scripts with either .rpy nor .rpyc extensions, as the developer has chosen another option offered by Ren'Py to build the game: compressing the scripts and other files (such as images) into a .rpa file that is basically a container folder. That way, instead of a list of .rpy and .rpyc scripts, in the "game" folder we might have something like this:

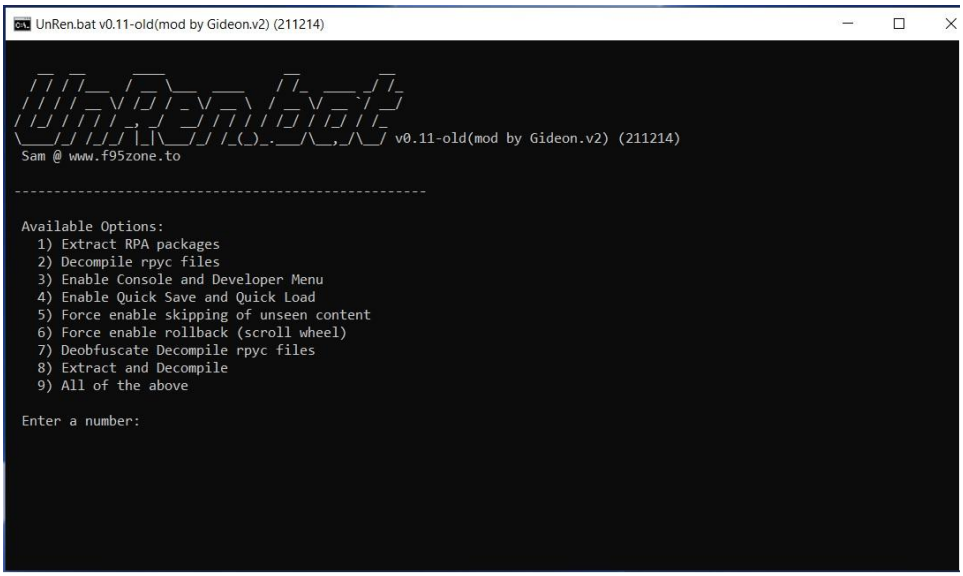


First of all, we can confirm that, as you probably already knew, this is not a problem. The game will run perfectly as it is, but in order to translate it we will have to give some extra detour, since we **ALWAYS** need the scripts in .rpy format to start the translation process. Fortunately, there are several tools that perform the necessary tasks for it without us having to learn Python language. For its simplicity, I think the handiest is **UnRen**. LINK *(beware: link to a forum with plenty of adult content).*

Once downloaded, we extract UnRen from its .zip and paste it into the root folder of the game we'd like to translate, at the same level where the game executable is. I guess it's easier to understand if you see it:



Then it's only a matter of running it and select what we want to do. This is UnRen's start screen:

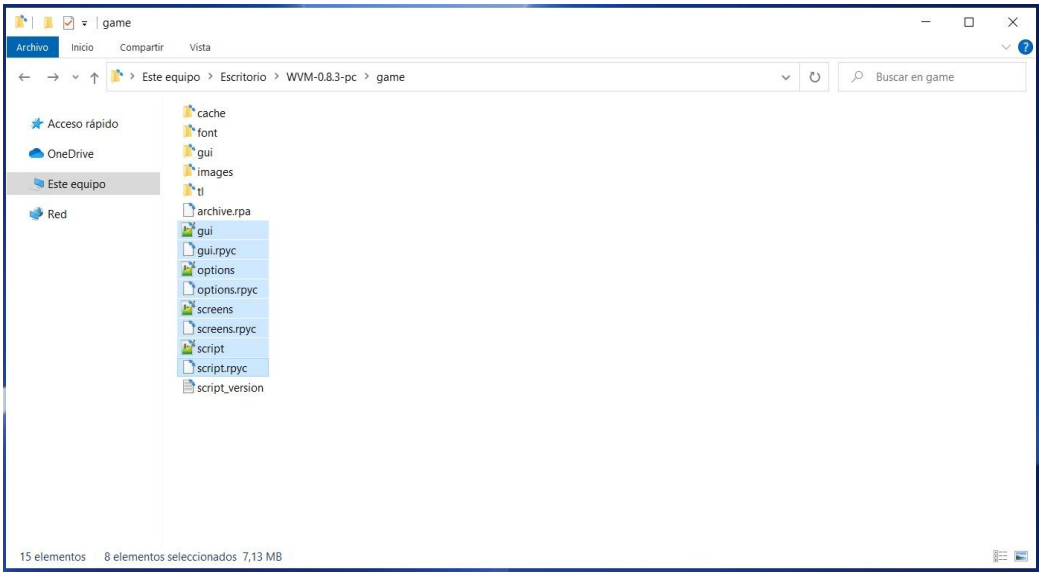


As you can see, here we are only going to use the keyboard. We just need to press the key with the number that appears next to the action we want it to perform.

Option 1) will "unzip" the .rpa file, extracting all its content onto the "game" folder. Once it's done, if we can already see .rpy files, that's great, we won't need to do anything else. Those scripts are the original ones written by game's dev, and they'll contain all the comments (if any) jotted down by the game's creator after a # symbol (those comments can be quite useful as they might allow us to get a better understanding of a certain string we don't exactly know how to correctly translate). It may be the case, though, that within that .rpa file there were only scripts with a .rpyc extension, so, after finishing that step 1), we would have to ask UnRen to execute option 2), which consists in decompiling the .rpyc files to create some scripts in .rpy format that we could already use for our translation – note that in some recent games, we might even have to force that decompilation with option 7). But this decompiled .rpy files are NOT the original ones: they are generated by UnRen thanks to the specific AST identified in the .rpyc files and they will only contain the basic info needed to make the game work (and to let us translate it), but developer's comments won't be present, as those comments are never compiled in the .rpyc files.

We can simply select option 8), which in one single step does the same as 1) and 2); or even 9), which also enables the option to open the game's console and developer menu (that's useful to examine and modify variables, to access specific parts of the script, and also to auto-reload the game when we are editing our translation) as well as the rest of listed options. But I'd recommend going step by step, because, if the orginal .rpy files were built into the .rpa container, UnRen will extract them while executing step 1) but then will overwrite them when forcing the decompilation of .rpyc files in step 2), and we'll lose dev's comments.

Well, we let UnRen to do its job and then, if we go back to the "game" folder, we will see that, where previously there was only a file with .rpa extension, now we'll have all the scripts that were compressed inside it. Depending on the game, we might even find some extra folders which may contain scripts too.

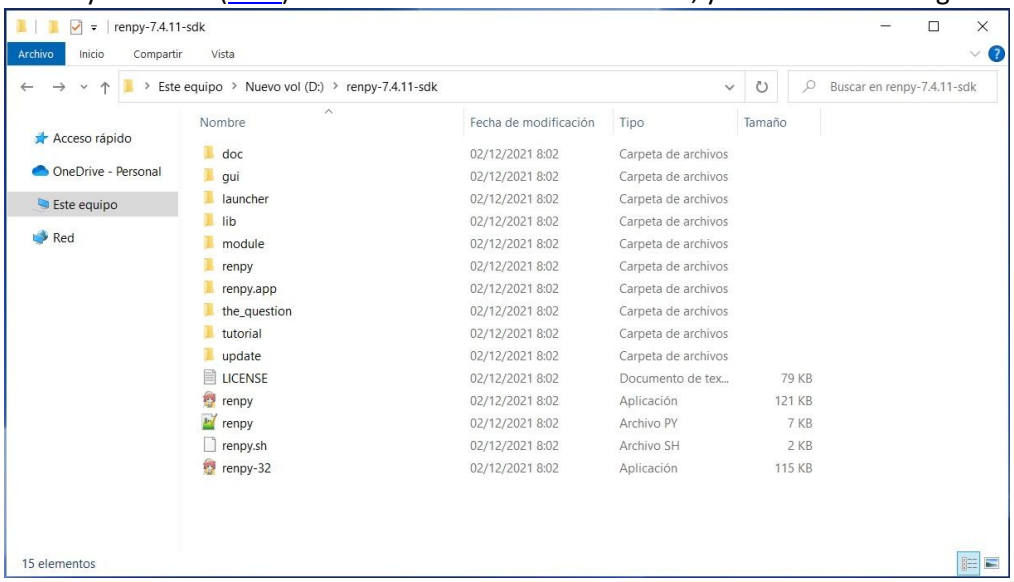


Now we could start translating. But, firstly, let me use a paragraph to explain what is going to happen with the game. Right now, in the “game” folder there are scripts with .rpy extension and scripts with .rpyc extension but, in addition, inside the "archive.rpa" file there will be those very same .rpyc scripts (and perhaps even the .rpy ones, plus all the pics and videos that have been also extracted). So now we do have cloned files. Fortunately, Ren'Py is designed to solve these duplication’s issues in an easy way: **if two scripts with identical name are found, Ren’Py will run the most recently edited one**. And, in this case, the most recent ones are those we’ve just extracted with UnRen. Therefore, we could safely delete the "archive.rpa" file. But, observing the Third Commandment, we should never delete the .rpyc files extracted from that file.

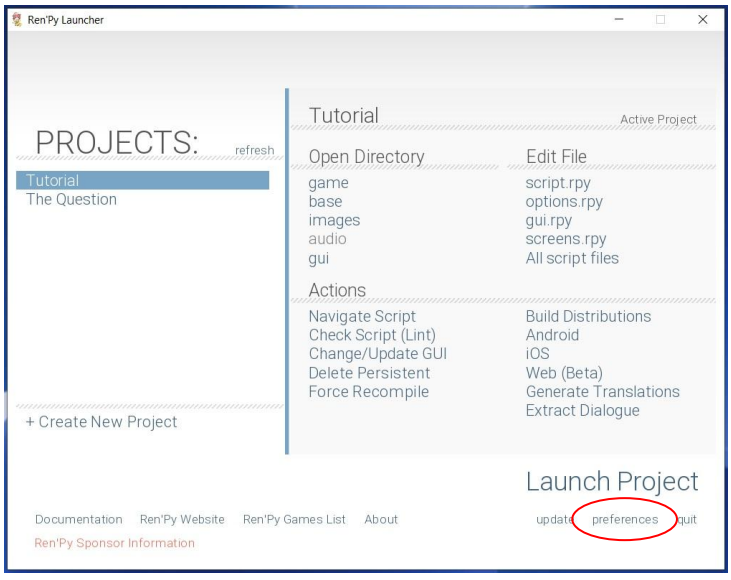
|Index|

1.4.- Ren'Py SDK

As we have seen, Ren'Py games can be played without installing any extra software to run them. They are self-executable and also allow us to modify their scripts with a simple text editor. Logically, however, to create them you do need a specific software. **Ren'Py SDK** is the application that allows us to create Ren'Py games - and their translations too. So the first step to translate any Ren’Py game is to download this app. It’s free, clean and fully trustable (LINK). Once downloaded and extracted, you’ll see something like this:

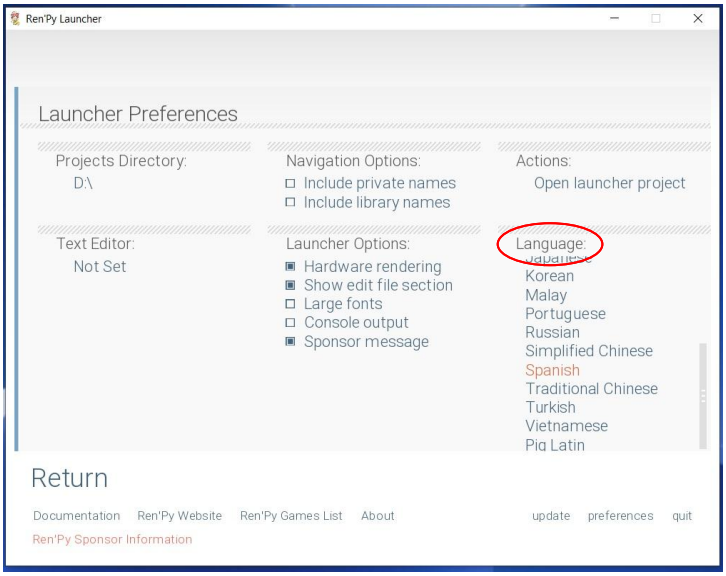


That’s a very similar structure to what we see in games, with several subfolders and, at least in Windows version, two .exe files (the main, generic one and a -32 bits one, intended for older OS such XP and Vista). We just need to run the appropriate “renpy” exe and start working with it. After setting up the software for the first time, we’ll get to the main screen. Under the “Projects” section we’ll see two titles: “Tutorial” and “The Question”; both games are tutorials that will allow us to learn some Ren’Py concepts and utilities. It doesn’t hurt to take a look at them, but they do not help that much to the translation process.



So, first of all, and just in case the Ren’Py SDK is still being displayed in English after the initial installation and we don’t like it, we’ll need to change its language in order to work in ours. To do that, we first click on “preferences” at the bottom right.

Under the label “Language”, on the right side of this “preferences” screen we will see a list of available languages. Ren’Py SDK can be run in all those languages, so we just have to look for our desired one. As soon as we click on it, the screen’s language will change and it will stay like that until we change it again.



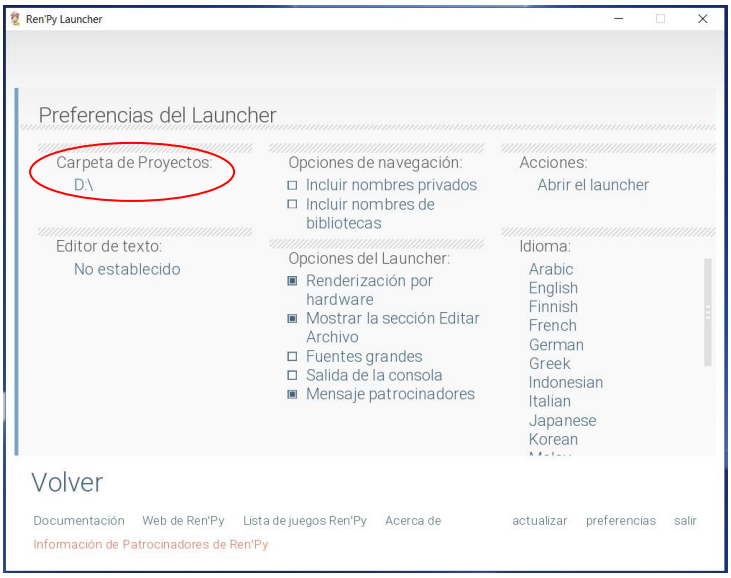
|Index|

2.- LET’S TRANSLATE

Either because the developer kept it easy for us, or because we have fight our way to achieve it, at this point we have our game with its .rpy scripts totally exposed and ready to be translated. Usually, our first impulse is to open those scripts and start rewriting their strings in our own language. Although this is not exactly wrong per se, when editing the original scripts we’re smashing the game’s creator work and also exponentially increasing the chances of generating a bug. But, above all, Ren'Py games include a translation supporting system: a whole bunch of functions that help us to extract the translatable strings and that make the translations to be correctly displayed without us tampering with the original code. Sure, sometimes we’ll still need to do a little bit of work on those original scripts, but the Ren’Py “sanctioned” translation method is way much cleaner than a direct rewriting of the scripts, and learning how to use it and how to overcome its limitations are the only reasons this guide exists. So, your choice.

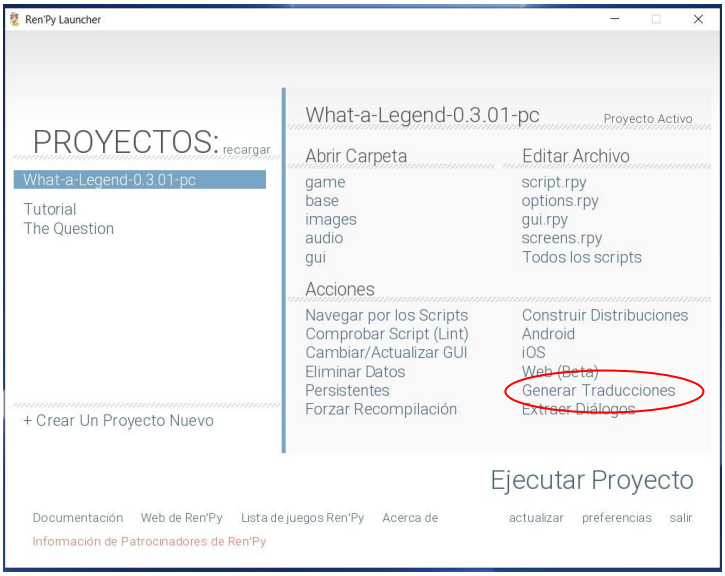
|Index|

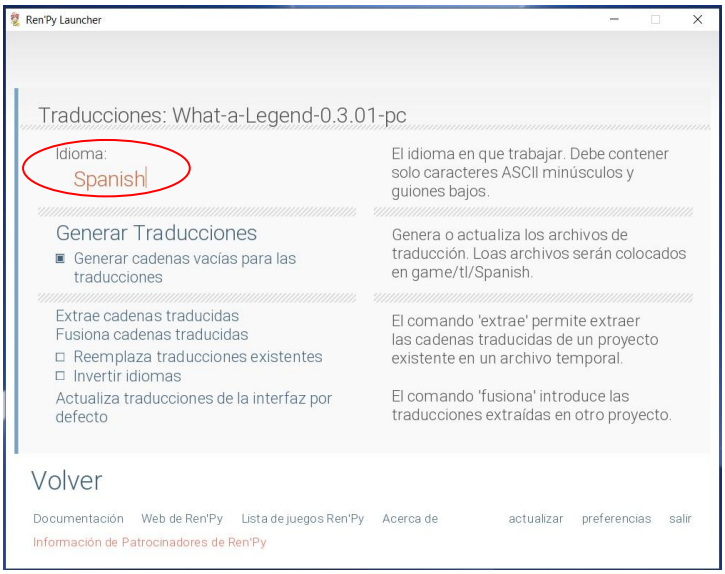
2.1.- Generating translation scripts (and learning the Fourth Commandment)



Once we got our Ren’Py SDK running, first step is to find the game we want to translate. We’ll need to click on “preferences” on the bottom right of the main screen and then, in the ‘Project’s Folder” section of this preferences screen, we’ll select the folder in which we have stored the root folder of the game. So, if our “What a Legend!” folder is located at My Documents, we’ll select My Documents. Then we click on return.

Now back on the main screen we'll see that all the games stored in that folder are listed under the "Projects" label, along with the two Ren'Py tutorials. We only need to select, by clicking on it, the game we want to translate, and then click on the "Generate Translations" button on the right side of the screen.





In the "Language" box of the next screen we must write the language we want to translate the game into. It's just an internal identification, so we can write whatever we want (note that no special characters are allowed, such as ñ or accented vowels). The most important thing is to remember what we have written here, and to keep always in mind Ren'Py's Fourth Commandment:

> **Ren'Py's Fourth Commandment:** In Ren'Py, a capital letter is not equal to a lowercase letter, never, under any circumstance.

So you can write Spanish, spanish, espanol, French, french, francaise, ship, yellow or Monkey. Whatever you want. But, obviously, it is advised to use a rational word that people could easily identify with your language. And, as I say, the most important thing is to always remember what you write there and how did you write it. For instance, I always use the word "Spanish", with a capital "S", and that's what you will see from now on in the examples provided.

Then we are presented with several options, but the only one that matters to us at this point is the one that reads "Generate empty strings for translations" (or whatever is said in your language). Here, as almost always, it's all a matter of tastes. When translating, we will always have a visible line with the original text to guide us, but if we select that option, the translation scripts will be generated with some blank lines under those original lines, where we will write our translation. If we do not select it, those lines will display again the game's original text and we'll have to delete those words and replace them with our translation.

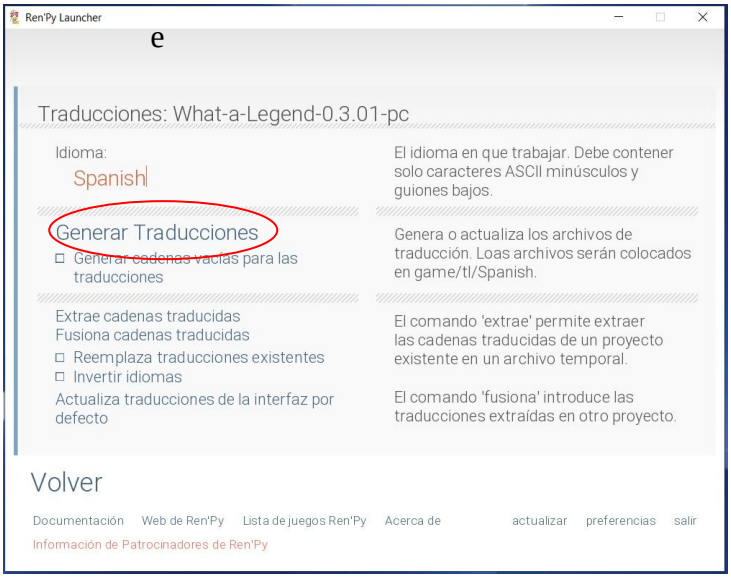
It may seem more convenient (it actually is) to generate empty strings, but, if somehow we forget to translate a line, when players reach that line in the game, no text at all will be displayed on screen. If we don't generate empty strings, the text would be displayed in the original language, which can be useful while testing an incomplete translation. This is especially important in menus, as it will allow us to have all the choice options active even if we haven't translated them yet. That's why I prefer NOT to generate empty strings, so I leave that box unchecked. But feel free to experiment and choose what suits best for you.

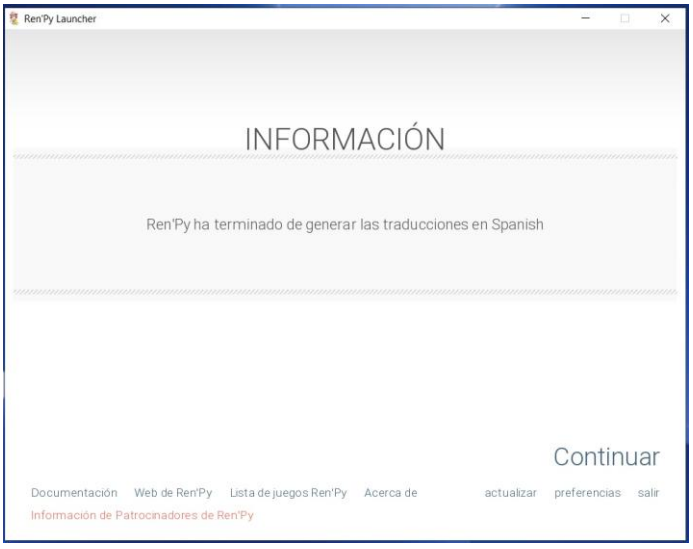
Here we can see a side by side comparison of how the translation files would look if we check that box to generate empty strings (left) and if we don't check it (right).

```
# TODO: Translation updated at 2020-12-11 13:16

# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "" with vpunch

# game/convo_oc.rpy:18
translate Spanish convo_gate_e5ccb99b:

    # kevin "Did you manage to get a passage permit?"
    kevin ""

# game/convo_oc.rpy:21
translate Spanish convo_gate_bc693870:

    # pov "Umm..."
    pov ""

# game/convo_oc.rpy:24
translate Spanish convo_gate_6a0bcf57:

    # kevin "I'm sorry, buddy..."
    kevin ""

# game/convo_oc.rpy:26
translate Spanish convo_gate_26193626:

    # kevin "...but no permit, no passage. That is the law."
    kevin ""
```

```
# TODO: Translation updated at 2020-12-11 13:41

# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "STOP!" with vpunch

# game/convo_oc.rpy:18
translate Spanish convo_gate_e5ccb99b:

    # kevin "Did you manage to get a passage permit?"
    kevin "Did you manage to get a passage permit?"

# game/convo_oc.rpy:21
translate Spanish convo_gate_bc693870:

    # pov "Umm..."
    pov "Umm..."

# game/convo_oc.rpy:24
translate Spanish convo_gate_6a0bcf57:

    # kevin "I'm sorry, buddy..."
    kevin "I'm sorry, buddy..."

# game/convo_oc.rpy:26
translate Spanish convo_gate_26193626:

    # kevin "...but no permit, no passage. That is the law."
    kevin "...but no permit, no passage. That is the law."
```

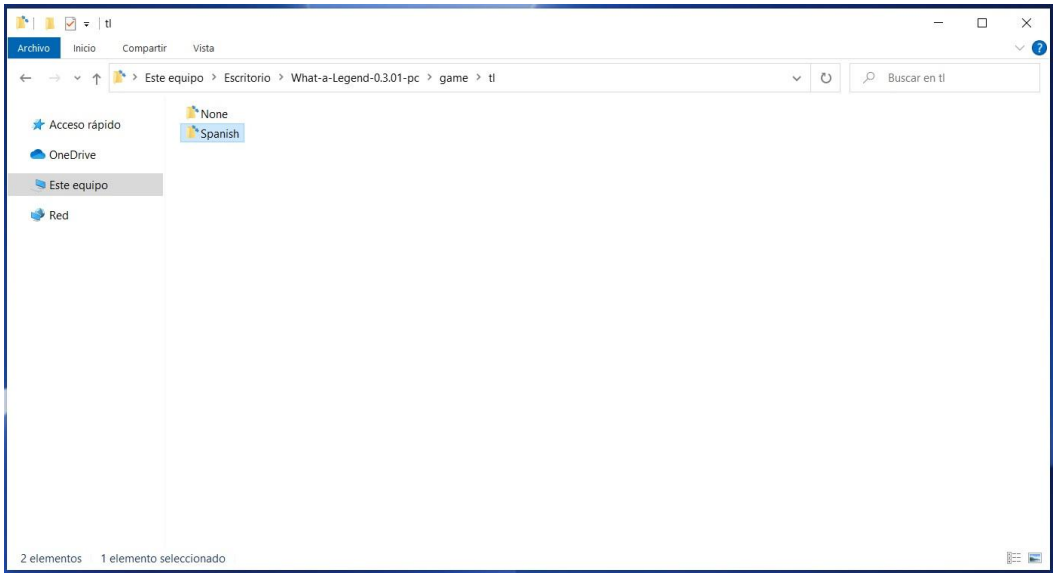
Now we just have to click on the "Generate Translations" button (the circled one). As the informative text in the right column says, by doing so Ren'Py SDK will create the translation scripts in the "game/tl/Spanish" folder (the name of said folder will be the one we have written in the "Language" box).



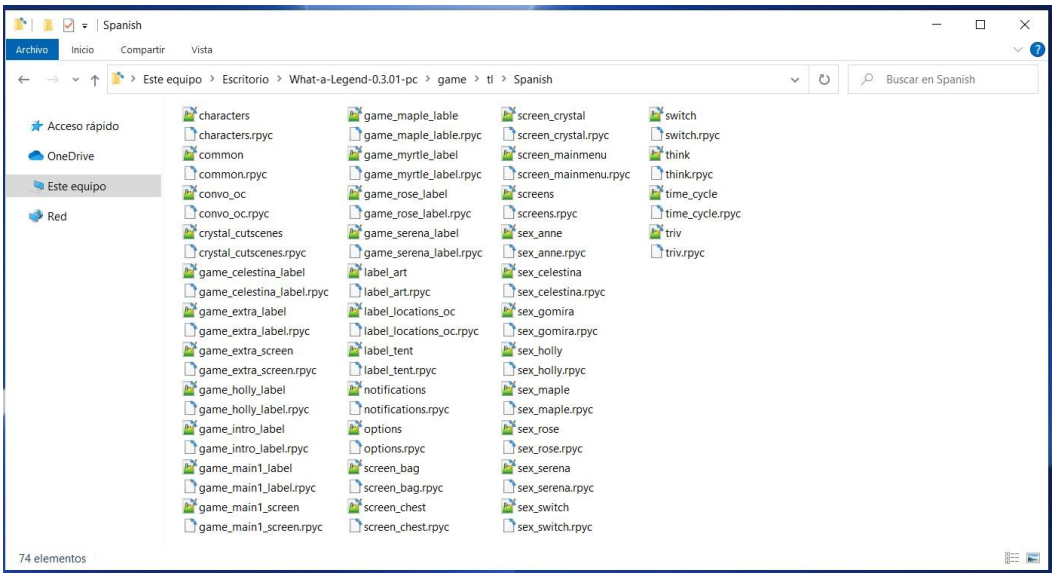


If there are no further problems, when Ren'Py SDK finish working its magic a message will be displayed telling us so. We can click on "Continue", to return to the main screen, or directly close the Ren'Py SDK app.

If we go now to the "game" folder in our game, we will see two other folders inside the "tl" subfolder: one named "None" (that was already there) and a recently created one with the name we wrote in the Ren'Py SDK's language box. In this case, "Spanish":



And within the "Spanish" folder we will find the translation scripts. What has Ren'Py SDK done to create them? Well, it has gone through all the original .rpy scripts, one by one in alphabetical order, and, whenever some translatable text was found in them, it has generated a .rpy file (and its matching .rpyc) with the original script's name, writing in it those strings that we will need to translate.



Also, an additional script has been generated, named "common.rpy" (the original common.rpy script is stored in the "tl/None" subfolder). This script contains the translatable commands of Ren'Py menus that are not specific to the game. For instance, the alert message that shows up when we close the game, asking us to confirm our choice, accessibility and developer menus, exception screens, and so on. It is a large file and a quite tedious one to translate, but there's a silver lining: usually, it's common to most games created with the same version of Ren'Py SDK, so it can be interchangeable from one game to another. So we can recycle a translated "common.rpy" script that we already have in a previous game. Of course, before replacing it, we should always check that the strings match, as sometimes there are small changes and we could accidentally break something or even duplicate a string, generating a bug.

At this point, we just need to open the .rpy scripts of the "tl/Spanish" folder with our favorite text editor and start working on them. We can change the name of these scripts (and even merge them in a single file),

since Ren'Py will look for translations string by string and does not care about the file they are stored in, but the standard practice is “don’t touch anything”: that way it will be easier to identify each translated script with its original script.

|Index|

2.2.- The two translation functions (and yet another Commandment)

It’s not unusual that, first time we start translating, we suddenly realize that the text to be translated isn’t following exactly the same order in which it’s written in the original script and/or displayed during the game. Don’t panic, that’s perfectly fine, because, when extracting translatable strings from the game’s scripts, Ren'Py divides them in two groups: one with all the strings that form what we’ll call the dialog block, and another group with all those strings that refer to menu options, variables, character names, etc. Why is that? Because each one of these groups is going to use a different extraction and translation function.

2.2.1- Dialog block and encryption codes: The strings that form the dialog block will be identified and extracted by Ren’Py SDK without problems. What it does with them, though, may cause us some nuisance, because they will be encrypted to save memory: using the MD5 hexadecimal encryption method to 8 bytes, each line is identified with an alphanumeric code that depends on the label where that text is located in the original script and also on the content of the line itself. We can see it better with an example. First of all, let’s remember the first lines of the “convo_oc.rpy” script of the game "What a Legend!”:

```
# ==================== OLD Capital Base Conversations ========================================
# Being stopped at the gate of the old capital ==============================================
label convo_gate:
    call hide_ui from _call_hide_ui_104
    call silent from _call_silent_42

    scene scene_oc_bridge_gate_talk
    if current_hour == "Night" or current_hour == "Evening":
        show kevin at g_cright:
            xalign 0.6
        show pov at m_left
        with quickfade
        show pov ewide bup mdislike hfshock hbshock
        show kevin hbstop eangry
        kevin "STOP!" with vpunch
        show kevin hbpoint edoubt
        show pov -mdislike hfneu1 hbneu1
        kevin "Did you manage to get a passage permit?"
        show pov mno bdoubt eneub hfhead
        show kevin eneu hbneu
        pov "Umm..."
        show pov eneu bsad msad
        show kevin esad hfspearmove
        kevin "I'm sorry, buddy..."
        show pov mcry eflat bneu hfneu1
        kevin "...but no permit, no passage. That is the law."
```

The first line with some translatable text in it (the first “string”) is on line 15 of the original script: under the label “convo_gate”, a character identified as Kevin says “STOP!” (with vpunch, an effect that will “shake” the screen). The rest of the code up to that point does not contain any text to be displayed on screen.

```
# TODO: Translation updated at 2020-12-11 13:16

# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "" with vpunch

# game/convo_oc.rpy:18
translate Spanish convo_gate_e5ccb99b:

    # kevin "Did you manage to get a passage permit?"
    kevin ""

# game/convo_oc.rpy:21
translate Spanish convo_gate_bc693870:

    # pov "Umm..."
    pov ""

# game/convo_oc.rpy:24
translate Spanish convo_gate_6a0bcf57:

    # kevin "I'm sorry, buddy..."
    kevin ""

# game/convo_oc.rpy:26
translate Spanish convo_gate_26193626:

    # kevin "...but no permit, no passage. That is the law."
    kevin ""
```

So, when generating the translation script with empty strings, Ren'Py has transformed all that info into what we can see on the picture at the left: all those functions and coding commands have been removed in order to extract only those lines with text strings in them, generating a four-line block for each one of those original strings. We can tell each block apart by its indentation: whenever we see a line starting on the left margin, we know that’s where a translation block begins.

Now, let's analyze those blocks:

```
# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "" with vpunch
```

As we already know, lines starting with a # symbol are merely informative comments and we could delete them with no consequences, although doing it wouldn't be my advice. First commented line tells us where that line is located in the original script: it's line number 15 in "convo_oc.rpy" file, which is stored in the "game" folder. The other line starting with # is the original text to translate, which Ren'Py displays here to let us know what we need to translate. Last line is where we'll write our translation (we should only translate the quoted text we see in the upper line, leaving the rest as it is). Had we generated translation scripts without empty strings, here we would see the original text again, and we'd need to overwrite it.

Second line is the key. This is what activates the translation function for this specific line of the original code. When we play with a translated enabled, Ren'Py is actually running the original .rpyc files from the "game" folder, but it has been ordered to display the translated texts instead of the original ones. This order tells Ren'Py that, for every string on the dialog block, it has to search the scripts looking for a line with the command "`translate`" plus the appropriate language ("`Spanish`", in this case, which is what we wrote when we generated the translation script with Ren'Py SDK), plus the encryption code assigned to the original line. This encryption code starts with the name of the label where that line is located ("convo_gate") and continues with the result of applying the MD5 system to that line's content: apparently in that encryption system, the whole `kevin "STOP!" with vpunch` line equals to `fdd309da`.

Now, if under that same "convo_gate" label there's an identical string (another "STOP!" said by this "kevin" character with vpunch), Ren'Py should assign it the same encryption code. But that would generate conflicts, so it will add a `_1` at the end of that second line's code, and so on. This way**, encryption codes are always unique for each string, even for literally identical strings**, and this means that every single string within the original dialog block will be extracted into the translation scripts. So, if there are ten identical strings, we'll need to write the same words ten times because we're actually translating ten different codes.

But what if we go to the original script and now write "Stop!" instead of "STOP!", or we change the character who's speaking? Well, then the translation we have for that line will stop working, because its MD5 code will be a different one (as we have changed the elements used to calculate it) and Ren'Py won't find this new code in the translation scripts, so the original text will be displayed instead. This causes some annoyances when translating new versions of games still under development, as devs usually correct minor typos and rewrite a few sentences here and there. Any minimal change in the original line will change its encryption code, so for Ren'Py it will be a completely different line and we'll have to translate it again. This will happen too if the label where a string is located is renamed, even if the line's content is not modified.

> **Ren'Py's Fifth Commandment:** Even the most minimal change made to the original line will invalidate that string's pre-existing translation.

Ren'Py includes now a feature that could avoid us this problem: if game's dev add the old MD5 id to that line's new edited version (for instance: `kevin "Stop!" with vpunch id convo_gate_fdd309da`), Ren'Py won't generate a new one and the old translation will still match. Sadly, very few devs know this.

2.2.2.- The rest of strings and the `old` and `new` commands: As it has been said, Ren'Py uses two different translation systems. In addition to the encryption code-based one, for the rest of strings that do not belong to the dialog block a simpler system of direct translation is used. What strings are we referring to? Well, I'm basically talking about the options that are presented to the player during the game, but also about character's names, hypothetical variables used in game which possible values are text-based, system menus like preferences or save/load screen, and so on.

When Ren'Py is running a translation, all those strings are going to be replaced on screen by direct substitution, without any encryption. To do that, when generating the translation scripts, Ren'Py SDK will group them all together at the bottom of the script, under a "`translate Spanish strings:`" command.

```
8001 translate Spanish strings:
8002
8003     # game/convo_oc.rpy:107
8004     old "Permit"
8005     new ""
8006
8007     # game/convo_oc.rpy:107
8008     old "Old Capital"
8009     new ""
8010
8011     # game/convo_oc.rpy:107
8012     old "Helping pixies"
8013     new ""
8014
8015     # game/convo_oc.rpy:107
8016     old "Dungeon"
8017     new ""
8018
8019     # game/convo_oc.rpy:107
8020     old "Go back"
8021     new ""
8022
8023     # game/convo_oc.rpy:320
8024     old "Passage permit"
8025     new ""
8026
8027     # game/convo_oc.rpy:320
8028     old "Challenge"
8029     new ""
```

The figure on the left shows the beginning of the second section of the "convo_oc" translation script from "What a Legend!". As you can see, there are no strange codes here. And if the "`translate Spanish`" command was applied before to each line, now all of them are translated using just one single command. There's still a first line with the symbol # to let us know where that string is located in the original script, but then it displays the original text with the "`old`" command, and after the "`new`" command we'll have to write its translation. Note that now the original text is not an informative line anymore, but a proper part of the translation function: this "`old`" value is what Ren'Py will look for, instead of some MD5 code.

We should take into account that, now, when generating the translations scripts, if there are several strings literally identical (even if they belong to different scripts and labels), Ren'Py SDK will only extract the first one it finds while looking through all the original scripts in alphabetical order (note the difference with what it does with the dialog block, when all strings are extracted and assigned different encryption codes even when lines are strictly identical). Then, when running the game, every string that **literally** matches the value of an "`old`" line will be replaced with the translation included on its "`new`" line. So, from the moment we write its translation in this script, we won't need to translate the string "Permit" anywhere else, and that translation will be displayed every time the literal expression "Permit" is mentioned in the original scripts.

While this can save us some work, it becomes an inconvenient when we come across identical strings that may have different meanings depending on the context. For example, think of a string like "Right": sometimes it could mean "Correct" and sometimes it can be referring to the right side of something, and maybe in our language we use a different word for each concept. But Ren'Py SDK would only extract the first "Right" string, so we could only translate it once and that one translation would always be displayed, so sometimes the translated text would not make any sense. Later we'll see how to solve this problem.

Theoretically speaking, this direct translation system could be used for absolutely every string in the game, but dialog block is usually very large and is made of much longer sentences that aren't frequently duplicated. So, when Ren'Py has to display a translation, searching and replacing thousands of encryption codes takes way less time than searching and replacing thousands of longer text strings. However, this second group of strings are usually much shorter, way less numerous, and they are more likely to appear duplicated in different parts of the scripts, so Ren'Py can afford itself to execute an specific search of their literal content without slowing down its performance. That's why two different translation functions are implemented.

Anyway, those are just some perfectly forgettable notes. What matters is that Ren'Py SDK identifies two different "string" types, with each one of them using a different translation system, and that's the reason why we will find them separately coded in the translation scripts: first we'll see all the strings that belong to the dialog block, and then all the strings from menus, options and variables.

|Index|

2.3.- Helping and/or replacing the Extractor

The reason behind explaining in the previous point the two types of Ren'Py's translation functions is that, sadly, Ren'Py SDK is not always able to detect absolutely every translatable string: although it will always

extract the whole dialog block (all the strings that will be translated thanks to their encryption code), it might not be able to correctly identify and extract all the strings that need to be translated using the direct method (fortunately, this problem won't effect in-game's menu choices, that will be perfectly extracted).

For instance, for Ren'Py SDK to be able to automatically extract characters' names, the game's developer should have defined those characters in a very specific way that, due mostly to ignorance, many devs do not use. But if they don't, we can do it.

As "What a Legend!" creators did their homework, making life much easier for us translators, for these examples we are going to use the other game that we already used to talk about .rpa files and UnRen. So let's see how the developer of "WVM" defines one of the characters involved in the story. In this case, in the file "script.rpy" he has created an "MC inner self" to indicate players that what we read in the textbox is a thought from our own character:

```
224 define mcm = Character ("Your thoughts",color="#FFFFFF", who_outlines=[ (2, "#000000") ], what_outlines=[ (2, "#000000") ])
```

This is a typical line of Ren'Py code. It starts with a command (`define`) that indicates what this specific line does, in this case defining a variable. That variable is assigned a name (`mcm`) that will be used in the rest of the script to identify it, and it is ordered to behave as a character type variable (`Character`). Now Ren'Py knows that, whenever it finds the letters `mcm` before a string, it has to display a name above the textbox so the player knows which character is talking. And what will be displayed there? What appears afterwards in parentheses: the character's name ("`Your thoughts`"), in a specific color, with a line surrounding the letters to highlight them.

Obviously, "Your thoughts" " is a string that we should translate, but, if we generate the translation scripts with Ren'Py SDK, that string won't be anywhere to be seen. Ren'Py's mysteries. What can we do, then? Well, we have three options. The first option, which we plainly discard, is to just leave it like that, so this string will always be displayed in English. The second option, much preferred, is to let Ren'Py SDK understand that the quoted text is actually a translatable text string and not just some piece of code like color's tag. Look:

```
224 define mcm = Character (_("Your thoughts"),color="#FFFFFF", who_outlines=[ (2, "#000000") ], what_outlines=[ (2, "#000000") ])
```

We have edited the original script to put the string "Your thoughts" between parentheses and we have typed an underscore _ before those parentheses, so it looks like this: _("Your thoughts"). This combination of `_( )` symbols will allow Ren'Py SDK to identify the quoted text inside parentheses as a translatable string. And if we generate now the translation scripts, we will find this block in the direct translation's section:

```
translate Spanish strings:

    # script.rpy:224
    old "Your thoughts"
    new "Tus pensamientos"
```

That's the result after translating it into Spanish, of course. But the important thing is that, hadn't we edited the original script to replace "`Your thoughts`" for `_("Your thoughts")`, this string would not automatically appear in the translation scripts.

That doesn't mean that we could have never translated it: it being a direct translation, we could always write it in our translation script without Ren'Py SDK's help. This is the last of the three options I mentioned before: instead of editing the original scripts, we can manually write in the translation scripts all the translatable strings that Ren'Py SDK won't detect. In order to do that, we just need to go to the translation script's section that starts with the function `translate Spanish strings:` and write a translation function with the same format we've seen above: always respecting the 4 spaces indentation, as well as quotes as the first two Ren'Py's Commandments say, we would write the command `old` followed by the string to translate (scrupulously respecting its writing, as Ren'Py's fourth and fifth Commandments say), and then, in the line below, we would write the `new` command followed by our translation. The line that starts with a # symbol wouldn't be necessary as it is merely informative.

Personally, I'm used to review and edit the original files first, including the `_( )` symbol where needed, so that Ren'Py SDK can then do a full extraction job, only resorting to the manual method to add some particular string that I forgot to include in that first edition and it's displayed untranslated when playing the game. Other translators prefer to generate translation scripts first, without editing the original files, and then manually add every string that Ren'Py SDK left unextracted. But it's also possible to edit the original script and then regenerate the translation scripts as many times as we want or need: if we tell Ren'Py SDK to use literally the very same language as before and to NOT replace existing translations, our translations scripts will be updated with the newly detected strings, that will be added at the bottom of the file.

Whether you prefer one method or another, in order to achieve a full translation of the game you'll need to go through all the original scripts and look for these types of commands, as their strings won't be automatically detected by Ren'Py SDK unless the game's dev had included them between the `_( )` symbol:

- `Character("...")`, used to define characters, as we have seen in the example
- `text "..."`, used to display text on a specific screen, outside of the textbox
- `textbutton "..."`, used for texts that perform an action when clicking on them
- `tooltip '...'`, used to display a pop-up message when hovering over a point
- `renpy.input("...")`, used to allow in-game typing (to let players to name characters, usually)
- `$ renpy.notify('...', 'unlock')`, used to display messages after completing an action

In addition, we have variables' values that are text strings. Here we can find several possibilities, but none of them will be automatically extracted by Ren'Py SDK unless they are found between the `_( )` symbol:

- `default` *variable_name* = `"..."`
- `define` *variable_name* = `"..."`
- `$` *variable_name* = `"..."`

So it's a matter of wrapping all those quotation marks with the `_( )` symbol and generate the translation scripts, or copying its content (quotations included) directly into a translation script (no matter which one) with the `old` command, writing its translation below with the `new` command. Note that you have to copy everything that appears in quotation marks, not just the text to be translated, since sometimes there are tags inside `{}` symbols that are used to display the text in **bold**, *italics*, with another font or in a different size, etc. Those tags are also part of the string that Ren'Py will search and replace, so they have to appear behind the `old` command in order to help Ren'Py to correctly indentify that exact string, literally. And sometimes we'll still need to do something else if we want the translation to be correctly displayed.

Finally, just an extra remark: the `_( )` symbol **is only used by Ren'Py SDK to extract the strings** in parentheses and is not needed for the translation to be displayed on screen. Therefore, when translating a game's new update, editing the original scripts again would not be necessary if those strings are already present (and translated with the `old` and `new` commands) in the old translation scripts. Also, there's no need to add in our patch the original scripts we edited to include the `_( )` symbol, as players don't need that symbol for their translation to be correctly displayed.

|Index|

2.4.- Translating updates

In these times of Patreon it's very common for games to not be released as completed projects, but in episodic format, and we are forced to update our translations as new versions are released. The procedure doesn't have major complications... unless we want to do it the way Ren'Py SDK suggests. The official Ren'Py app offers us an option to extract the existing translations of a game's old version and insert them into the new one, updating the translation scripts with the new content thanks to the extract and merge translated strings commands. But the sad truth is that, the moment the new version's original script doesn't exactly match the old one, Ren'Py SDK will generate a brand new translation script, without merging the existing

translation, so we'll need to repeat our work. So, the most effective way to do it is to just follow these steps:

1. Download the game's new version.
2. Extract the .rpy scripts with UnRen, if needed.
3. Move the "game/tl/Spanish" folder from our old version (that is, our old translation folder) into the game's new version folder, using a simple copy-paste action.
4. Generate the new version's translation scripts with Ren'Py SDK.

If in point 4 we choose the SAME language of the previous version (that is, if we write in Ren'Py SDK's language box the very same word we used before, so "Spanish", in my case), and we **DON'T** check the "Replace existing translations" option, Ren'Py SDK will simply update the existing translation scripts with the strings for which it cannot find a valid translation. At the bottom of each translation script, Ren'Py SDK will add both the game's new translatable strings (the new version's content) and those strings that have been modified since the last time translation scripts were generated, creating two blocks again (first we will see the dialog block's strings and then menu options and other strings to be translated by the direct translation method). And logically, if there's any new original script, a new translation script will be generated too. But, as I said, at the end of the process only the untranslated strings will be added, keeping our previous work intact.

Now, if we sort the files from the "game/tl/Spanish" folder by date modified, we'll see which of them have just been created or modified, which means that those files contain some new strings to translate, while the other files with an older date are perfectly valid as they are.

|Index|

2.5.- Translating translations

Sometimes, games are released in a certain language that makes impossible for us to create a proper translation, due basically to our own ignorance of that language. We can always resort to some machine translator, but we all know that machine translations are vastly improvable and, even if we take our time to polish that translation, chances are that we won't be able to fully understand and translate some idioms and expressions. But it may be also possible that someone with a better grasp of that language had already translated the game into their own language, or maybe even the same game developer is releasing their game with a translation already implemented; if that second language is a language we do know, we can benefit from that previous work to create our translation.

Supposing that the previous translator has followed the translation method designed by Ren'Py's developers (that is, the one I'm trying to explain in this guide), there should be a "game/tl/MyLanguage" folder (where "MyLanguage" is the name that this other translator typed in their Ren'Py SDK) with all the translation scripts and, obviously, the game's strings already translated into that language. As we should already know, those translation scripts will be divided in two big blocks: one for the dialog strings and then another group for variables, menu choices and so on. In those scripts we'll see the same translation functions we've already explained, only that they will read "`MyLanguage`" where in my examples we read "`Spanish`" (and, of course, we'll also see the texts translated into that language).

In this case, we can forget about Ren'Py SDK: to generate our translation scripts we just need to create a new subfolder inside the "game/tl" folder and name it "Spanish" (or our desired language) and copy-paste into this new folder all the scripts present in the "game/tl/MyLanguage" subfolder. Then we only need to edit these scripts inside our language folder to replace "MyLanguage" with "Spanish" in every translation function, something that can be easily done with our text editor's search and replace function.

Now, for dialog lines that are translated using encryption codes, each `translate MyLanguage XXXXXXXX:` function will become `translate Spanish XXXXXXXX:` (remember that each line's encryption code, represented by XXXXXXXX in this example, only depends on the original label and line's

content, so it's identical in every language). And the translation function for strings translated by direct substitution will become `translate Spanish strings:` instead of `translate MyLanguage strings:`

And that's all: by overwriting in our language the texts written in "MyLanguage", a language we do know and understand, we will get our translation ready to be implemented in the game.

|Index|

2.6.- The switch languages option

Lastly, or maybe firstly before we start translating, we have to stop and think about how are we going to activate our translation in the game once we finish our work. First thing we should know is that, by default, every Ren'Py game will be initially launched in the original language it was written, a language that is internally identified as `None` (whatever the actual language is named); from that first launching, and every time the game is launched, the last language selected will be displayed. For instance, first time we launch a game made in English, the texts will be displayed in English, but if we switch languages to Spanish in game and then quit, next time we launch the game it will start in Spanish.

There are, however, some functions that can alter this default behavior, and we are going to mention them now. First one is used to order Ren'Py to launch the game initially in another language instead of the original `None` language. To do so, we can open any script (we can do it even in the translation scripts) and write this line of code, without indentation:

```
define config.default_language = "Spanish"
```

I wrote Spanish because that's the name I gave to my language when generating translation scripts; you should use the word you used back then. This short line will make the game to be initially launched in Spanish but only if this is the first launching ever, all while respecting Ren'Py's ordinary behavior: so, if player switches back to the game's original language during their playthrough, that language will be stored as default for the next time the game is launched. The problem being that, if the player has already tried out the game before patching it with our translation (so, before this new command becomes part of their game's code), the command is totally useless, as there are another language already stored by Ren'Py as 'last used'. So the game won't be launched in Spanish, but in that other language instead.

It could also happen that the developer had created their game in their own language and then released it with an in-build translation into another language, making this second language the default launching one by using this very same `config.default_language`, so somewhere in the original scripts (in gui.rpy or options.rpy script, most likely, although the exact location is not important) we'll find that coding line we have just seen. We could edit that line, replacing that original language for ours (or, better yet, we could override it with the `init` command we'll see later on), but, anyway, if player has already launched the game before installing our translation patch, the launching language won't change automatically next time they run the game.

So, if we want the game to be always launched in our language, even if the last time it was played in some other language, we'll have to write this other line of code, anywhere in the scripts, without indentation:

```
define config.language = "Spanish"
```

Now the game will ALWAYS start in Spanish, even if the player has chosen another language during their last playing session. So the `config.language` variable prevails over `config.default_language` but also disables the default system used by Ren'Py to decide in which language the game should start once it has been played for the first time.

Anyway, even if we decide to force the game to be launched in our language, we should always offer the

player an option to switch languages during their playthrough, and the most logical approach is including that option in the Preferences menu. Assuming the game uses the Ren'Py's default preferences screen, if we take a look at the original "screens.rpy" script we'll see this `screen preferences():` section:

```
759 ## Preferences screen ##################################################################
760 ##
761 ## The preferences screen allows the player to configure the game to better suit
762 ## themselves.
763 ##
764 ## https://www.renpy.org/doc/html/screen_special.html#preferences
765
766 screen preferences():
767
768     tag menu
769
770     use game_menu(_("Preferences"), scroll="viewport"):
771
772         vbox:
773
774             hbox:
775                 box_wrap True
776
777                 if renpy.variant("pc") or renpy.variant("web"):
778
779                     vbox:
780                         style_prefix "radio"
781                         label _("Display")
782                         textbutton _("Window") action Preference("display", "window")
783                         textbutton _("Fullscreen") action Preference("display", "fullscreen")
784
785                 vbox:
786                     style_prefix "radio"
787                     label _("Rollback Side")
788                     textbutton _("Disable") action Preference("rollback side", "disable")
789                     textbutton _("Left") action Preference("rollback side", "left")
790                     textbutton _("Right") action Preference("rollback side", "right")
791
792                 vbox:
793                     style_prefix "check"
794                     label _("Skip")
795                     textbutton _("Unseen Text") action Preference("skip", "toggle")
796                     textbutton _("After Choices") action Preference("after choices", "toggle")
797                     textbutton _("Transitions") action InvertSelected(Preference("transitions", "toggle"))
798
799                 ## Additional vboxes of type "radio_pref" or "check_pref" can be
800                 ## added here, to add additional creator-defined preferences.
801
802             null height (4 * gui.pref_spacing)
```

In this guide's first version, I said that we should edit that code to include a switch languages option by adding these lines at the same indentation level we see the code is using for "Rollback Side" and "Skip" options (remember to always type in this indentation with the space bar):

```
vbox:
    style_prefix "radio"
    label _("Language")
    textbutton _("English") action Language(None)
    textbutton _("Español") action Language("Spanish")
```

Obviously, this `_("English")` textbutton is valid when the game's original language is English; if the game uses some other language we should write that language's name without touching anything else, as this will always be the `None` language. And, as I've said, this code is only valid for a classic preferences screen, but it might not be useful if the game's uses a custom one. In that case, you should take a closer look at the other elements you see in that preferences screen and try to replicate its style. The easiest solution is copy pasting one of those elements and then changing their textbuttons and actions.

But the most important thing to know is that, in case we have a screen with "textbuttons" (or "imagebuttons") we have to add an option that, when clicked, activates the `action Language("Spanish")` command, where `"Spanish"` is the name of the language we entered in Ren'Py SDK when generating the translations scripts. If we misspell it (`"spanish"`, for instance) Ren'Py will not detect our translation and the button will become inactive. It's also important to note that the word `None` is never quoted, but all the other languages are.

As you can see, in that piece of code I wrote there are several strings wrapped with the `_( )` symbol, which will allow Ren'Py SDK to detect and extract those strings when generating the translation scripts. But we should only translate the string "Language", because all languages' names should be always displayed in their own language, no matter in which language we are playing the game. I mean, if we don't translate "English" and, by chance, some English-speaking person who can't read our language stumbles upon our translated version of the game, this person will easily identify that word and will probably realize that this is where they can change the language back to English. Just imagine how relieved you'd be if you find your

language's name written in the middle of a screen full of strange symbols from an unknown language, and you'll understand why you should keep languages' names in their original language form.

There are, of course, a myriad of different options to add a switch languages function out of the Ren'Py's default preferences screen. So many, in fact, that I can't list and detail them all. But, eventually, all of them are going to use one of this two methods: either theyl add the `action Language("Spanish")` command we have just seen above, or theyl activate a `$ renpy.change_language("Spanish")` variable (this method is often used when the switch languages option is presented as a question in-game). But I really encourage you to do a little bit of reverse engineering and to dig in Ren'Py online documentation and coding forums, in order to learn about how to create and edit screens, splashscreens, imagemaps, keymaps, buttons… There are lots of relatively complex but visually attractive options out there to explore.

That said, after one more year of practicing and learning, I've come to the conclusion that it's best to replicate the preferences screen, so that the one with this switch languages option is displayed instead of the original one while the player is using the translation, but leaving intact the original screen in case that person decides to delete the translation and get the game's original version back. So now I'm using a specific .rpy script where I include this custom screen as well as other fixes we'll see later on.

|Index|

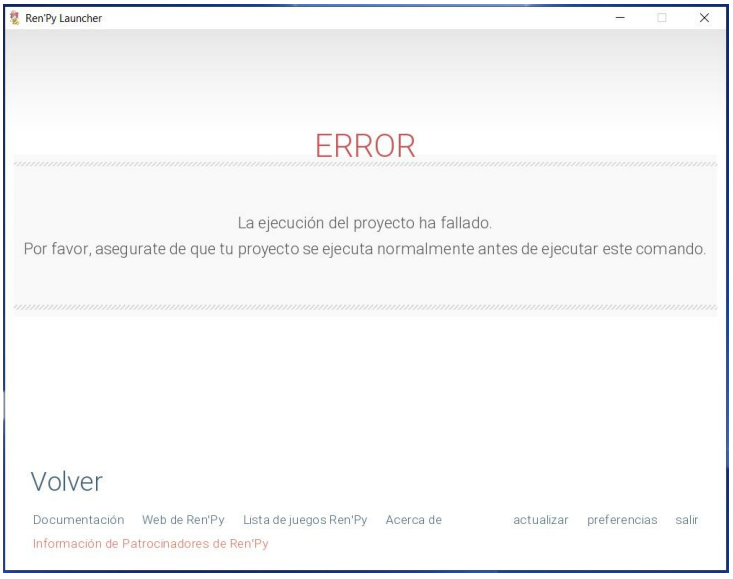
3.- WHY CAN'T I MAKE IT WORK

After telling you all of this, if you have been able to create your own translation and everything is working properly, then congratulations for being such an attentive pupil, and also for having chosen for your practices a quite simple game. Because the most common thing to happen is that, when playing your translated version, every now and then some texts will still appear untranslated, even though you may have translated them in the translation scripts. Next, I will try to explain some of the most common incidents.

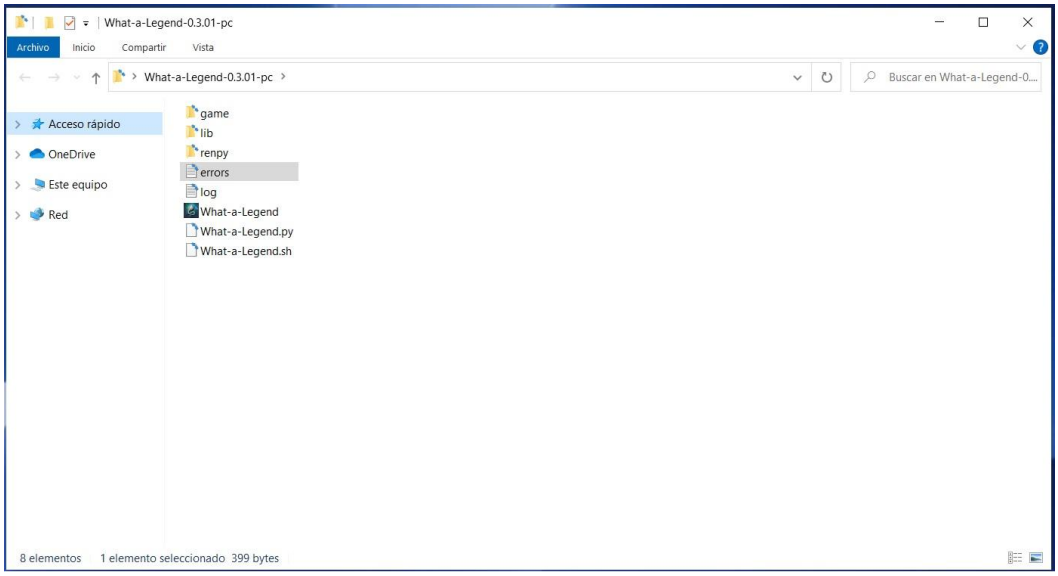
|Index|

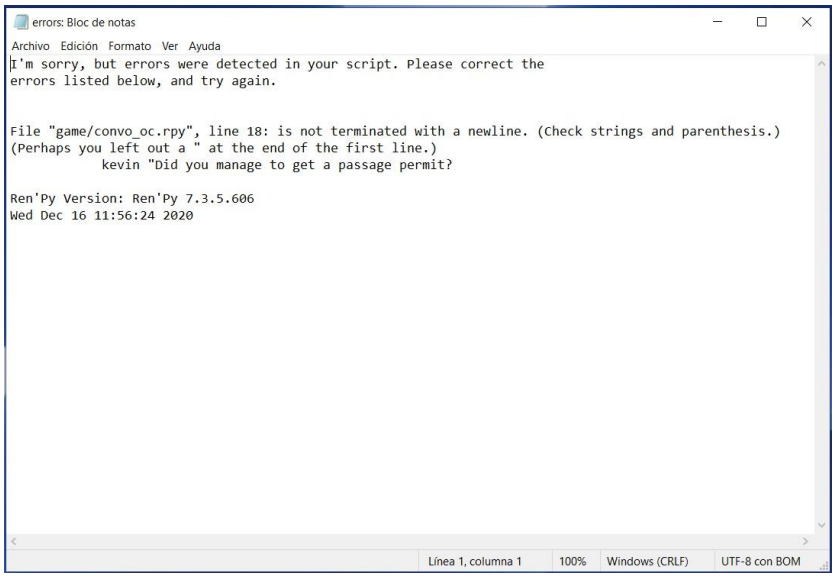
3.1.- Bugs and errors detection

First things first, we should familiarize ourselves with the screen Ren'Py uses to warn us when something is wrong. In an ideal world, games would be released bug-free and without any critical errors, so the first fatal screenshot a translator might face should be the one that warns us that Ren'Py SDK could not generate the translation scripts. Which can only mean that we have made some mistake(s) while editing the original .rpy scripts before generating translation files, or maybe that we are using a Ren'Py SDK version which is older than the one used by game's dev and some newly implemented functions have broken the translation scripts. Obviously, this second option can be easily fixed within a couple of minutes, just by downloading the last Ren'Py SDK version available, which is always recommended.



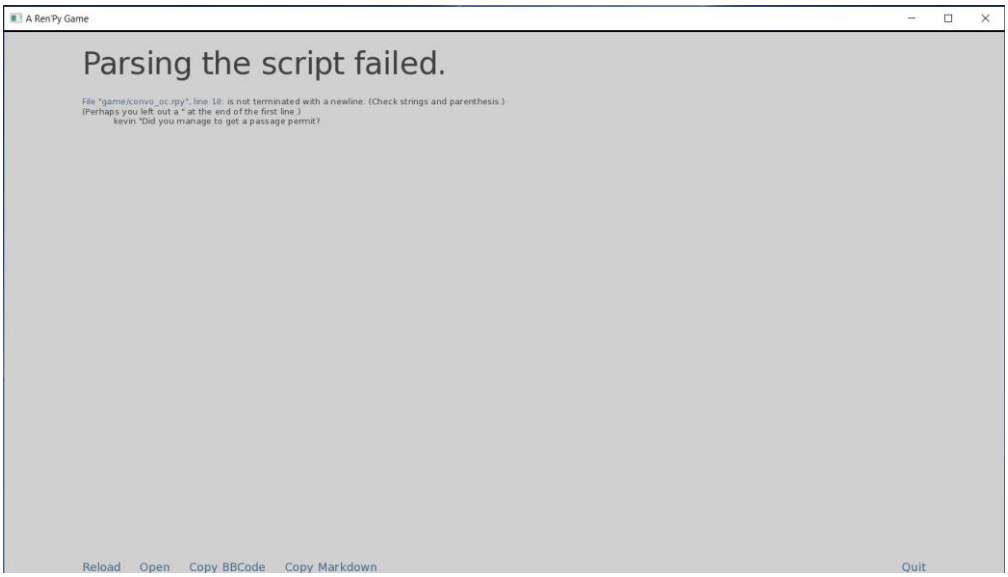
If we suspect the problem is due to us making some mistake when editing the scripts, we should check the game's root folder to see what has happened. There, next to the game's .exe file, multiple .txt files will appear as we can see in this image:



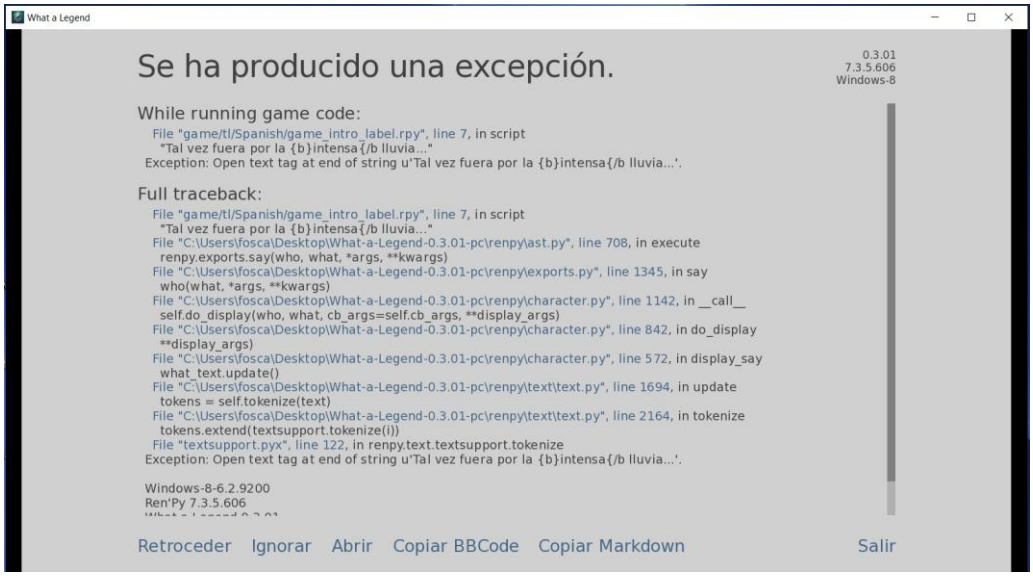


We should focus on the one named "errors.txt", which can be opened with any text editor. There we will see an error message showing the line or lines of code that caused the problem. In this case, it's something as usual as forgetting to close quotation marks (on line 18 of the "convo_oc" script), but it could have been an indentation marked with the Tab key, or an unclosed {tag}, or whatever. We only need to open that script, fix it, save changes and return to Ren'Py SDK to try again what we were doing before.

Had we tried to run the game by clicking on its executable, this message would have been also displayed. In that case the screenshot would be something like this, with the same information of the "errors.txt" file:



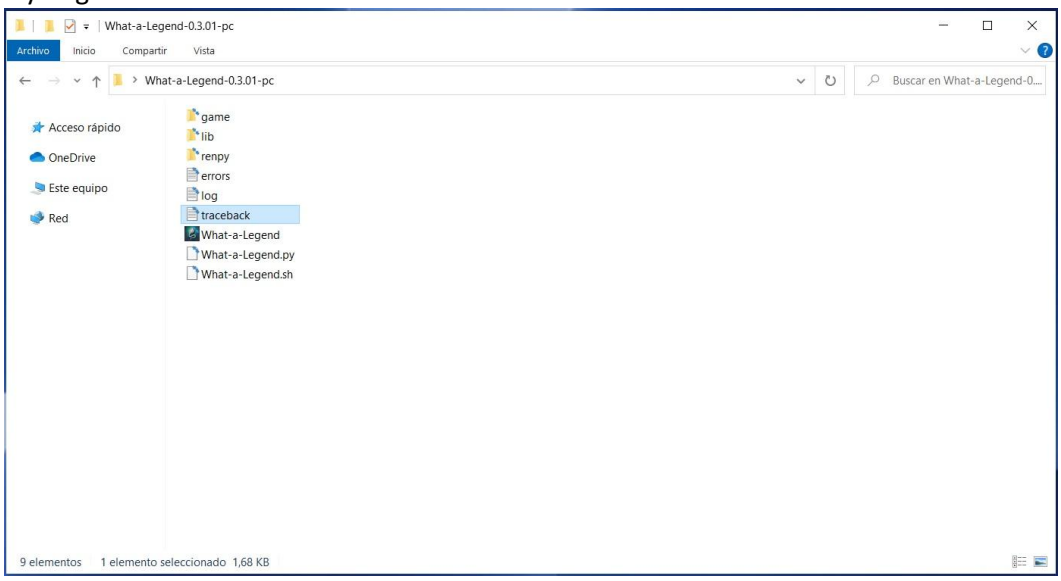
Sometimes, the error is not a critical one in the sense that it allows us to launch the game and generate the translation files, but it does generate an exception while playing the game. Generally, these failures are due to a misspelled `{tag}` or a problem with the game's variables. That's when, during our gameplay, this gray screen with all those lines suddenly pops up:



Here we are given the option to try to ignore the exception and resume playing (with a high chance of getting further warning screens and even game's crashes). We can also rollback to an earlier point to save our game before trying to fix what's wrong.

In addition, we can copy this message with a BBCode format (that will allow us to insert it in online forums) or to copy it as Markdown (which then we could attach to a Discord message) just in case we want to ask for help. However, these are almost always minor errors that, even when they are not originated by our work, are relatively easy to fix. To fix them, we always must pay attention to the message's first line, since it indicates the location of the coding line that caused the problem. In this case, in the "game_intro_label.rpy" translation script (if you look closer, it's said that it's inside the "game/tl/Spanish" folder, and that's why I know this is the translation script and not the original one), line 7, there is an unclosed tag: a tag was opened with the command `{b}` to **bold** a word, but it wasn't properly closed. And in the last line just before the info section (where we can find the player's OS, the software's version and the date and time of the exception), the exception's cause is displayed again, although without any reference to the script where it occurred. Just by looking at these two lines, we will be able to locate and fix the mistake.

If we now take a look at game's root folder, we would see that a "traceback.txt" file has been generated too. This file contains the same information we saw in the in-game exception screen. Both this "traceback.txt" file and the "errors.txt" file we saw before are regenerated every time an exception occurs, deleting its pre-existing content to show only the most recent issue. In all cases, it's just a matter of locating and fixing the error, saving the scripts and trying again what we were doing when the exception appeared, hoping that was the only bug. Good luck with that.



Lastly, if we see a more complex message, or a message referring to scripts within the "renpy" folder, it's likely due to us accidentally deleting some apparently insignificant symbol (such as a % or a slash \ /), or adding some other (such a spacebar stroke before or following a %), thus breaking Ren'Py's pre-coded functions. This kind of exceptions, quite common when using machine translators (as they tend to replace that kind of symbols because reasons), are so much difficult to fix, as we'd need a high comprehension level of Ren'Py's internal procedures in order to know where to look for the error's actual source in the translation scripts. So buy yourselves some serious amount of patience and get ready to dig in forums, or just forget about it and restart the whole process from a clean and bug-free game's version.

|Index|

3.2.- Lines with too much text

Generally speaking, words and sentences written in English are usually shorter than in most languages. This can cause some translations to exceed the limits of the textbox that we see on screen, or to visually obstruct other elements in the user's interface. Game becomes hard or unpleasant to read and sometimes even the functionality of some buttons is altered. There's a triple and quite simple solution to this.

For starters, we can always try to rewrite our line to say the same with fewer characters. . If the result does not convince us, we can take advantage of the huge freedom that Ren'Py gives us to write the translated text from dialog block. Because, although Ren'Py SDK only offers us one line to translate the original string, we can split it into two or more lines, as long as we respect the Ren'Py's Commandments regarding quotes and tabs. Look at the example:

```
# game/script.rpy:10
translate Spanish label_start_XXXXXXXX:
    # mc "Hello. My name is John."
    mc "Hola."
    mc "Me llamo John."
```

Although the text is displayed in just one string in the original script, in the Spanish translation it will be split into two messages, with no further consequence. Actually, we can introduce any variation we can think of, like adding functions, modifying variables, etc., which will only be applied when playing in our language. That's because, although the translation function is intended to replace one text string with another text string in a different language, what it actually does is to replace a whole line of code with what we write in the translation script, so nothing can really prevent us from replacing that line of original code (which happens to be just some text said by a character) with a whole different block of code.

The third option, a bit more complex, is to change font size and/or textbox width, so that more characters can be displayed in each line. But, as this means modifying the so-called "styles" (that is, the aesthetic configuration for game's texts and menus), I'll explain that in the next paragraph.

|Index|

3.3.- Fonts that don't allow special characters: changing styles

Sometimes, the game we are translating uses a font which doesn't contain some special characters such as accented vowels or any other symbols that are quite specific to some languages. Therefore, when running the translated version, the words will be displayed on screen without those letters or with an odd symbol instead. To solve this issue without modifying the original game's code, we could just press the 'A' key while playing: provided that the game has been created with a relatively recent version of Ren'Py, an accessibility menu will pop up, and we can choose DejaVuSans as the default font. Obviously, this changes the game's original aesthetics; besides, if we can 'fix' that issue, we might as well do it, don't you think?

If we are somewhat skilled and have a lot of spare time, we could go all-in and edit the original font with a font-editing software (such as TypeLight) to design the missing characters and symbols. That way we

wouldn't change that much the game's aesthetics originally imagined by its developer and we'd only need to remember adding the new .ttf file to our patch so it replaces the older one in the "game" folder. But the most sensible thing to do is to start learning about what a "style" is in Ren'Py, and how to change it.

As you could probably tell by its name, in Ren'Py, a style is an array of several elements (text color and font size, textbox's position and design, and so on) that defines the aesthetics looks of the game's UI. As we saw above, sometimes we need to modify some of those original elements in order to get our translation to be correctly displayed, so we could look for them and edit them directly in the "game/gui.rpy" or game/options.rpy scripts, as that's where the elements that determine how the game's text is displayed are usually defined.

That was the method I explained in this guide's first version, but we must take into account that, if we change something in the original's "gui.rpy" or "options.rpy" files, those changes will be applied to all available languages, breaking the original aesthetics created by the game's dev. Besides, if our intention is to share our translation with others, we should include those edited scripts in our patch, forcing players to replace the original game's scripts; if they ever want to delete the translation, they will never get the original version back.

So, if we want to make some changes that will effect the game's appearance when our translation is enabled, while respecting the game's original settings, we can use this translation function, writing it in any script (preferably, in one of those we have in our translation subfolder):

```
translate Spanish python:
```

It's not indented and `Spanish` is, as always, the word we wrote when generating translation scripts with Ren'Py SDK. I'd advice to write it in a specific script, along the preferences screen that includes the switch languages option, as we'll see later. This function opens a python coding block in which we can redefine absolutely every element we want to customize for our translation, while respecting the game's original layout if we are playing in some other language. Applying it to the font issue, we could write something like:

```
translate Spanish python:
    gui.text_font = "myfont.ttf"
    gui.name_text_font = "myfont.ttf"
    gui.interface_text_font = "myfont.ttf"
    gui.button_text_font = "myfont.ttf"
    gui.choice_button_text_font = "myfont.ttf"
```

Where `"myfont.ttf"` is the font we want to use in our language. Each line effects a certain type of texts (from top to bottom: dialogs, character's names, UI texts, buttons and choice menu texts), so we don't actually need to change them all, only those we do need to change. And remember to include the myfont.ttf file in your patch so it's stored in your language's "game/tl" subfolder.

We can also use this function to change other UI elements, such as font sizes or textbox positioning and width, which would allow us to solve the issue that prevented our translation to be correctly displayed when our text was too long:

```
translate Spanish python:
    gui.text_size = ...
    gui.name_text_size = ...
    gui.interface_text_size = ...
    gui.label_text_size = ...
    gui.notify_text_size = ...
    gui.textbox_height = ...
    gui.dialogue_width = ...
    gui.dialogue_xpos = ...
```

And lots more. We just have to replace those three dots with a numeric value: for the different `text_size` we'll add a lower value than the one originally assigned to this variable in the "gui.rpy" script, thus making the font smaller; if we dial up `gui.textbox_height` the textbox will be higher, thus maybe allowing us to include some more lines to it; if we increase the digits for `gui.dialogue_width`, we'll increase the textbox width, so lines can be a little longer; and by changing the number assigned to `gui.dialogue_xpos` we could move horizontally the point from where the text starts (a lower number means the text will start more on the left side of the screen). These variables and some others will allow us to adjust our translation's texts, and usually are a valid solution for most games. Although, as it's been said, we'll be changing the game's original aesthetics, so we might be creating other visualization issues such as superposed texts. We'll have to try and experiment until we get a satisfying result.

All the options mentioned so far refer to elements that are broadly applied to the whole game. But it might happen that the game's developer had defined specific settings for specific characters, screens or game's sections (like a dream sequence, for instance, where texts will be displayed differently), so they won't be using those general definitions. That's a proper "style", defined with the `style` command. In those cases, the biggest issue for translators is to actually find where that style is defined within the original scripts, because we need to know its name; then the process is basically the same as before: using the `translate Spanish python:` function, we'll redefine the specific element from that style that we want to change. For instance, if we want to replace the font used by a style named "dream", as well as its fontsize, we should write this in our script:

```
translate Spanish python:
    style.dream.font = "myfont.ttf"
    style.dream.size = 22
```

As I said, the key here is to find where that style's elements are originally defined, in order to know the style's name and their original values so we can modify them as we please or need.

|Index|

3.4.- Translating pictures

Sometimes the game's developers decide that an important part of the action has to be based on something displayed on a SMS message, for example, or any other visual element (a computer screenshot, a newspaper headline, a poster on the wall or whatever you can imagine). Although Ren'Py offers several coding solutions to write these texts in the scripts and, therefore, make its translation easier, devs usually find more convenient to just create an image with the text included on it.

If we are lucky enough that, while the image is being displayed, there is also some dialogue going on, in our translation we can just add the text to the character's words, as if they were reading it aloud:

```
# game/script.rpy:12
translate Spanish label_start_XXXXXXXX:
    # mc "Look. He sent me a message."
    mc " Look. He sent me a message. He said that (whatever is said in the pic)"
```

Alternatively, we could "dub" the image's content thanks to a solution we already explained when we needed to fit the translated string within the textbox: we can split that string in two or more translation lines, so we can write in them the translation of the image's message, maybe in *italics* or in some other way that lets players know what's going on there.

```
# game/script.rpy:12
translate Spanish label_start_XXXXXXXX:
    # mc "Look. He sent me a message."
    mc " Look. He sent me a message."
    "{i}(And here we translate the message that is displayed on the image.){/i}"
```

If, sadly, we don't have any dialog lines associated with the image, we can create a blank one in the original script. To do this, the first step is to open the **original script**, find the line of code in which we think Ren'Py is ordered to show the image (we can use nearby text strings to find it) and then, respecting the indentation, we would insert a new line below, with a blank string (writing only quotation marks). Then we just need to regenerate the translation scripts and include the pic's translated text as a translation for the blank string. Of course, we'd need to include in our patch the original script we have just edited, or, alternatively, replace the edited labels in the way we'll explain later on.

However, sometimes this option is simply not viable or the results don't look aesthetically good. In that case, the only solution is editing the original image (or, to be more precise, creating a new one) with some picture edition software like Photoshop or GIMP (or even MS Paint, if it's a very simple editing). First, we would go to the original script to find image's name (generally whatever appears after the `show` or `scene` commands). Then we would look for it inside the "game/images" folder and edit it to write its text in our language. In an ideal world, we could kindly ask game's developer to provide us the base image, without any text on it, to make this editing work easier. Good luck with that.

Once the edition is finished, we shouldn't replace the original image in that "game/images" folder; instead, we need to save a copy with our changes, with identical name and image format, in an identical path but inside our translation folder (in my case, inside "game/tl/Spanish"). So, if the original image is stored as "`game/images/ch1/sms00.jpeg`", we **MUST** save the translated image as "`game/tl/Spanish/images/ch1/sms00.jpeg`". That way, when we are playing the game's translated version and the image named "sms00.jpeg" has to be displayed, Ren'Py will first look for it in the "images" subfolder of the translation folder, and only when the pic cannot be found there, it will display the one in the original "game/images" folder. And, logically, if we are playing in the game's original language, the original image will be displayed.

|Index|

3.5.- Homonyms: different translations for identical words

This issue already came up when talking about the two translation functions used by Ren'Py. Strings that are translated directly, without encryption code, only admit one translation, even though they might appear several times in the original scripts: in fact, if we try to translate literally the same `old` string twice, Ren'Py won't allow us to run the game, and in the error message we'll be told that there's a duplicated translation, so we'll need to delete both the `old` and the `new` lines from one of those pairings.

But, as I said in that example, sometimes we find identical strings that need different translations, like the English word "Right" which, among other things, could mean "Correct" or just a side or a direction opposite to "left". The solution is to edit the original script so those identical strings stop being identical, without effecting the game in the original language. And to do that we only need to use the # symbol.

As we stated long ago, everything to the right of a # symbol will not appear on screen and is discarded by Ren'Py in all its processes... unless we put it inside a tag. Tags are commands embedded in the strings with brackets, **`{tag}`**, and are usually used to change font's size and color, to highlight the sentence in **bold** or *italics*, etc. Ren'Py reads these tags as part of the string that contains them and executes what they say, but its literal content is not displayed on screen. So, if we include in a tag some text after a # symbol (which Ren'Py uses to know that it doesn't need to do a thing with what's written there), we'll have a string which is now different than the untagged one (so it will be extracted as a different string by Ren'Py SDK) but both of them will be displayed exactly in the same way, as our new tag isn't requesting Ren'Py to do any action.

Therefore, we'll find in the original scripts the string that we want to translate differently and we'll incorporate a tag in it in which we will write something that will allow us to identify it later, such as the translation we want to apply. Thus, when reading and extracting strings, the text "`Right`" and the text "`Right{#Direction}`" are no longer identical for Ren'Py, although both will be displayed as "Right" on

screen. Now we only need to regenerate the translation scripts (unless we choose to manually include in them, with the `old` command, the new 'tagged' string) and then assign this tagged string a different translation with the `new` command. Eventually, we should have something like this (note that these pairings wouldn't necessarily appear one after the other, and maybe not even in the same translation script):

```
translate Spanish strings:

    old "Right"
    new "Correcto"

    old "Right{#Direction}"
    new "Derecha"
```

Remember that, in this case, writing this in the translation script is not enough: **we also have to edit the original .rpy script to embed the tag** into the string we want to translate differently so that, when playing the translated version, Ren'Py will look for the tagged string's translation. And obviously, we'll have to remember to include these edited strings in our patch or to replace the labels involved as we'll explain later.

|Index|

3.6.- Translating text variables (I): interpolations

When we talked about Ren'Py SDK's shortcomings when it comes to extract all translatable strings from games, we already said that variables' text values aren't automatically detected by the extraction function. And, no matter how we achieve to include them in our translations scripts, the biggest problem is that, most of times, their translation won't be correctly displayed on screen while playing.

For instance, let's assume that, somewhere in the original scripts, the following variable is defined:

```
default fruit = "apple"
```

And, later on, maybe in other script, there's a line like this next one that is telling Ren'Py to modify that varaible's value to store this other one:

```
$ fruit = "orange"
```

Sometimes we need to do a thorough script research, but let's suppose that we've made a good helping job for Ren'Py SDK and we have found the original value assigned to the variable "fruit" and also all the possible values that can be assigned to that variable throughout the whole game. And, either because we included those values in a `_( )` symbol before generating the translation files, or because we write them down manually, the fact is that we already have their translations ready thanks to the `old` and `new` commands, and it looks something like this:

```
translate Spanish strings:

    old "apple"
    new "manzana"

    old "orange"
    new "naranja"
```

Well, when a variable's current value is displayed on screen, in the code it's actually embedded in a string, and the technical term for that is "interpolation". We can identify an interpolation because, in the string, we'll see the name of a variable in brackets, `[variable]`. Sometimes the string will consist only in that object, and sometimes it will be embedded in the middle of a sentence. And the problem for translations will show up when that variable is like the one in the example, "fruit"; that is, a variable whose possible values are words or sentences that should be translated.

For instance, in a translation script, we can stumble upon something like this:

```
translate Spanish start_XXXXXXXX:
    # mc "I don't like this [fruit]."
    mc "No me gusta esta [fruit]."
```

In this case, `[fruit]` is the interpolated variable: depending on what we have done during our playthrough, the sentence will be different and the character will say what fruit he dislikes. By coding it this way, game's dev doesn't need to write the same sentence as many times as possible "fruit" values exist. But the problem is that, if we leave the interpolation just like that, in the same way it's written in the original line, we'll be asking Ren'Py to display the current value of the variable `[fruit]`, and that value is stored in the original language, so it will be displayed on that language even though we had all those values perfectly translated.

Why is that? Think that, even when we play with our translation enabled, Ren'Py is still running the original scripts, and it only 'jumps' to the translation script whenever there's a translatable string. All the coding lines that change variables' values are in the original scripts, written in the original language. In our example, while playing the game, Ren'Py has been instructed to store "apple" as the default value for the variable "fruit", then it might have been requested to replace that value with the new value "orange", and now this string is asking Ren'Py to display the value currently stored for the variable "fruit". But Ren'Py can't possibly know by itself if that value is a number, a translatable word or just a random combination of symbols, so it just displays what is stored in the way it was literally stored.

So, if we want Ren'Py to know that this value is actually another string that should be translated before being displayed, **we must add this `!t` appendix** within the object with the variable's name. Like this:

```
translate Spanish start_XXXXXXXX:
    # mc "I don't like this [fruit]."
    mc "No me gusta esta [fruit!t]."
```

Now, when executing the translation function for this dialog string, Ren'Py will find that, in addition, the translated string is requesting it to look for a valid translation for the interpolated variable's current value, because that value is actually a translatable string too. It would be ideal that the original string were already written with the interpolation `[fruit!t]`, because it wouldn't affect the original game and it would be harder for translators to miss that appendix in our translation string: if we generate the translation scripts without empty strings, we'd already have that interpolation written in our string, and if we chose to generate them with empty strings, at least we'll be seeing the right interpolation in the upper line, so we could just copy that. Sadly, most game developers don't know about these Ren'Py singularities but, as you can see, it's not that difficult to fix: as long as the `!t` appendix is present in our translation string, Ren'Py will display the translated value, so we don't need to edit the original scripts at all.

|Index|

3.6.1.- Interpolating game character's names: Now, let's see a specific type of interpolations: game character's names. Maybe because it's just convenient when writing the dialog block's strings, or maybe because it's mandatory in case they are giving players the option to name certain characters, the fact is that, sometimes, game developers replace the character's full names with the definition of those "character" variables. You'll recognize this because the interpolated variable you'll see embedded in the string is the same you can see at the start of that character's dialog lines.

For instance, in the game "Deliverance" there are some characters that don't have a first name, but a generic noun, which is a translatable word. This is how one of them is defined:

```
8  define li = Character("Lieutenant", ctc="ctc_default",ctc_pause="ctc_default") #fowler
```

In this case, every time the character `"li"` speaks, we see his name (Lieutenant, which is just a rank, not an actual name) above the textbox. As we already saw, if we want that word to be displayed in our language,

we need to extract that string from the character's definition, so it appears in some translation script with the `old` and `new` commands. The singularity here is that, as this `[li]` variable was defined as a "character" one, whenever it's interpolated in a string, we don't need to add the **`!t`** appendix to that object: Ren'Py will automatically know that its value is a text and will look for an appropriate translation if we're playing with the translation enabled.

```
177  # game/script.rpy:350
178  translate Spanish interrogation_3864c01a:
179
180      # co "[li], you have a visitor. Mayor wants to see you."
181      co "[li], tiene una visita. El alcalde quiere verle."
```

In this example, even though the interpolation in our translation string is just `[li]`, when playing the game in our language the word displayed on screen will be the translated one. We **don't** need to write `[li!t]`.

Sometimes, though, things are a bit messier. A common case is a supporting character who is first identified with a generic noun because players still don't know who he/she really is, but later on we can see that character's real name. It's possible that this character has been defined this way:

```
define p = Character("[pname]")
```

So, "p" character's readable name is actually a string entirely made of another variable interpolation. And this second variable called "pname" may have those values scattered throughout the original scripts:

```
default pname = _("Unknown Man")
$ pname = "Mike"
```

In that case, we should look for that "pname" variable's definition and extract its default value in order to translate it with the `old` and `new` commands; but, also, we'd need to extract the string from the "p" character's definition too, adding to that string's translation the **`!t`** appendix. Something like this:

```
translate Spanish strings:
    old "[pname]"
    new "[pname!t]"
```

That way, when "`pname`" has a generic noun as its current value (and we have found and translated that value), the variable called "`p`" will be translated both in the little textbox that displays the character's name above the dialog textbox, and also in every `[p]` interpolation that might appear in the dialog block. So we won't need to write `[p!t]` in our translation strings because "p" has been defined as a character-type variable, and Ren'Py will automatically display its translated value when needed; a translation that, in this case, it's another variable's translation. But beware: if the interpolated variable in a dialog block's string is `[pname]` and not `[p]`, then we would need to write `[pname!t]` in our translated string, because "`pname`" is an ordinary variable as those we saw in the previous section, not a character-type variable as "`p`".

Defining characters with an interpolated variable is a very common feature in Ren'Py games, as it's a part of the coding method that developers use to allow players to name their characters. Generally speaking, that naming variable would be defined thanks to a `renpy.input` function located somewhere near the start of the game, which allows players to type in their desired name (a name that, obviously, translators can't possibly know, and doesn't need to be translated). But we should pay attention to that variable's default value, because it will be displayed in case players leave that name blank, or it might be displayed on screen if the character 'speaks' before the naming option appears: if that default value is a generic noun, we'll need to translate it using the above mentioned method (that is, we'd need to translate that generic value and then translate the interpolation in the character's definition string too).

|Index|

3.6.2.- Grammar concord: In some languages, certain words can experiment some changes due to

grammatical gender or number, verbal tense or mode… and those changes, which are essential if we want our translation to be fully intelligible, may be forced by some variable's values that don't cause these effects in the game's original language.

Back to the generic example used when talking about interpolations, what if, besides "apple" and "orange", which are feminine nouns in Spanish, our `[fruit]` variable has "peach" as another possible value, that value being a masculine noun in our language? Let's remember the string in where `[fruit]` was interpolated:

```
translate Spanish start_XXXXXXXX:
    # mc "I don't like this [fruit]."
    mc " I don't like this [fruit!t]."
```

In Spanish, the word "this" is a demonstrative that will change its form depending on the variable [fruit] gender. So, while in English that gender is actually inexistent, in Spanish we'll need a string for feminine values of "fruit", and another string for its masculine values. Fortunately, as we already saw when fixing other issue, a translation script is actually a bunch of functions that replace a coding line with another coding line, so nothing can prevent us from writing a whole code block instead of just one translation string. Therefore, we can perfectly use the "fruit" variable to add an alternative string to our translation. In order to do that, we need a very basic knowledge about how to code in Ren'Py, but it's really easy.

```
translate Spanish start_XXXXXXXX:
    # mc "I don't like this [fruit]."
    if fruit == "peach":
        mc "No me gusta este [fruit!t]."
    else:
        mc "No me gusta esta [fruit!t]."
```

First, and observing Ren'Py's First Commandment about indentations, we start writing our function at the same spot where an ordinary translation string begins (that is, four spacebar strokes to the right of the left margin, as the `translate` function ends with a colon `:` which is the symbol used by Ren'Py to know where a new coding sub-block starts). Then we write a conditional function: with an `if` command we tell Ren'Py that, if the variable named `fruit` has `"peach"` as its current value, the first string must be displayed (we'll write that string with another indentation level); and, with the `else` command we are telling Ren'Py that, if "fruit" current value is a different one, the second string must be displayed instead.

If we want to be sure that everything works properly, it's essential to respect how the variable and its reference value are literally written in the original scripts; we also need to write everything as we saw above, with the double = symbol, the quotes wrapping the variable's value that Ren'Py will search, and the colon : at the end of those coding lines, as well as the extra indentations for the translated strings. If we don't write everything right, the game either won't launch or will throw an exception when it reaches this string.

Obviously, we must analyze in a case by case basis which value (or values) we should take as reference for the `if` command. For instance, if we know the variable "fruit" can also take "melon" as possible value ("melon" being a masculine noun too, in Spanish), we should write our function this way:

```
translate Spanish start_XXXXXXXX:
    # mc "I don't like this [fruit]."
    if fruit == "peach" or fruit == "melon":
        mc "No me gusta este [fruit!t]."
    else:
        mc "No me gusta esta [fruit!t]."
```

In this case, thanks to the `or` particle, firs string (with the masculine translation for "this") will be displayed every time "fruit" has one of those masculine nouns as its value. If we have more values from one gender than the other, the reasonable thing to do is using the shorter list in the `if` expression.

I've written this example with an interpolated variable so we could easily see the reason for the concord change, but obviously a string doesn't need to include an interpolated variable to require an alternative version due to some variable's possible values. Maybe the most usual case is those games that allow players to choose their main character's gender. In English, this option generates very few variations during dialog strings, but in Spanish it can easily become a coding nightmare. A simple sentence as "You're my best friend.", which in English can be addressed to both men and women, in Spanish will force us to include a conditional function in our translation script, so first we'd need to find the variable that stores the character's gender and then its possible values in order to use one of them as filter in the `if` expression. It can be a really tedious task, as we'll need to write a conditional function for every possible string that may be affected, taking an extra care in each of those lines to not generate any bug, but the results are worth it.

|Index|

3.7.- Translating text variables (II): string concatenations

There are even more complex variables than those we saw in the previous section, as their text value is generated by the addition of several elements. Those strings will need an extra touch to be correctly translated and displayed. These are the so called string concatenations, which usually are a result of a python function like this one.

```
$ score = "Your score is " + strCalc + "out of " + strTotal + "."
```

In this example, the value for the variable `"score"` is a sentence in which the results of two functions (`strCalc` y `strTotal`) are interpolated. Those functions are defined somewhere else in the scripts; in some other spot there'll also be a string in which this `[score]` variable is interpolated so the player's score is displayed on screen. We have already learnt to extract that string and translate that interpolation as `[score!t]`, but it won't work just like that. So, how can we get that text to be displayed in our language?

First, we need to realize that, when we interpolate `[score!t]`, the string that Ren'Py will look for in order to display its translation is the whole literal sentence. So, if player's current score is 1 out of 10, Ren'Py will search the translation scripts looking for an `old` string that literally says `"Your score is 1 out of 10"`. But that whole sentence won't be fully written anywhere in the original scripts and Ren'Py SDK won't extract it. So the problem is that we should write in our translation scripts a full sentence like that one for every possible score combination, and that's a highly inefficient task – and, depending on the game's design, simply an impossible one.

The solution to this is to edit the original scripts, wrapping each text element from that string with a parentheses preceded by a double underscore. In our example:

```
$ score = __("Your score is ") + strCalc + __("out of ") + strTotal + "."
```

The `__( )` symbol will allow Ren'Py SDK to extract those 'partial' strings, but it will also make Ren'Py to store that variable's value in the language the game is being played when that coding line is read. Without that double underscore, the variable's value will be always stored in the scripts' original language.

Now there's a little problem. We are playing with our translation enabled, and Ren'Py has stored the "score" variable's value in our language. But if we now switch back to the game's language, Ren'Py will display the original strings and, when it comes across the `[score]` interpolation in an original string, that text will be displayed in our language instead. To avoid that, we should edit that interpolation in the original script, so it's written `[score!t]`. This will "undo" the translation, translating the translated stored value back into the original language. Obviously, in our translation scripts we will always interpolate it as `[score!t]`.

In addition, if `strCalc` y `strTotal` values were also text strings, we should find those functions and wrap their possible results with this parentheses plus double underscore symbol. And, obviously, we'll need to add to our patch all the original scripts we have edited to include the `__( )` symbol or, alternatively,

make use of the label replace function or the `init` command (depending on whether the variable is defined inside a label or in a separated python block). So yeah, a bit more complex, but still doable.

|Index|

3.8.- When everything else fails: "massive translation weapons"

Despite all what we've seen till this point, it's possible that some parts of the game's texts are still being displayed in the game's original language. Keep calm, there are still some solutions we can try as a last resort option. Leaving aside the most obvious one, which is editing the original scripts to rewrite them in our language (basically because, if you do that, then why I'm writing this guide?), we have at least three "massive translation weapons" more we can use to see every game's text in our language; although, before applying them, we should be at least aware of what their real impact is and what 'casualties' and undesired effects they might cause. Please use them under your own responsibility.

|Index|

3.8.1.- Double underscore: We've just been introduced to the first weapon, which is nothing more and nothing less than the `__( )` symbol. We can use it as a last resort solution whenever we can't find the string where a text variable is interpolated (this can happen in complex menus or inventory screens, or when we don't have the time nor the willpower needed to thoroughly investigate the scripts looking for those interpolations). As I'm saying, it is a desperate solution that might generate some internal logic problem, if by chance that variable's text value is used in one of those expressions that determine the game's routes or scenes to be displayed. In those cases, in order to avoid bugs and narrative inconsistencies, we should wrap that text value with the `__( )` symbol in those expressions too.

Looking again at our example with the variable "fruit", we could have just added the `__( )` symbol when we found its possible values in the scripts:

```
default fruit = __("apple")

$ fruit = __("orange")
```

That way, whenever the variable `[fruit]` is interpolated in the scripts (both in the original and in the translation ones), the word that will be displayed on screen will be the translated term for "apple" and "orange", respectively. This is useful, as I said, if we can't find a particular interpolation so we can't translate that string as `[fruit!t]`.

But, as we also have seen before, the first side effect is that those values will also be displayed in our language if we switch back to the game's original language once those values are set, because Ren'Py would have stored the translated value and the original script still shows the original interpolation `[fruit]` Remember that we couldn't find that interpolation in the first place (that's the reason why we resorted to the double underscore) so we won't be able to fix that.

The second possible side effect will arise if that variable is also used somewhere to decide which dialog is going to be displayed or which route are we going to follow from that point, in an expression like this one:

```
if fruit == "apple"
    mc "I like red apples."
elif fruit == "orange"
    mc "I like orange juice."
```

If we don't edit that code, the game will break at this point because the variable's stored value would be the translated term for "apple" and "orange", but not literally "apple" and "orange", so Ren'Py wouldn't identify any valid value in that expression and wouldn't be able to choose which string to display. It might not be a critical error, and depending on how the game is actually coded this could go totally unnoticed or maybe cause just a little incoherence in dialogs, but, anyway, that's not how the game's developer designed their

game. To solve this issue, we should use the `__( )` symbol in the logical expression in the original script:

```
if fruit == __("apple")
    mc "I like red apples."
elif fruit == __("orange")
    mc "I like orange juice."
```

There's no need to mention that, had we translated everything in a proper way, this edition would be unnecessary, because the logical expression would be checked with the original language's values, as usual. But if we are forced to do it, we'll need to add the edited original script to our patch, unless we opt for the label replacing function we'll see later on.

|Index|

3.8.2.- The "replace" function: The second massive translation weapon is probably the most complex of all of them, as it requires us to write a python coding block and being extremely careful with how we write the elements we want to replace, but it's also the most powerful and indiscriminate tool. It consists on a python function that will automatically replace at the very last second what it was going to be displayed on screen, no matter neither where it's coded nor if it's a complete string or just a part of a string. It looks like this:

```
init python:
    def replace_text(s):
        s = s.replace('text 1', 'texto 1')
        s = s.replace('text 2', 'texto 2')

        return s
    config.replace_text = replace_text
```

This function can be written in any script, no matter whether it's an original or a translation one, so the most logical choice would be to write it in a translation script. What this function does is to carry out a replacement of literal characters, substituting the first quoted expression (what I called `'text 1'`, `'text 2'`, etc.) by its pairing (`'texto 1'`, `'texto 2'`, etc.). This replacement is executed at the very last moment before the text is actually displayed on screen, after Ren'Py has searched and found (or not) a valid translation for that string and even applied the possible `{tags}` on it: whenever that exact combination of characters is found in the message to be displayed, the player will actually read what we wrote as replacement. This is useful for those strings that are not automatically extracted by Ren'Py SDK and we aren't able to find in the scripts (nor to write them literally in an `old` line that matches the original, maybe because they have some tag in them we don't know, thus preventing the translation to be displayed). This replace function only acts over the selected text, not over the whole string, so it bypasses that limitation.

What's the problem, then? Well, it could happen that a certain combination of characters in the original language might be also present in a perfectly translated string, and this function will replace them every time, so that sentence that was perfectly translated will stop making sense in our language. It's not a frequent issue, but it can happen: let's imagine that we weren't able to translate an inventory screen where the English word "pint" is still being displayed when playing in our language. We can use this function and replace 'pint' with its translation ('pinta', in Spanish), but then every time that the exact combination of characters 'pint' is found in the Spanish strings, they will also be replaced by 'pinta'. So a word like '**pint**or' (the Spanish word for 'painter') will now be displayed as '**pinta**or', leaving the impression that there's a typo despite we translated it right. As a result, I only recommend using this function with extreme care and in a sensible way (for instance, only for long strings that we are certain they're impossible to be found literally written that way in our language), and always combining it with the next weapon, in order to avoid the replacement to take place also when playing in the game's original language.

|Index|

3.8.3.- Language detection: Lastly, we still have to talk about a third weapon that can be used as a supporting line in order to limit the side effects of the second one, and also to let ourselves taking some

unorthodox but effective actions over the original scripts. In a way, it's similar to what happens when we include a function to determine which string is going to be displayed to solve a concord issue; the main difference being that now we would be working on the original scripts and not in the translation ones. We will include a logical expression to check in which language the game is currently being played, so if the current language is the game's original language, Ren'Py will keep running the original code, and if the player has a translation enabled, the game will run an alternative piece of code.

To achieve this, we must understand that Ren'Py stores the language choice in an internal variable named `preferences.language`. And, as any other variable, we can use it in a conditional `if` function to order Ren'Py to display a specific text or to execute a block of code whenever a certain requirement is met.

Applying it to the replace function we've just seen in the previous paragraph, we'll limit that characters' substitution to the times when we are playing with the translation enabled and the text to be replaced is going to be displayed in our language, so players playing in the game's original language won't be affected and won't read anything in our language. The replace function will now look like this:

```
init python:
    if preferences.language = "Spanish":
        def replace_text(s):
            s = s.replace('text 1', 'texto 1')
            s = s.replace('text 2', 'texto 2')

            return s
        config.replace_text = replace_text
```

As you can see, we must include the conditional statement at the beginning of the block, which forces us to increase the indentation of the next coding lines. Now those lines will only be executed when playing in our language. But, obviously, this won't fix the accidental substitutions we mentioned earlier.

The language variable also allows us to incorporate customized coding elements to the original scripts. If we use it in a conditional function before any of those original elements, Ren'Py will check in what language we are playing the game and, according to that, it will execute the original code or some other alternative piece of code we wrote (an alternative piece of code which strings we could write in our own language, by the way, as they will only be displayed when playing with the translation enabled). For instance, we could solve a hypothetical issue with some concatenation by writing this in the original script where it's located:

```
if preferences.language = "Spanish"
    $ score = "Tu puntuación es de " + strCalc + "sobre " + strTotal + "."
else:
    $ score = "Your score is " + strCalc + "out of " + strTotal + "."
```

It's advised to write the function in this order, so that the `else` block executes the original code when playing in the game's original language but also in any other different language than the one we use as check and that might be already (or not yet) implemented. In this example, we wouldn't need to add the `!t` appendix to the `[score]` interpolation's translation, as its value will be already stored in our language. And, obviously, if we are going to create a patch to share our translation with others, we must include in it the original scripts we have edited. This is not a fancy solution as it actually shows we weren't able to do things the right way, but at least it's effective, and it never hurts to know we have this resource available.

|Index|

4.- THE PATCH

Once we managed to successfully complete our translation, we could just enjoy it privately. But, if we want to share our master piece with the rest of the world, we need to create a patch made only of the strictly

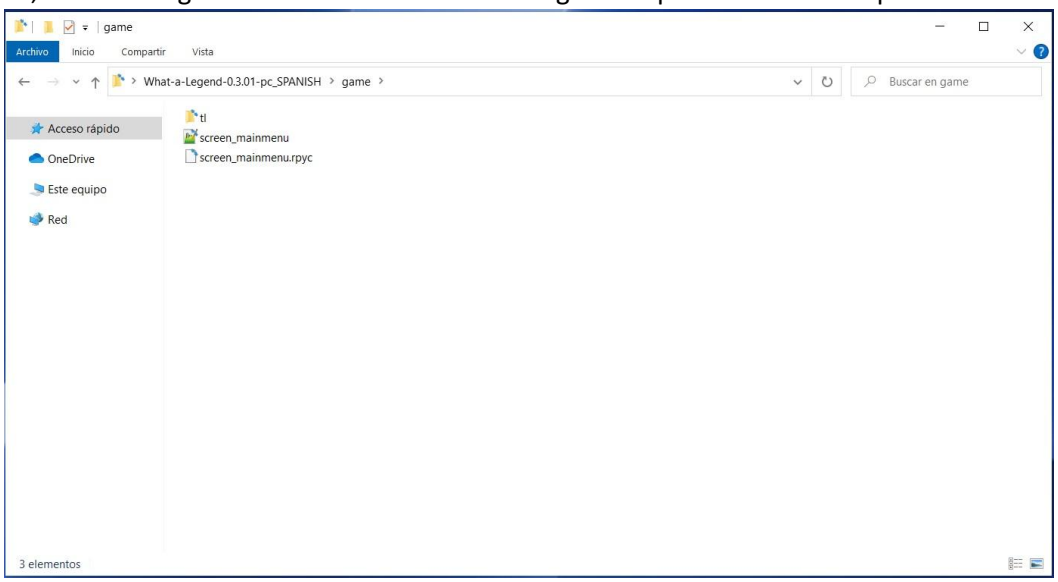
needed files that will make our translation work for anyone who has already installed the original game.

|Index|

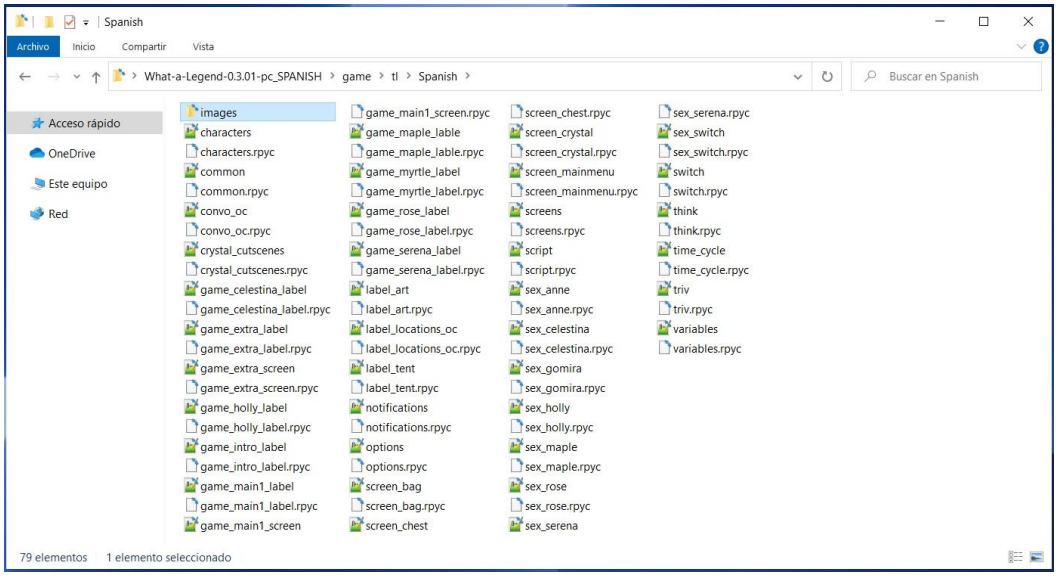
4.1.- Basic patch

Knowing that, if we want a patch to work, it should respect and emulate the game's original folder's structure, the easiest way to create a translation patch is to just create a new folder in our PC, naming it like the original game plus an identifier that will allow people know this a translation patch; inside this new folder, we'll create a new "game" subfolder in which we'll include the original .rpy scripts we have edited (along with their respective .rpyc files), and another new folder named "tl"; this "tl" folder will contain our language subfolder with our translation scripts (and also the pics and fonts required to solve the issues from section 3.3, when needed). Users will only need to paste our patch's "game" folder in their copy's root folder (where the game .exe is) and accept to replace or overwrite all files, if that warning shows up.

For instance, this is the "game" folder from a "What a Legend!" Spanish translation patch:



In this game there's no need to edit any other original script in order to get the translation to be perfectly displayed on screen, so the only file in the "game" folder is the "screen_mainmenu.rpy" script (with its .rpyc), which was edited to include the switch languages option. Those files will replace the original ones currently present in player's "game" folder. Obviously, there's also a "tl" folder that includes the "Spanish" subfolder I had in my computer with the full translation done. This subfolder looks like this:



There you can see all the translation scripts plus yet another subfolder named "images", which stores every pic I had to translate. And that subfolder is structured exactly as the original "game/images" folder, as it was

explained in section 3.4.

As it has been said, in order to let players enjoy our work in a proper way, we should include in our patch's "game" folder every original script we edited to help translate homonyms, plus the "gui.rpy" or "screens.rpy" files we edited to change the default language or game's fonts, and also any script in which we had to include a __( ) symbol in order to translate some especially tricky text variables or resorted to an emergency solution. You can also throw in there any original script you only edited with the _( ) symbol to allow Ren'Py SDK extract every translatable string, but only if you want to offer a helping hand to other language's translators, as players don't really need those.

This had always been my patching method; however, and due to the technical inconveniences and suspicions that replacing original files may arise between players (and developers), since some months ago I've been using another method, maybe a little bit more complex, that I'll explain in next section. If, despite everything I've just said, you choose to create a 'classic patch', I'd recommend you to ALWAYS include all the .rpyc files from the original .rpy scripts you've edited. Even though they have been also modified due to your editions in the .rpy scripts, as long as you didn't delete them since you first downloaded the original game (or since you extracted them with UnRen to generate the translations scripts), the .rpyc scripts in your PC will keep the AST that was generated during the very first compilation made by game's developer, and the same goes for player's own .rpyc files if they have never deleted them. So, after installing your patch, players shouldn't have any problems to load saves they stored while playing with the unpatched game.

What would happen if you only include the edited .rpy scripts in your patch? Well, it depends. If the game is released with its .rpyc files stored in plain sight within the "game" folder, your modified .rpy scripts will replace the original ones in player's PC and, next time the game is launched, Ren'Py will update player's .rpyc files to compile in them the changes you made to the .rpy scripts. This process shouldn't break anything… unless the player has deleted at some point the original .rpyc scripts, but then it wouldn't be your fault if something goes wrong.

The really big problem may arise when the game's original scripts are stored within a .rpa file: if you only include in your patch the .rpy scripts, Ren'Py will create their matching .rpyc files in player's computer, but those files won't be generated according to the same AST used as reference by the ones you extracted from the original game in order to generate your translation scripts. And, if you don't include in your patch absolutely every .rpy script, when playing the game Ren'Py will have to run some .rpyc that are still inside player's .rpa file and were compiled with the original AST. So it will have to run some "internal" .rpyc compiled with an AST, and some "external" .rpyc compiled with a different AST. In those cases, besides the old saves issue I've mentioned earlier, it is quite possible that the function that executes the translation of the dialog block (the one that uses encryption codes) won't detect the existing translation, due to that difference in the .rpyc's ASTs. As a result, the in-game menus and choices will be displayed translated (because they are translated using a different function that looks for their literal content) but dialogs from those edited scripts will be displayed in the game's original language because Ren'Py won't be able to link the dialog block with its translation due to that AST mismatch.

It's an odd error but it can happen, especially with games originally compiled with a Ren'Py version older than the one you used to generate the translation scripts. But, as I said, you can avoid all this hassle just by adding the .rpyc scripts to your patch: even if they have been updated with the changes you made to the .rpy scripts, they'll keep intact the AST structure from the original game. Better safe than sorry.

|Index|

4.2.- Advanced patch

As we have just seen, when we create a "basic" patch we're forcing players to replace their game's original files with our edited scripts. And beyond all those problems with old saves and even translation malfunctions, when we replace an original file with some external file we are opening the door to several

situations not easily reversible. The most extreme case would be that we might have accidentally generated a bug we are not even aware of; the most common case, though, is that we might have modified some aesthetical elements that will be carried over to other languages that didn't need them. This is something to take into consideration because we never know where our patch could end up, and it's also possible that some players might try out our translation but then decide to keep playing in the game's original language.

In this examples, and some others, many players would prefer to be able to just delete our patch and get the game's original version back without any changes, and we shouldn't deny them this option by forcing them to keep some aesthetical modifications they don't need and, most probably, they don't like nor want either. Besides, if we create a translation patch that doesn't affect the original files it may be easier to convince the game's developer to add our work to their official build, which is always a reason of pride and sometimes even a source of legal monetary income (or, at least, a more honest way to earn some bucks than trying to sell our translation directly to players without game dev's consent).

Fortunately, both Ren'Py and the Python coding language offer several solutions to store in our "game/tl" language subfolder all these modifications needed for the game to be correctly displayed in our language, allowing those who want to delete the translation and get the game's original version back to do so just by deleting that subfolder. Be aware, though, that this requires a bit more of work and a minimum comprehension of how Ren'Py game's coding works, but nothing too complicated at this point.

|Index|

4.2.1.- The zzz.rpy script and the init command: First step we'll take will be creating a new file where we will be including all those modifications and editions we made to the original scripts so our translation could run smoothly in our computer. To do that, we'll open a new blank file in our text editor that we'll name as we prefer (obviously, it's advised to use an easily identifiable name, and it's mandatory that it doesn't match any of the already existing scripts in the translation subfolder), adding the extension .rpy as part of that file name. For instance, "zzz.rpy" or something like that, and we'll save that file in our "game/tl/…" language subfolder as a plain text file without any specific format (Ren'Py will recognize it as a new script thanks to the .rpy extension we wrote).

Following a somewhat logical order, in this script we should first include the only element that should be always present in every translation: the switch languages option. So we will copy the whole original preferences screen's code, which is a bit longer piece of code than what you can see in the next image *(note: this block assumes the game is using Ren'Py's default preferences screen; if it's using a custom one, you'd obviously have to copy that customized screen)*.

```
## Preferences screen ##########################################################
##
## The preferences screen allows the player to configure the game to better suit
## themselves.
##
## https://www.renpy.org/doc/html/screen_special.html#preferences

screen preferences():

    tag menu

    use game_menu(_("Preferences"), scroll="viewport"):

        vbox:

            hbox:
                box_wrap True

                if renpy.variant("pc") or renpy.variant("web"):

                    vbox:
                        style_prefix "radio"
                        label _("Display")
                        textbutton _("Window") action Preference("display", "window")
                        textbutton _("Fullscreen") action Preference("display", "fullscreen")

                vbox:
                    style_prefix "radio"
                    label _("Rollback Side")
                    textbutton _("Disable") action Preference("rollback side", "disable")
                    textbutton _("Left") action Preference("rollback side", "left")
                    textbutton _("Right") action Preference("rollback side", "right")

                vbox:
                    style_prefix "check"
                    label _("Skip")
                    textbutton _("Unseen Text") action Preference("skip", "toggle")
                    textbutton _("After Choices") action Preference("after choices", "toggle")
                    textbutton _("Transitions") action InvertSelected(Preference("transitions", "toggle"))

                ## Additional vboxes of type "radio_pref" or "check_pref" can be
                ## added here, to add additional creator-defined preferences.

            null height (4 * gui.pref_spacing)
```

The idea is to copy all this indented block, starting by `screen preferences():` till the next unindented command, which is usually a `style` (we won't select that command). Everything between those coding lines is the preferences screen, and that's exactly what we have to move entirely to our new script, in order to override this original screen. Once we have this code planted out in our script, we'll start to edit it.

Firstly, if we hadn't established the switch languages option before, we must add it now. As we saw, the code we need to write in a default preferences screen is this one (remember that it should be written at the same indentation level than "Rollback Side" and "Skip" options, and don't forget to use the spacebar instead of the tab key):

```
vbox:
    style_prefix "radio"
    label _("Language")
    textbutton _("English") action Language(None)
    textbutton _("Español") action Language("Spanish")
```

Then we need to let Ren'Py know that this is the preferences screen that must be used, instead of the original one that players have stored in their "screens.rpy" original script. To do so, we'll use the **`init`** command, that tells Ren'Py which functions must be executed when launching the game (before anything is actually displayed on player's screen) and in what order. This `init` command can be assigned a numeric value (from `init -999` to `init 999`) that will determine the order in which those commands will be load in Ren'Py's internal memory. And this loading order is important because, if there are two commands with an identical name, at the end of the launching process Ren'Py will only keep stored in its internal memory the last loaded one; that is, the element with a higher init number overwrites any identical element it might exist. If those identical elements have the same init number too, then the one that prevails is the one included in the last script in alphabetical order.

Under ordinary circumstances, the original preferences screen's code doesn't explicitly include this init command, but we know it's internally loaded at `init 0` (so after the elements with a negative init and before any element with a positive init). As a result, if we assign our new script's preferences screen an `init` value bigger than 0, Ren'Py will load it in its internal memory after loading the original one, which will then be overwritten, so when playing, every time the `screen preferences` is called, the one to be displayed will be ours. Long story short, we should write this just above `screen preferences():`

```
init offset = 1
```

This command is telling Ren'Py that the next definition in the script must be loaded in the init order we have written there, so in init 1, in this case.

```
### SPANISH SCREENS ###

# Preferences Screen

init offset = 1
screen preferences():

    tag menu

    use game_menu(_("Options"), scroll="viewport"):

        vbox:

            hbox:
                box_wrap True

                if renpy.variant("pc"):

                    vbox:
                        style_prefix "radio"
                        label _("Display")
                        textbutton _("Window") action Preference("display", "window")
                        textbutton _("Fullscreen") action Preference("display", "fullscreen")

                vbox:
                    style_prefix "radio"
                    label _("Rollback Side")
                    textbutton _("Disable") action Preference("rollback side", "disable")
                    textbutton _("Left") action Preference("rollback side", "left")
                    textbutton _("Right") action Preference("rollback side", "right")

                vbox:
                    style_prefix "check"
                    label _("Skip")
                    textbutton _("Unseen Text") action Preference("skip", "toggle")
                    textbutton _("After Choices") action Preference("after choices", "toggle")
                    textbutton _("Transitions") action InvertSelected(Preference("transitions", "toggle"))

                vbox:
                    style_prefix "radio"
                    label _("Language")
                    textbutton _("English") action Language (None)
                    textbutton _("Español") action Language ("Spanish")
```

Alternatively, we could also edit that screen's first line in this way (the other method is preferred, though):

```
init 1 screen preferences():
```

In those examples I've used an init 1, but you can use any positive number (or a higher number that the one the game's developer had assigned to their original screen, because some devs do assign a specific `init` to the preferences screen). Once it's done, our preferences screen will be the one displayed in all languages.

Now, something important: if we generated our translation scripts before including the switch languages option in the original preferences screen, it's very likely that we should need to translate now the string "Language" (unless it's literally written in some other script). We can see that string wrapped with the `_( )` symbol in our new preferences screen, so we could think that, if we regenerate now the translation scripts, it will be extracted by Ren'Py SDK, but we would be wrong: no translatable strings will be extracted from scripts within the "game/tl/…" languages subfolders. The quickest (and probably best) solution would be resorting to the "manual extraction" I explained in section 2.3, as we only need to translate the string "Language", but not the language's names (names that, as I said, should be kept in their original language).

So, below our preferences screen, we'll write this piece of code, with the `translate` function without any indentation (but only, I insist, if we hadn't translate this string before; otherwise, this translation will be already present in the "screens.rpy" script within our translation subfolder and the game won't launch due to this duplicate translation):

```
translate Spanish strings:
    old "Language"
    new "Idioma"
```

Next, we should also write in this zzz.rpy script the functions used to launch the game straight in our language. As a quick reminder, if we want the game to be launched in our language the first time it's ever launched on a given device, we should define the variable `config.default_language`:

```
define config.default_language = "Spanish"
```

"Spanish" being my language, remember to write yours. If the game already includes this variable for some other language, we should override it by writing this coding line under an `init` block that will be executed after the original one. So, like this:

```
init offset = 1
define config.default_language = "Spanish"
```

And if we want the game to be always launched in our language (even if players choose to change to another language while playing), we should write this line and forget about the above mentioned one, because that `default_language` variable will be relegated due to Ren'Py's priority order:

```
define config.language = "Spanish"
```

Usually, this variable is never defined for any other language, but if we found ourselves in a situation where a game already includes this variable, we just have to do the same as before: creating a new and higher `init` block (by default, the original variable will be loaded at init 0).

And so on. In this script we would keep writing all those functions and variables we had to add to modify the user interface's visuals (that is, what it was explained when talking about changing fonts ans styles), and also any other desperate solutions such as the replace function, because Ren'Py will find them and apply them without issues despite of them being "hidden" in our translation subfolder.

|Index|

4.2.2.- Label overrides: But, what happens with all those script editions that altered the content in some game's label, such as the changes we made to let us differently translate identical words or to get something translated thanks to the double underscore? Our game's private copy will have those editions included in the original scripts, which has allowed us to make a full translation, but if we don't want to overwrite player's original files, we must resort to a label override so our patch can keep displaying a perfect translation over a game's intact version.

To do so, we must copy the edited labels **entirely** and paste them in this zzz.rpy script (or in another new script we can specifically create for this purpose, in case we want to keep separated the editions that belongs to game's settings from this other kind of editions that involve the game's displayable texts). Then we will rename this label we have just pasted in, preferably by adding a prefix or a suffix that let us tell it apart from the original label (actually, the only thing that matters is that the new name shouldn't contain accented vowels nor any other 'non-English' symbol and doesn't match any existing label). For instance, assuming the label we had to edit was the "start" label, we could rename it in our zzz.rpy script as `label start_es`. Lastly, we need to write this function, without indentation:

```
define config.label_overrides = {"start":"start_es"}
```

It's very important to write everything correctly: the name of each label is quoted, the colon orders the second label to be displayed instead of the first one, and both labels must be wrapped with a pair of brackets. If we need to replace more labels, we will copy-paste and rename them in the zzz.rpy script, and then we will add them to this very same function, separating each pairing with a comma. Like this:

```
define config.label_overrides = {"start":"start_es", "example1":"example1_es"}
```

That said, once we have given Ren'Py the order to run our label instead of the original one, the translation we have for the original label's dialogs won't work with the new one, because the label's name is part of the identifying code for that line in the translation function. In addition, if we regenerate now the translation scripts, this new label won't be extracted by Ren'Py SDK as it's located in a script within a "game/tl" subfolder.

However, the solution it's somewhat easy. We need to open the translation script where the original label's translation is, and then do a search and replace for the label's name, so all the encryption codes that contained the original label's name will now contain the new one's name. In the prior example where we replaced the "start" label, wherever we had something like `translate Spanish start_XXXXXXXX:` we should now have `translate Spanish start_es_XXXXXXXX:`

We'll do this with all the effected lines. We just need to be extremely careful to change only the label's lines and leave the others intact. Fortunately, changing the label's name doesn't change the alphanumeric MD5 encryption code that appears next to it (the XXXXXXXX in this example), because we haven't modified the original lines' content since they were translated, so after this quick change our translation will be perfectly valid again. In that regard, remember that this label's strings that are translated with the literal translation function (like menu choices we might have edited to solve some homonyms issue, or text values of a concatenated string) don't depend on the label's name in which they are located, so their old translations will always be valid regardless that label renaming.

|Index|

That was all. I really hope you found this guide useful.
Feel free to contact me with your doubts, suggestions or corrections (especially with the latter).
Take care!