Python, extensions and pickle in Ren'Py

Presentation

IMPORTANT	All code examples are available in example games in the directory `examples` in the project's repository. You will need a working Ren'Py SDK install to run them.

What is Ren'Py?

How does it work?

Ren'py uses text files with the extension `.rpy` to implement a game's content.

Those files may at any depth in the `game/` directory in the project. When Ren'Py reads them, they are compiled into .rpyc files

NOTE	Code in `examples/01-simple-game`

```
# game/game.rpy
default gender = None
label start:
  "Hello"
  menu:
    "Are you a boy or a girl?"
    "A boy":
      $ gender = 0b0
    "A girl":
      $ gender = 0b1
    "Both":
      $ gender = 0b10
    "Neither":
      $ gender = 3
```

```
init python:
  def square(x):
    return x**2
```

A simple Jar

Now, let's build something simple. Let's say we want to be able to count the number of instances of an item and, instead of storing this information as a number in a dedicated variable like any normal human being, we want to use OOP to represent the data, so that it is "future-proof*TM*".

We want to create the object class in an `init python` block. This a scope loaded at the start of the game where we can use python code freely.

This class will take an initial amount as a contructor parameter and will provide two helper functions: `add` and `take`.

> **NOTE** Code in `examples/02-extension`

```
# game/awesome_extension.rpy
init python:
  class Jar(object):
    def __init__(self, initial_count = 0):
      self.count = initial_count

    def add(self, added = 1):
      assert added >= 0
      self.count += added

    def take(self, taken = 1):
      assert taken >= 0
      if self.count < taken:
        raise Exception("Underfull jar")
      self.count -= taken
```

Once we have our class, we can start to use it in our game... Or create a game to test our extension if we don't already have one.

The following code is a full, mostly-working [1] , Ren'Py game.

```
# game/game.rpy
default fruit_jar = Jar()
label start:
  menu:
    "You have [fruit_jar.count] items"
    "Add one":
      $ fruit_jar.add(1)
    "Take one":
      $ fruit_jar.take(1)
    "Quit":
      $ renpy.quit()
  jump start
```

That's it, we're done. This is the end. We can pack those two files and share them with any potential user, and they will have both our "extension" and an example to build from should they need to. Ain't that great?

And you might feel at peace for a while, but that will all change when the user reports attack.

The load order

And indeed, user reports will come. The first ones should have errors that look like this:

> **NOTE** Code in `examples/03-extension-load-order`

```
Full traceback:
  File "game/game.rpy", line 1, in script
    define fruit_jar = Jar()
  File "/games/renpy/ast.py", line 2222, in execute
    self.set()
  File "/games/renpy/ast.py", line 2236, in set
    value = renpy.python.py_eval_bytecode(self.code.bytecode)
  File "/games/renpy/python.py", line 1202, in py_eval_bytecode
    return eval(bytecode, globals, locals)
  File "game/game.rpy", line 1, in <module>
    define fruit_jar = Jar()
NameError: name 'Jar' is not defined
```

What happened?

When trying to debug with the users you might notice their directory layout looking like this:

```
-game
  |-game.rpy
  -jar_extension
    -awesome_extension.rpy
```

This should have given you a hint of where the issue lies: the extension is loaded too late by Ren'Py and its content is thus not available when the code using it is executed.

Ren'Py uses the file's full path (e.g.: `game/jar_extension/awesome_extension`) to determine the order in which files should be loaded. By convention, files with names starting with `00` are reserved for Ren'Py's internal use [2] , thus if we want to ensure that our file is loaded as soon as possible, we should name all directories (and the file itself) with a `01` at the start to limit the problem.

Now, this is a good thing to do and a good habit to take, but it also feels very fragile: if I download an "extension" for my game, I want to be able to place it where I want. Luckily, we can enforce the load order differently in Ren'Py with init levels.

Init levels are a Ren'Py concept that allow developpers to enforce a specific load order for code

sections. They range from -1500 to 1500 though the 501 first and last init levels are reserved for Ren'Py [3] . By default, all user-defined code in a game is loaded at init level 0, so we simply need to provide a lower init level at the start of our `python` block to prevent this kind of issue:

NOTE Code in `examples/04-extension-with-load-order`

```
# game/01_jar_extension/01_awesome_extension.rpy
init -999 python:
  class Jar(object):
    def __init__(self, initial_count = 0):
      self.count = initial_count

    # [...]
```

Updates and pickle

We now have a small package of working code, which is nice. But we suddenly realize that an infinite jar is yet to be invented and that we should probably add the possibility to raise an error if a jar is overfilled [4] .

NOTE Code in `examples/05-extension-updated`

To do so, let's simply add a `maximum` size attribute to the `Jar` and check its value when adding items:

```
# game/01_jar_extension/01_awesome_extension.rpy
init -999 python:
  class Jar(object):
    def __init__(self, initial_count = 0, maximum = None):
      self.count = initial_count
      self.maximum = maximum

    def add(self, added = 1):
      assert added >= 0
      self.count += added
      if self.maximum is not None and self.count > self.maximum:
        raise Exception("Overfull Jar")

    # [...]
```

Now, if a user simply instanciates our `Jar` with a `maximum`, an error will be thrown if they try to add too many items. After running a few games with this new `Jar`, we can release a new version... and get submerged by user reports. Again.

```
Full traceback:
  File "game/game.rpy", line 6, in script
    $ fruit_jar.add(1)
```

```
  File "/games/renpy/renpy/ast.py", line 823, in execute
    renpy.python.py_exec_bytecode(self.code.bytecode, self.hide, store=self.store)
  File "/games/renpy/renpy/python.py", line 1178, in py_exec_bytecode
    exec(bytecode, globals, locals)
  File "game/game.rpy", line 6, in <module>
    $ fruit_jar.add(1)
  File "game/01_jar_extension/01_awesome_extension.rpy", line 10, in add
    if self.maximum is not None and self.count > self.maximum:
AttributeError: 'Jar' object has no attribute 'maximum'
```

Saves in a pickle

What happened? When we tested, everything was fine!

Users claim that they can't resume previous games as all saves appear to be corrupted. Starting new games works fine though, but this is clearly not an acceptable situation for our users.

The secret lies in the way Ren'Py saves and reloads games. Internally, Ren'Py uses the Python library `pickle` to pack and unpack a game's state. Pickle's behavior makes it skip the `init` method when re-creating objects; it then injects the saved attributes in the created object. In our case, this means that we're using `Jar` with pre-update attributes (only `count`) with code that expects post-update attributes (both `count` and `maximum`).

When developping a Ren'Py game, we may use the `after_load` label [5] to resolve this kind of migration issues, however this is something that should be used by the game developers, not by extension developers.

To resolve migration issues in extension classes, we should use `__setstate__`, which - if defined - is called by pickle to initialize object instances. But that's not all! We also need to handle rollback, which is handled by Ren'Py itself. When rolling back, Ren'Py tries to call the function `_rollback` to initialize the object's state - which it may do with a previous-version save's data, so we also need to overload it.

NOTE Code in `examples/06-extension-updated-with-pickle`

Note that the state provided to `_rollback` may be reused when rolling forward, so we should not modify the incoming dict directly (though it should not be an issue functionnally **if** we can guaranty that the update is idempotent). Considering both of those use-cases, it is better if we can implement an `_update` method and call it from the setter methods:

```
# game/01_jar_extension/01_awesome_extension.rpy
init -999 python:
  class Jar(object):
    def _update(self, state):
      if 'maximum' not in state:
        state['maximum'] = None

    def __setstate__(self, state):
```

```
      self._update(state)
      self.__dict__ = state

    def _rollback(self, compressed):
      self.__dict__.clear()
      self.__dict__.update(compressed)
      self._update(self.__dict__)
    # [...]
```

This is good, but not very future-proof. To be able to easily update the class, we'd like to have a version in our state and leverage it to know what changes we should make to have an up-to-date object. Like this:

NOTE Code in `examples/07-extension-future-updates`

```
# game/01_jar_extension/01_awesome_extension.rpy
init -999 python:
  class Jar(object):
    def _update(self, state):
      version = state.pop('_version', 0)
      if version == Jar.VERSION:
        return
      if version <= 0:
        if 'maximum' not in state:
          state['maximum'] = None
      # Handle next versions updates here
    # [...]
```

To be able to use this version number, we need to add it to the saved state. To do so, we need to implement the function used by pickle (`__getstate__`) as well as the one used by Ren'Py (`_clean`) to extract the object's state [6] . Note that the `_compress` method is implemented here for the sake of completeness and is called by Ren'Py to "compress" the value returned by `_clean`.

```
# game/01_jar_extension/01_awesome_extension.rpy
init -999 python:
  class Jar(object):
    VERSION = 1
    def __getstate__(self):
      return self._clean()

    def _clean(self):
      data = self.__dict__.copy()
      data['_version'] = Jar.VERSION
      return data

    def _compress(self, clean):
      return clean
```

Note on slotted classes

Slotted classes provide better attribute access performance and memory usage for python classes at the cost of declaring their valid attributes as a `__slots__` class attribute [7] . Ren'Py does not quite support those as handling attribute changes with slots is less trivial than with dicts, is a use-case that is not relevant to most users, and that games that use Ren'Py should not be performance-sensitive as Ren'Py is not designed for high-frequency play [8] . You may still create slotted class, but should only use them for global game state that is neither part of rollback, nor part of saves.

NOTE	Code in `examples/08-slot-fail`

The following shows how to declare our `Jar` class with slots and the error that appears in Ren'Py if we do so:

```
# game/01_jar_extension/01_awesome_extension.rpy
init -999 python:
  class Jar(object):
    __slots__ = ('count', 'maximum')
```

```
Full traceback:
  File "game/01_jar_extension/01_awesome_extension.rpy", line 1, in script
    init -999 python:
  File "/games/renpy/renpy/ast.py", line 823, in execute
    renpy.python.py_exec_bytecode(self.code.bytecode, self.hide, store=self.store)
  File "/games/renpy/renpy/python.py", line 1178, in py_exec_bytecode
    exec(bytecode, globals, locals)
  File "game/01_jar_extension/01_awesome_extension.rpy", line 2, in <module>
    class Jar(object):
  File "/games/renpy/renpy/revertable.py", line 495, in __init_subclass__
    raise TypeError("Classes with __slots__ do not support rollback. "
TypeError: Classes with __slots__ do not support rollback. To create a class with
slots, inherit from python_object instead.
```

Ren'Py native extensions - Who needs doc ayway?

Ren'Py supports a "native" kind of extensions, which is undocumented, forgotten, and requires some work to get to work correctly. It has the notable advantage and drawback of running very early (before Ren'Py even starts parsing `.rpy` files), which is great to make code modifications to the way Ren'Py processes data. They are also packaged as zip files, which makes users much less likely to mess with the extension's content. The following is a primer on what to do to get a basic extension running.

IMPORTANT	All showcased extension code in the examples is in the `extensions/` directory. This works thanks to the `extension-loader` extension added as `game/01-`

> `extension-loader.rpe`, however extensions would normally need to be zipped with the `.rpe` extension and added directly under the `game/` directory to work.

To build an extension's archive from the command-line, create a zip from its content and make sure that the `autorun.py` is at the archive's root:

```
zip my_extension.rpe autorun.py
```

To migrate our extension to use Ren'Py extensions, we should start by moving our class' declaration to the `autorun.py` file. Because the autorun is executed in a different context, the class will not be available in renpy if we don't explicitly add it to the Ren'Py store, which we should immediately do:

NOTE Code in `examples/09-rpe-basics`

```
# extensions/jar_extension/autorun.py
import renpy

class Jar(object):
  VERSION = 1
  def __init__(self, initial_count = 0, maximum = None):
    self.count = initial_count
    self.maximum = maximum

  # [...]
renpy.store.Jar = Jar
```

[1] We also need a `yesno_prompt` screen to avoid throwing an error when quitting the game via the window's button.

[2] Warned by the VSCode extension, this is not enforced by Ren'Py itself

[3] https://www.renpy.org/doc/html/python.html#init-python-statement

[4] Don't actually do this for a game, prefer non-intrusive warnings and discreet handling of such errors when possible so as to not stop the game catastrophically in simple cases

[5] https://www.renpy.org/doc/html/label.html#special-labels

[6] We use a class attribute to store the version in our example as it is what Ren'Py does in its own codebase, however we could also use an object-local attribute, which would remove the need to overwride the default `__getstate__` and `_clean` implementations - See the `Object` class in `renpy/object.py` in Ren'Py's codebase for an example.

[7] See Python's documentation for details https://docs.python.org/3/reference/datamodel.html#slots

[8] For performance-sensitive VN games, take a look at Godot with the Dialogue Manager plugin