# SOWRL – World State & NPC Scheduling Extension for Ren’Py

## Overview

**SOWRL** (State Of World Ren’Py Library) is a modular backend extension for the Ren’Py engine that adds game-world simulation features to visual novels. It introduces a persistent **world state**, in-game **time and calendar**, character **schedules**, and basic **NPC logic** on top of Ren’Py’s narrative framework. Using SOWRL, developers can create visual novels with **open-world or life simulation elements**, such as day/night cycles, NPC daily routines, timed events, and dynamic game variables that persist and change over time. This framework addresses Ren’Py’s lack of built-in world simulation by providing structured modules to manage time progression, scheduled events, and entity states.

**What problems does SOWRL solve?** In a standard Ren’Py game, game state is often limited to story variables and linear script flow. SOWRL solves common challenges when implementing sandbox or schedule-based gameplay in Ren’Py: tracking the current time and date, scheduling events to occur at specific times or after delays, managing NPC locations/availability, and maintaining global state (like counters, flags, or stats) in an organized way. It is particularly useful for games that require:

* **In-game calendars or day cycles:** e.g. morning/afternoon/evening cycles, weekdays/weekends, or calendar events.
* **Character schedules and routines:** NPCs that move between locations or change behavior based on time (school by day, home by night, etc.).
* **Timed events and delays:** triggering story events after a certain in-game time has passed or at a particular time of day.
* **World state tracking:** a structured store for game-wide variables (e.g. player stats, quest flags, resources) that need to be accessed and modified by various parts of the game.

**Use Cases:** SOWRL is ideal for visual novels or simulation games that go beyond linear storytelling. For example, a **school life sim** where each day is tracked and characters follow class schedules; a **mystery game** where events happen on certain days regardless of the player’s location; or a **dating sim/RPG** with a day-night schedule and recurring events. By using SOWRL, developers can focus on game design and story logic while relying on the extension to handle the timing and world state mechanics.

## Installation

Installing SOWRL in your Ren’Py project is straightforward. Since SOWRL is distributed as a Python-based extension for Ren’Py, you typically just need to add its files to your project and perform a one-time initialization:

1. **Get the SOWRL code:** Download or clone the SOWRL repository from GitHub. You should obtain a folder (or set of .rpy files) containing the modules (e.g. time.rpy, event.rpy, person.rpy, puppeteer.rpy, utils.rpy, etc.).
2. **Copy files into your game directory:** Place the SOWRL .rpy files into your Ren’Py project’s game/ directory (you can keep them in a subfolder if desired, Ren’Py will include them as long as they are under game/). For example, you might have game/SOWRL/time.rpy, game/SOWRL/person.rpy, etc.
3. **Ensure correct load order:** The SOWRL files should be loaded early in the game. Ren’Py loads .rpy files in alphabetical order by default. If necessary, rename or add the SOWRL files in a way that they initialize before your game’s script that uses them. (For instance, you can prefix filenames with a number or letter to control ordering, or use the init blocks with appropriate priority.)
4. **Initialize the world object:** SOWRL organizes its functionality under a central **game world object** (often simply called game). In SOWRL’s default setup, an OkiMaster (master controller) class instance is created to hold all controllers. If the extension does not automatically create it, you should create this object. Typically, this can be done in an init python block at the start of your script:

init python:  
 import SOWRL # (if SOWRL is a package – otherwise its classes are already in context)  
 default game = OkiMaster() # Create the central game object (if not already created by SOWRL)  
 # Optionally create shortcuts for controllers:  
 default date = game.date\_controller  
 default persons = game.person\_controller  
 default puppeteer = game.puppeteer\_controller  
 default machine = game.machine\_controller

**Note:** If SOWRL is implemented as a Ren’Py module (.rpym), simply placing it in the directory might auto-initialize the game object for you. Check the SOWRL README for any specific initialization call. By default, the main world object is named game – you can use a different name, but then you must update references in SOWRL’s scripts or your code to match.

1. **Verify setup:** Launch the game. If installed correctly, no import errors should occur and you should have the SOWRL controllers available (either via the game object or the shortcut names you defined). It’s good to test in the Ren’Py console that, for example, game.date\_controller.get() returns the current date/time dictionary to ensure everything is in place.

There are no special dependencies beyond Ren’Py itself – SOWRL is pure Python and uses Ren’Py’s standard capabilities. Ensure you are using a Ren’Py version compatible with SOWRL (the extension was developed with Ren’Py 7.x/8.x in mind). No additional Python packages are needed.

## Module Overview

SOWRL is divided into logical modules, each handling a specific aspect of the world simulation. These modules work together but can be understood in isolation for clarity. Here is an overview of each module, what it manages, and how they interact:

### Time Module (time)

The **time module** manages the in-game **date and time**. It keeps track of the current time-of-day, day, week, and other calendar information, and provides methods to manipulate time. Internally, it uses a DateController (often accessed as game.date\_controller or a date shortcut) to maintain the game clock.

**What it manages:** The Time module stores the current hour, minute, day, etc., and enforces the progression of time in the game world. It can be thought of as the “game clock.” It knows things like the current hour/minute, which day of the week it is, and what the current day part is (morning, afternoon, etc., if defined).

**Key responsibilities:**

* **Game Time State:** Keeps a data structure for time with fields such as hour, minute, day of month, day of week, week number, month, etc.. For example, it tracks 'hour', 'minute', 'weekday' (number of weekday), 'weekname' (name of weekday like "Monday"), and so on.
* **Time Progression:** Provides functions to change the current time: you can set a specific time, or add hours/minutes/days to move time forward.
* **Interval Checking:** Utility to check if a given time falls within a time interval (useful for schedule logic).
* **Timed Event Scheduling:** This module also introduces **timer events** via methods to schedule code to run after some time or at a specific time (using do\_in and do\_at). These are essentially world events not tied to a character – often used for delayed actions or global timed events.

**Interactions:** The Time module works closely with the Event module (timers) – in fact, they are part of the same system of managing time-based triggers. It also feeds into the Puppeteer module: the current time (specifically the current “time of day” and day) is used by Puppeteer to determine which NPC events are active. When time is advanced via the Time module, scheduled events (do\_in/do\_at) are checked and may fire, and the Puppeteer may also update NPC schedules accordingly. In summary, the Time module is the backbone that drives scheduled checks – without time progressing, scheduled NPC events (or any events) wouldn’t trigger.

### Event Module (event)

The **event module** handles **timed events and scheduling of code execution**. In SOWRL, this primarily refers to the system of timers (which could be called “world events” or timed tasks) that can trigger arbitrary code after a delay or at a specific date/time. Essentially, if the Time module is the clock, the Event module is an alarm system where you can set alarms to go off and run some code.

**What it manages:** Scheduling and managing **timer tasks**. You can create a timer event that says “after X in-game minutes, do Y” or “at 6:00 PM on a given day, do Z”. Each such event is tracked in an internal list of timers with details like when it should execute and what code to run.

**Key responsibilities:**

* **One-time Delayed Actions:** Using do\_in(), you schedule a one-time action after a given amount of time elapses. For example, you might trigger a reminder or a game event 30 minutes later in the game world.
* **One-time Scheduled Actions:** Using do\_at(), you schedule an action to occur at an exact future time (optionally a certain number of days from now). For example, trigger an event exactly at 7:00 AM the next day.
* **Timer Management:** It stores all active timer events in a structure (each timer has properties like a unique name, the code to execute, whether it has executed, the target day/time, etc.). The Event module ensures that once the game’s current time reaches or passes the scheduled time, the code is executed (and not executed again if it’s one-time).
* **Event Execution:** When a timer event’s condition is met, the module executes the associated Python code. This is often used to call Ren’Py labels or set variables. (Under the hood, the do parameter of these functions is a string of Python code that gets executed at the right time.)

**Interactions:** The Event module relies on the Time module for current time – it checks timers against the current in-game time. It likely integrates with the main game loop or screens: for example, after advancing time or when entering a new scene, the game might check if any timers should fire. The Event module is mostly self-contained (just tracking scheduled tasks), but those tasks can affect anything in the game (since they run arbitrary code). This means Event timers might update the world state (variables), trigger story events (via Ren’Py label jumps), or even interact with the Puppeteer (e.g., enabling/disabling an NPC schedule at a certain time). In practice, the developer will use Event module functions to script autonomous happenings in the game world.

### Person Module (person)

The **person module** manages **NPC data and logic**. It provides a structured way to create and manage characters (NPCs) as dictionary objects, storing their attributes like name, current location, whether they’re active, etc. Think of it as a simple NPC database. Rather than using many separate Ren’Py variables for each character’s info, you use the Person module’s API to add characters and set/get their properties.

**What it manages:** A list (or dict) of **person records** (NPCs). Each NPC is represented by a dictionary of properties describing that character’s state in the world. The PersonController (e.g. game.person\_controller or persons) is responsible for creating, updating, and retrieving these NPC records.

**Key responsibilities:**

* **NPC Creation:** You can create a new character/NPC by calling the add function with basic info like name, a unique identifier, and possibly initial location. This generates a dictionary entry for the character and stores it internally.
* **NPC Properties:** Each character entry includes properties such as:
* **Identity:** name (first name), surname, and a unique code name cfname (character’s ID).
* **Appearance/Assets:** arts (the base directory or identifier for the character’s images), and image (the current image path if one is set/displayed).
* **World Position:** location (the location key or list of locations where the character currently is), and coordinates x, y for positioning the character on screen or on a map.
* **State Flags:** isActive (whether the character is presently in the world or should be ignored), status (current status/state name, especially if using schedules), and use\_daypart (whether the character’s image should auto-variation based on time of day).
* **Interaction Hooks:** action – a label name or code to run when this NPC is interacted with (clicked). By default this might be set to 'person\_' + cfname (so if cfname is "john", action becomes "person\_john" as a suggested Ren’Py label to define for interaction).
* **Visual Effects (optional):** There are fields like scale, hue, brightness, saturation that define how the character image might change on hover (these are optional enhancements for UI hover effects).
* **Accessing/Updating NPCs:** The module lets you retrieve a character’s dictionary by their cfname or query by location, etc., and modify properties via provided methods (so you don’t have to manually manipulate the dictionary). For example, you can get an NPC’s current location or set their isActive to False when removing them from the world.
* **Removal:** You can remove a character entry entirely if needed (e.g., if a character leaves the story, to clean up).

**Interactions:** The Person module works with the Puppeteer module to handle **NPC logic**. Person module is about static data of NPCs; Puppeteer uses that data to control where NPCs are or what they’re doing at a given time. For example, when a scheduled event triggers an NPC to move to a new location, Puppeteer will call the Person module to update that NPC’s location and status. Likewise, when displaying a scene, you might use Person module queries (like “get all persons at location X”) to know which characters to show on screen. The Person module doesn’t inherently move characters on its own – it’s the data layer. It relies on Puppeteer or your game logic to change a character’s data over time or in response to events.

### Puppeteer Module (puppeteer)

The **puppeteer module** is the heart of SOWRL’s **schedule and NPC event system**. If we imagine the game world as a stage, the Puppeteer is the stage director – it “pulls the strings” to put characters (and other entities) in the right place at the right time according to a predefined schedule of events. This module lets you define **scheduled events for entities** (characters, but also places or items) and takes care of activating or deactivating those events when appropriate.

**What it manages:** A collection of **schedules (events) for various entities**. Each schedule event defines *what*, *where*, and *when* something happens with an entity. For example, a schedule entry might say “Character John (entity) is in the Library (location) from 14:00–16:00 on weekdays, with status ‘studying’”. The Puppeteer keeps track of all such events.

**Key responsibilities:**

* **Defining Events:** You can add schedule events via the API by specifying:
* The entity **type** – e.g. 'person' for a character, 'place' for a location-specific event, 'clicky' or 'unclicky' for interactive or static objects (these are categories the system supports).
* The entity’s **cfname** (linking to an entry in the respective controller, like a person’s id).
* A unique **event id** for this entity’s schedule (e.g. "john\_morning\_class").
* **Status** – a label describing the entity’s state during this event (e.g. "in\_class"), often used to determine which sprite or behavior to use.
* **Location** – where this event takes place (a location key, matching one from your places if using them).
* **Position (x, y)** – coordinates on screen or background for where to show the entity during this event.
* **Day or Days** – which day(s) this event occurs. This can be a specific day number or a list of days (for recurring weekly events, you might specify, say, [1,2,3,4,5] for Monday–Friday).
* **Time Interval** – the start and end times for the event on those days, given as a string "HH:MM-HH:MM" (24-hour clock).
* **Arts (optional)** – an alternate image directory if needed (otherwise it defaults to using the entity’s cfname + status to pick images).

All these are stored in an **event dictionary** for that schedule entry. Once added, the event is managed by Puppeteer.

* **Activating/Deactivating Events:** The Puppeteer continuously (or upon request) **processes** these schedules to determine which events are “active” given the current time and day. For example, at 15:00 on Monday, the Puppeteer will mark all events whose interval covers 15:00 on Monday as active. Active events typically cause a change in the world, such as moving a person to a certain location, changing their status or sprite, etc. When the time moves outside an event’s interval, that event becomes inactive (allowing perhaps another event to take over or the character to be free). The event dictionaries have an isActive flag that Puppeteer uses – if set False, the event is ignored. Puppeteer’s process() method is responsible for checking current time against each event’s schedule and executing any needed changes. (In practice, this might update the Person’s location/status, or trigger other side effects you’ve defined in your game’s screens or logic.)
* **Cleaning and Consistency:** Puppeteer also provides utility methods to keep schedules clean. For instance, if an entity is removed or renamed, you can call a cleanup to remove orphaned schedule events that no longer refer to a valid person/place. This prevents leftover events from trying to activate for non-existent entities. There are also methods to manually remove events or fetch them.
* **Multi-entity scheduling:** Not limited to people, you can schedule **place events** (e.g., an event that at a certain time a location changes state or a shop opens/closes) or **object events** (maybe an item appears somewhere at a time). The kind parameter on schedule methods distinguishes these. This gives flexibility to treat various game elements as “puppets” that can be placed or activated on a schedule.

**Interactions:** Puppeteer is closely tied to both the Time and Person modules (and optionally a Place module if your game uses location dictionaries). It relies on the current time (from Time module) to know when to activate events. It also manipulates Person data – for example, when an event for a person becomes active, Puppeteer might update that person’s location and status to the ones defined in the event. Thus, the Person module’s data is dynamically changed by Puppeteer as time goes on. In a typical usage, you would call puppeteer.process() whenever time advances or when the player enters a new area, to ensure the world state is up to date for that time and location. Puppeteer ensures that NPCs follow the logic you set (their schedules) without you scripting every movement manually. You, as the writer, define the schedules; SOWRL’s Puppeteer handles the execution.

### Utils Module (utils)

The **utils module** encompasses miscellaneous supportive functionality, primarily the management of global **world state variables** and any general utilities needed across SOWRL. A key component likely included here is the **MachineController** (often accessible as game.machine\_controller or machine), which acts as a simple global database for arbitrary variables. In addition, utils might contain helper functions for things like image filename resolution or other minor features used by multiple modules.

**What it manages:** Broadly, anything that doesn’t fit in the core categories of time, events, persons, or schedules – typically global state and helper logic.

**Key responsibilities:**

* **Global Variable Store (World State):** SOWRL provides a way to create and manage global key-value pairs that represent the world’s state, through a “machine” or **variable controller**. This is essentially a dictionary where you can add any arbitrary data you want to keep track of (money, relationship points, plot flags, etc.) in a structured way. The Machine controller’s API allows adding a new variable, retrieving it, updating it, or removing it. For example, you might do machine.add("player\_money", 100) to store the player’s money, then later machine.set("player\_money", new\_value) to update it, or machine.get("player\_money") to read it. Storing these in SOWRL’s world state can be cleaner than using many separate Ren’Py variables, and it keeps related data in one place.
* **Utilities for Sprites/Images:** SOWRL’s controllers often need to figure out which image to show for a character given their current status and time of day. There are utility functions such as get\_image(obj\_arts, dp=True) that construct an image path given a base art directory and (optionally) the current day part. The logic typically is: if dp (day part) is True, it will choose an image appropriate for morning/afternoon/evening based on current time (e.g., appending a suffix or picking from subfolders), otherwise it ignores day part and just picks a default. These utilities are used internally by Person/Place controllers when an event’s status changes or when refreshing the scene.
* **Other Helpers:** The utils may also include other minor functions or classes used across the framework (for example, in the full framework OkiMaster, there were functions to refresh screens or handle color matrices for screen transitions – in SOWRL, not all of those visual extras may be present since it’s “backend”, but any non-core supportive code would reside in utils).

**Interactions:** The global variable store doesn’t directly interact with other modules unless you make it part of your game logic. It’s simply available for you to use and can be combined with timed events or conditions (e.g., a timed Event might, when firing, call machine.set("questX\_unlocked", True)). The image utilities are used by Person/Puppeteer behind the scenes to load appropriate images for characters based on status and time of day (for instance, if a character has arts="john" and status "sleeping", and day part night, get\_image("john\_sleeping", dp=True) might yield "john\_sleeping\_night.png" automatically depending on your file structure). As a developer, you might call these utilities if needed (for example, to get the file path of a sprite to display), but often the higher-level modules handle it.

Overall, the Utils module ensures the rest of SOWRL modules have the support and data they need (global variables, image lookup, etc.), helping tie the system together.

## API Reference

This section provides a reference for the main classes and methods provided by SOWRL. For each module, the key methods are listed with their purpose, parameters, return values (if any), and usage notes. Short examples (in Ren’Py Python context) are included to illustrate usage.

**Note:** In these examples, we assume you have the default shortcuts set (e.g. date = game.date\_controller, persons = game.person\_controller, puppeteer = game.puppeteer\_controller, machine = game.machine\_controller). If you didn’t set those, you can always call the methods via game (for instance, game.date\_controller.add\_hours(1) instead of date.add\_hours(1)). Also, many methods don’t return a value (they perform an action or update state), and thus return None unless otherwise noted.

### Time (DateController) – Methods for Time Management

* **get(keyname=None)** – Retrieve the current date/time information. If keyname is not provided, this returns a dictionary of all current time fields (hour, minute, day, etc.). If keyname is given (e.g. "hour" or "time"), it returns the specific value for that field.  
  *Example:*
* $ current\_time = date.get("time")  
  # current\_time might be "13:30" (a string) if it's 1:30 PM.
* **set(keyname, val)** – Manually set a specific field in the current date/time. Keyname could be "hour", "minute", "day", etc., and val is the value to set. Use this with caution – it’s generally better to use the add or set\_time functions for consistency.  
  *Example:*
* $ date.set("hour", 8)  
  $ date.set("minute", 0)  
  # Now the time is set to 08:00 of the current day.
* **set\_time(h, m)** – Set the current time to the given hour h and minute m. This will adjust the internal state (including the composite time string and day minutes count) accordingly. It does **not** change the day or other fields.  
  *Example:*
* $ date.set\_time(18, 30)  
  # Forces time to 18:30 (6:30 PM) of the same day.
* **add\_minutes(m)**, **add\_hours(h)**, **add\_days(d)** – Increment the current time by the specified minutes/hours/days. These methods will handle overflow (e.g. adding hours past 23 wraps to next day, incrementing day etc., according to the constraints set for the calendar). Adding days will progress the date and possibly month.  
  *Example:*
* $ date.add\_hours(1)  
  $ date.add\_minutes(30)  
  # This advances the time by 1 hour 30 minutes from whatever it was.
* After using these, you may want to call puppeteer.process() to update any scheduled events/NPC placements for the new time.
* **time\_to(target\_time\_str, d=0)** – Fast-forward time *forward* to a specific time of day. target\_time\_str is a string "HH:MM". Optionally, you can specify d as how many days in the future to jump (default 0 means today). This method will advance the clock to the next occurrence of that time. It returns the number of in-game minutes that have passed during the skip. This is useful if you want to, say, skip to a certain time (like “wait until evening”). Keep in mind that any timers (do\_in/do\_at) that would have executed in the skipped interval will execute *after* the skip is done (they don’t stop the skip) – effectively, the skip will trigger them immediately once the time is reached or surpassed.  
  *Example:*
* # It's 10:00 now, we want to jump to 18:00 (6 PM) today:  
  $ skipped\_minutes = date.time\_to("18:00")  
  $ puppeteer.process("person", current\_location, date.get("time"), date.get("monthday"))  
  # ^ after skipping, update NPC placements for current location and new time.
* If d was 1, it would jump to 18:00 tomorrow instead.
* **is\_in\_interval(time\_str, interval\_str)** – Check if a given time falls within a time interval. Returns True or False. time\_str is a time like "14:30", interval\_str is a range like "14:00-16:00". This can be used to easily test if now is within a window (though SOWRL uses this internally for schedule processing too).  
  *Example:*
* $ if date.is\_in\_interval("07:30", "07:00-09:00"):  
   "...It's the morning class period..."
* **do\_in(cfname, do, timepiece=0, sort='minutes')** – Schedule a **timer event** that will execute after a given delay. This creates a new one-time timer. Parameters:
* cfname: a unique name/ID for this timer event. (If a timer with the same name exists, it will be overwritten by this call.)
* do: a string of Python code to execute when the time is up. (This could be a call to a label, e.g. 'renpy.jump("some\_label")', or multiple statements separated by ;.)
* timepiece: the amount of time to wait, measured in the unit given by sort.
* sort: the unit of time for the delay – can be 'minutes', 'hours', or 'days'. By default, 'minutes'.

After calling do\_in, the timer is set. When the specified time passes (considering any time skips), the code will run once. The timer is then marked as executed and will not run again. If you create a new timer with the same name later, it resets to the new schedule.  
*Example:*

# Schedule an event 60 in-game minutes from now to notify the player  
$ date.do\_in("bell\_ring", 'renpy.notify("The bell rings in the distance.")', timepiece=60)

This will cause a Ren’Py notification to pop up after 60 minutes of game time have passed. If you fast-forward time beyond 60 minutes at once, the notification will appear immediately after the skip (since the timer would be due).

* **do\_at(cfname, do, when, days=0)** – Schedule a **timer event at a specific time**. This is similar to do\_in, but instead of “delay X minutes” you specify an absolute time. Parameters:
* cfname: unique name/ID for the timer (again, it will overwrite an existing timer with the same name).
* do: Python code string to execute at that time.
* when: a target time as a string "HH:MM" for the event to occur.
* days: how many days in the future to schedule it. 0 means today at the next occurrence of when that is *after* the current time. (If the current time is already past the when time and days=0, it likely will schedule for that time on the next day; alternatively, you can specify days=1 explicitly for next day, etc.)

Use do\_at for things like “every day at 6:00 PM, do X” (you would call it each day, or set multiple days via multiple timers), or “on Day 7 at 08:00, trigger an event.” Remember that if the player skips over the exact time via time\_to or large add\_hours, the event will fire after the skip (since the time was reached). Also, a do\_at timer does not repeat automatically – if you want something to happen every day at a time, you might need to reset it or call do\_at again in the code each day (or use the Puppeteer for daily recurring events). *Example:*

# Assume it's Day 3 and we want an event on Day 5 at 09:00  
$ date.do\_at("festival\_start", 'renpy.call("start\_festival")', when="09:00", days=2)

This will call the label start\_festival at 09:00 two days from now (i.e., Day 5). If the game time reaches that point, the code executes once.

* **set\_timetask(cfname, keyname, value)** – Edit a scheduled timer’s property. This allows you to modify an existing timer event by specifying its name and the field to change. Common uses might be to flag it executed or to change the do code or delay, though typically you won’t use this often. (Possible keyname values include 'do', 'day' etc., but changing them manually can be tricky. Usually easier to remove and add anew.)  
  *Example:*
* # Postpone an event by a day (this assumes the timer stores 'day' as its target day count)  
  $ date.set\_timetask("festival\_start", "day", date.get("day") + 1)
* (Use with caution – depends on internal structure of timer. Many times you’ll prefer removing and scheduling a new one.)
* **remove\_timetask(cfname)** – Cancel a timer event by name. This deletes the scheduled event with the given name, if it exists, so it will not fire.  
  *Example:*
* $ date.remove\_timetask("bell\_ring")  
  # Cancels the "bell\_ring" event we scheduled earlier, if it's still pending.

**Timer Details:** Each timer scheduled via do\_in/do\_at has internal fields like cfname (its name), typetask ("do\_in" or "do\_at" to indicate how it was set), do (the code to run), day and dayMinutes (when it should execute, in terms of target day and time in minutes), and executed (flag if it has run already). Typically you won’t directly interact with these fields, but it’s good to know they exist (for example, executed=True means that timer already ran and won’t run again).

### Person (PersonController) – Methods for NPC Management

* **add(name, surname, cfname, location, x, y, arts=None)** – Create a new person (NPC) in the world. This returns the newly created person dictionary (or None). Parameters:
* name: The character’s display first name (string).
* surname: The character’s surname (string).
* cfname: A unique identifier for the character (string, no spaces). This is the key you’ll use to reference this character later. It’s also used by default to find sprites.
* location: The initial location key where this character is. This can be a string (one location) or a list of location keys if the character can appear in multiple places (initially). Typically it’s a single location like "school\_gate". You can use a special value like "" or an out-of-the-way location if the character shouldn’t start visible anywhere.
* x, y: Numbers (0.0–1.0 typically) for the on-screen position of the character. If you’re using an image map or coordinate system for locations, these coordinates will be used to place the character’s image on the screen. (They can represent fraction of screen width/height or absolute pixels depending on your usage, but commonly 0.5,0.8 might mean center horizontally and lower on the screen vertically.)
* arts (optional): If provided, this string will be used as the *base directory or name for the character’s sprite images* instead of cfname. If not provided, the system will assume image assets are organized under a folder or prefix named after cfname. For example, if cfname="john", by default it will look for images in a folder "john" or files named starting with "john\_". If you set arts="protagonist", it would use that instead.

This method registers the character in the person database. It does **not** automatically display the character on screen (that’s up to your screens), but the data will now be available for the game logic and other SOWRL modules (like Puppeteer).  
*Example:*

$ persons.add("John", "Doe", "john", "school\_gate", 0.5, 0.9)  
$ persons.add("Jane", "Smith", "jane", "", 0, 0, arts="jane") # Jane starts off-screen.

Now we have two characters. We might define an image for John like "john\_idle.png" in an images folder, and similar for Jane.

* **get(cfname, keyname=None)** – Retrieve information about a person. If only cfname is given, you get the full dictionary for that character (or False if not found). If you also provide keyname, you get just that specific property’s value. This is useful to check a character’s current state.  
  *Example:*
* $ john\_data = persons.get("john")  
  $ current\_loc = persons.get("john", "location") # e.g. returns "school\_gate"  
  $ if persons.get("john", "isActive") == False:  
   "...John is currently not in the world."
* **set(cfname, keyname, val)** – Update a property of a person. This finds the character by cfname and sets the given keyname to val. Common uses: toggling isActive, changing location, updating status, etc., during gameplay. This method returns nothing.  
  *Example:*
* $ persons.set("john", "location", "library")  
  $ persons.set("john", "status", "reading")  
  $ persons.set("john", "isActive", True)
* This would mark John as active and move him to the library with status “reading”. If you have a schedule running, be cautious: if an active Puppeteer event is also trying to control John, it might override these changes. Usually, you use set when you want to manually override or initialize something about the character.
* **get\_all\_from\_location(location='')** – Get a **list of all characters present at a given location**. This returns a list of person dictionaries (or IDs) that have their location matching the given location string. If the location parameter is '' (empty string) or not provided, it might return all characters with no specific filter (or possibly all characters with an empty location). In practice, you’d call this in a screen to find who to show at the current location.  
  *Example:*
* $ people\_here = persons.get\_all\_from\_location(current\_location)  
  for person in people\_here:  
   text "[person['name']] is here." # Or use their image etc.
* **get\_image(obj\_arts, dp=True)** – Get the filepath of an image for a given art identifier and (optional) day part. Usually you won’t call this directly; it’s used internally to construct the image field for a character or location based on their arts folder and current time. If dp=True, it will append or choose a subfolder for the current part of day (for example, it might pick "morning" vs "night" variant). If dp=False, it ignores the time of day. This function is generic and can be used for any “object arts” (characters, places, etc.).  
  *Example:*
* $ sprite\_path = persons.get\_image("john\_idle", dp=True)  
  show image sprite\_path
* (Under the hood, if it’s evening and dp=True, it might transform "john\_idle" to "john\_idle\_night.png" if such an image exists.)
* **remove(cfname)** – Remove a person from the database. This deletes the character with the given cfname entirely. Use this if a character is permanently leaving the story and you want to clean up. If you might bring them back, you could also set them isActive=False instead. Removing does not automatically remove any schedule events tied to that person – you should also call puppeteer.remove('person', cfname) to clear their schedule events if any, or use puppeteer.clean() (see Puppeteer methods below) to clear out events referencing nonexistent characters.  
  *Example:*
* $ persons.remove("jane")  
  $ puppeteer.remove("person", "jane")
* This ensures Jane and any schedule for Jane are gone.

### Puppeteer (PuppeteerController) – Methods for Schedule Management

* **add(kind, cfname, id\_, status, location, x, y, day, interval, arts=None)** – Create a new schedule **event** for an entity. This returns the created schedule dictionary, or None on failure. Parameters:
* kind: the type of entity – one of 'person', 'clicky' (interactive object), 'unclicky' (non-interactive object), or 'place'. Most of the time you will use 'person' here to schedule a character.
* cfname: the unique name of the entity (for person, the character’s cfname as used in Person module; for place, the place’s id, etc.).
* id\_: a unique identifier for this event (string). It should be unique per entity (and best unique across all for clarity). For example, "john\_school\_morning" or "library\_event1". This is how you’ll reference or modify this event later if needed.
* status: a label for the entity’s status during this event. It can be any string like "studying" or "open" or "asleep". It is used for things like picking the right sprite (combining with arts) and can also inform game logic if you check a person’s status.
* location: the location key where this event occurs. This should correspond to some location in your game (if you have a PlaceController or at least a consistent naming for locations). When this event is active, the entity will be considered to be at this location.
* x, y: coordinates for where to place the entity (if applicable) during the event. Same meaning as in Person.add – these might correspond to screen positions on the location’s background.
* day: the day(s) on which the event occurs. This must be given as a list of day numbers. You can supply a single day number (like 3 for day 3 of the game) or a list like [1,3,5]. For recurring weekly schedules, you might reset day numbering every week – for instance, day 1 could mean Monday of each week (depending on how you use it). Usually, if your game is not extremely long, you might just count days from start. **Important:** Even if one day, it should be in a list form (e.g. [3]).
* interval: a string "HH:MM-HH:MM" specifying the start and end time of the event within each of those days. The event will be active *from* the start time up to *until* the end time (assuming end is later on the same day; it doesn’t currently support an interval that wraps past midnight).
* arts (optional): an alternate art identifier to use for images during this event. If None, it defaults to using the entity’s own arts plus the status to find the image (i.e., effectively arts = cfname + "\_" + status by default). If you want to override where to get the sprite for this event, you can provide a different base name.

This method records the event in the schedule. The event won’t do anything until the appropriate time comes, at which point Puppeteer will “activate” it (usually by updating the Person or Place data). If an event is added that is currently in-progress (for example you add an event that covers 10:00-12:00 and right now it’s 11:00), you may need to call process to immediately apply it.  
*Example:*

# Schedule John to be in the library every weekday (days 1-5) from 14:00 to 16:00.  
$ puppeteer.add('person', 'john', 'john\_library\_afternoon',  
 status="studying", location="library", x=0.4, y=0.8,  
 day=[1,2,3,4,5], interval="14:00-16:00")

Now John has an event “john\_library\_afternoon”. When the game’s day is 1-5 (Mon-Fri, if we consider day1 Monday) and the time hits 14:00, this event becomes active: SOWRL will mark John’s status as “studying”, set his location to "library", etc., and (depending on your screens) John’s sprite would appear at the library coordinates. After 16:00, the event ends and John might either disappear (if no other event) or move to another event location (if another event starts).

* **get(kind, cfname, id\_=None, keyname=None)** – Retrieve schedule event data. This lets you inspect the scheduled events.
* If you provide id\_ of an event, you get that specific event’s dictionary (or a specific keyname within it, if provided).
* If you call with id\_=None, it will return **all** events for that entity (a dict of event-id -> event-data).
* If you also give a keyname in that case, it might return a dict of just that property for all events, but typically you use keyname only when looking at a single event.
* kind and cfname specify which entity’s schedule to look at (similar to above).  
  *Example:*
* $ event\_data = puppeteer.get('person', 'john', 'john\_library\_afternoon')  
  $ next\_time = puppeteer.get('person', 'john', 'john\_library\_afternoon', 'interval')  
  # next\_time would be "14:00-16:00" in this case.  
  $ john\_events = puppeteer.get('person', 'john') # get all John's events as a dict
* **set(kind, cfname, id\_, keyname, val)** – Modify a property of a scheduled event. This is analogous to person.set but for schedule events. For instance, you can toggle an event’s isActive flag off to disable it, or change its interval or location. Use with care, as altering a schedule while it’s running can have immediate effects (e.g., if you set interval to something that doesn’t include the current time, the Puppeteer might immediately consider it inactive). Common usage might be to deactivate or reactivate certain events via setting isActive.  
  *Example:*
* $ puppeteer.set('person', 'john', 'john\_library\_afternoon', 'isActive', False)  
  # Temporarily disable John's afternoon library event.
* (If John was there currently, you might want to also manually move him or call process to update.)
* **process(kind, location, day\_time, month\_day)** – Process (evaluate and execute) schedule events for a given type and location. This is a crucial method that you or the SOWRL framework needs to call at appropriate times (often automatically in the background). It checks all events of type kind (e.g., 'person') to see which ones should be active given the current time, and for those that are active and match the given location and current time, it will apply them. The parameters:
* kind: as usual, the category of events (you might call separately for 'person', 'place', etc., or 'all' if supported, but usually per type).
* location: the location context to process. Typically, you pass the player’s current location or the location you are about to display. The Puppeteer will then activate any events that should happen **at that location**. Events scheduled for other locations might be ignored in this call (and would be processed when those locations become relevant).
* day\_time: a time string (HH:MM) representing the current time of day to check against event intervals. Often you pass game.date\_controller.get("time") here – i.e., the current time.
* month\_day: the current day of the month (an integer) to check against the event’s day list. You’d typically pass game.date\_controller.get("monthday") or similar here.

In practice, you might call puppeteer.process('person', current\_location, current\_time, current\_day) whenever the scene is updated or time changes, to ensure NPCs are where they should be. The documentation notes that **you usually don’t need to manually call** this in simple scenarios, because if you use SOWRL’s provided screens they might call it automatically every frame. But if you are handling your own screen logic, you should call it at least whenever the player enters a new location or after time advances significantly. This method doesn’t return a value; it updates the world (e.g., toggling events, moving characters). Also note: Schedules are currently confined to one in-game month – events with days beyond the current month aren’t handled by a single call (you’d manage multi-month logic manually if needed).  
*Example:*

label morning\_routine:  
 $ date.add\_hours(2) # 2 hours pass (morning to midday)  
 $ puppeteer.process('person', player\_location, date.get("time"), date.get("monthday"))  
 "...Time for the next class..."  
 return

* **clean(kind, puppetdict)** – Clean up schedule events that reference nonexistent entities. This is a maintenance method. puppetdict should be the current dictionary of entities of that kind (for persons, you’d pass the entire persons database, e.g. persons.get() which might return the dict of all persons). The Puppeteer will compare its schedule entries for kind against the provided dict and remove any schedule events for which the target cfname is not found in the dict. In short, call this after you remove entities or if you suspect some schedules are dangling. It returns nothing.  
  *Example:*
* $ puppeteer.clean('person', persons.get())   
  # Removes any events for characters that no longer exist in persons.
* **remove(kind, cfname=None, id\_=None)** – Remove scheduled events. This method has two modes:
* If you provide a cfname and an id\_, it will remove that specific event for that entity.
* If you provide only cfname (and leave id\_ as None), it will remove **all events for that entity** (wipe their schedule).
* If you provide neither (both None), it will remove **all events of that kind** (e.g., wipe all person schedules – use carefully!).
* If you provide kind and set cfname=None, it will remove all events of that kind (which could be everything for all persons, etc.).

This is useful to cancel events dynamically or clear an entity’s schedule if, say, their routine changes or they leave the game.  
*Example:*

$ puppeteer.remove('person', 'john', 'john\_library\_afternoon') # remove one event  
$ puppeteer.remove('person', 'john') # remove all of John's events

### Utils (MachineController & Others) – Utility Methods

* **machine.add(varname, value)** – Create a new global variable in the world state. varname is a string key, and value can be any value (number, string, list, etc.). If a variable with that name already exists, this will overwrite it or update it (acts essentially like a dictionary assignment). Use this at game start to initialize important global values, or during the game to introduce new tracked state.  
  *Example:*
* $ machine.add("fame", 0) # track player's fame stat  
  $ machine.add("guild\_joined", False) # a flag for story event
* **machine.get(varname=None)** – Retrieve global variable(s). With no arguments, this returns a dictionary of **all** stored variables and their values. If varname is provided, it returns the value of that variable, or False if that key doesn’t exist. (Note: returning False for missing keys is a design choice; so store booleans carefully – you might not be able to distinguish a False value vs. not present except by checking var in machine.get().)  
  *Example:*
* $ stats = machine.get() # get all world variables  
  $ fame\_points = machine.get("fame") # get current fame (e.g. 0)  
  $ if machine.get("guild\_joined"):  
   "...Welcome back to the guild!"
* **machine.set(varname, value)** – Update an existing global variable’s value. If the variable doesn’t exist, this will **not** create a new one (to avoid typos silently making new entries). It will simply do nothing (or return False) if the key isn’t found. Make sure to add first before set.  
  *Example:*
* $ machine.set("fame", machine.get("fame") + 10) # increase fame by 10  
  $ machine.set("guild\_joined", True) # mark that the guild was joined
* **machine.remove(varname)** – Delete a global variable. Removes the key and its value from the world state.  
  *Example:*
* $ machine.remove("fame")
* **Image Utility:** *(get\_image is available through person, place, etc. as shown above.)* There may also be a global version or similar utility in the utils module depending on implementation. Its usage is identical to described: it constructs image paths for a given base name and day part selection.
* **Other Utility Methods:** Depending on SOWRL version, there might be other minor methods (for example, in OkiExtra there were methods to refresh screens or handle color effects, but those may not be included in SOWRL if it’s “backend only”). If present, they would follow similar patterns – e.g., game.refreshScreen() to force UI update, etc. These are not core to world state or schedule, so refer to SOWRL’s notes or code if needed.

## Usage Scenarios

Finally, let’s walk through a few typical scenarios to see how SOWRL modules come together in practice. These examples illustrate how you might use the API in your game code:

### 1. Creating NPCs and Placing Them in the World

Suppose we are making a slice-of-life visual novel set in a town. We want to create some characters and have them appear in certain places at game start.

**Step 1: Define characters** using the Person module:

# In an init or label after SOWRL is initialized:  
$ persons.add("Alice", "Brown", "alice", "park", 0.5, 0.8)  
$ persons.add("Bob", "Carter", "bob", "coffee\_shop", 0.3, 0.9)

This creates Alice and Bob. Alice starts at the park (at coordinates centered at 50% width, 80% height), Bob at the coffee shop. Both are active by default. If we have images for them (e.g., in folders “alice” and “bob”), we can set up a screen to show characters present at a location:

screen location\_scene(loc):  
 # ... background ...  
 for person in persons.get\_all\_from\_location(loc):  
 add Image(persons.get("{}".format(person["cfname"]), "image")) xpos person["x"] ypos person["y"]  
 text "[person['name']]" xpos person["x"] ypos person["y"] - 20

*(This is a conceptual example – in practice, you might integrate with ATL transforms or image definitions. The key is that you use persons.get\_all\_from\_location to find who to display.)*

**Step 2: Interact with characters:** Because each person has an action (like "person\_alice"), we can define labels or use the action field to respond to clicks. For example, define in script:

label person\_alice:  
 "Alice": "Hi there! Lovely day in the park, isn't it?"  
 return

And in the screen, if we want clicking Alice to trigger that, we could do:

imagebutton auto "alice\_avatar.png" xpos person["x"] ypos person["y"] action Jump(person["action"])

Since by default person["action"] for Alice would be "person\_alice", it jumps to that label. Alternatively, we could have set a different action via persons.set("alice", "action", "some\_other\_label") if we want a custom handler.

### 2. Triggering Scheduled NPC Movements (Daily Routine)

Continuing our example, let’s give Alice a daily routine: she stays in the park in the morning and goes to the cafe in the afternoon.

**Step 1: Define locations (optional):** If using a Place controller (not explicitly covered above, but assume we have one), we’d have added places with places.add("Park", "park"), places.add("Coffee Shop", "coffee\_shop"), etc. They would each have coordinates and images too. But even without that, we can reference location keys consistently (strings like "park", "coffee\_shop").

**Step 2: Schedule events for Alice:**

# Alice's morning in park: everyday from 08:00-12:00  
$ puppeteer.add('person', 'alice', 'alice\_morning\_park',  
 status="relaxing", location="park", x=0.5, y=0.8,  
 day=[1,2,3,4,5,6,7], interval="08:00-12:00")  
# Alice's afternoon in coffee shop: everyday from 12:00-18:00  
$ puppeteer.add('person', 'alice', 'alice\_afternoon\_cafe',  
 status="working", location="coffee\_shop", x=0.4, y=0.9,  
 day=[1,2,3,4,5,6,7], interval="12:00-18:00")

We gave Alice two events. In the morning interval she’s at the park (status “relaxing”), in afternoon at the coffee shop (status “working”). We used days 1-7 meaning every day of the week (assuming our game just counts days continuously).

**Step 3: Advance time and process events:** Initially, if the time at game start is, say, 09:00 Day 1, the morning event should be active. We should call process to enforce that:

$ date.set\_time(9, 0) # start at 09:00  
$ puppeteer.process('person', 'park', date.get("time"), date.get("monthday"))

Now Puppeteer will see that alice\_morning\_park covers Day1 08:00-12:00 at location "park", and since current time 09:00 falls in that interval and location is "park", it will mark Alice’s event active and likely update Alice’s data: persons.get("alice", "location") becomes "park", status becomes "relaxing", etc.. On our location screen for park, Alice will appear (with possibly an image corresponding to relaxing).

Later, when time passes beyond 12:00, say we do:

$ date.time\_to("12:00") # fast-forward to noon  
$ puppeteer.process('person', 'park', date.get("time"), date.get("monthday"))  
$ puppeteer.process('person', 'coffee\_shop', date.get("time"), date.get("monthday"))

At 12:00, the first event ends and the second begins. We call process for both relevant locations. The one for 'park' might find no active events for Alice now (so Alice can be removed from park), and the one for 'coffee\_shop' will find that alice\_afternoon\_cafe should activate (Alice appears at the cafe). In a real game, you might only call process for the location the player is currently in or going to, but if you want to simulate off-screen movements you can process globally or for all major locations.

The result: Alice automatically moves according to schedule without explicit scripting at that moment. If the player is in the park at noon, they might see Alice leave (if your screen updates accordingly). If they then go to the coffee shop after noon, they’ll find Alice there.

You can extend this scenario to many characters and complex schedules similarly.

### 3. Using Timers for Events and World State Changes

Now imagine an event: every day at 18:00, a bell rings in town, and after it rings 5 times, some global flag triggers a festival.

We can use the Event (timer) system plus the Machine (world state).

**Step 1**: Set up a machine variable for bell count:

$ machine.add("bells\_rung", 0)

**Step 2**: Schedule the bell ring event to occur daily. We could do this via do\_at in a loop or each day. Since SOWRL timers are one-shot, one approach is: each time the bell rings, we reschedule it for the next day.

For the initial day:

label start\_day:  
 # ... other morning init ...  
 $ date.do\_at("daily\_bell", 'renpy.call("bell\_ring")', when="18:00", days=0)  
 # schedule today's bell at 18:00  
 return

We create a Ren’Py label bell\_ring to handle the ringing:

label bell\_ring:  
 # This code runs at 18:00 due to the timer.  
 $ renpy.notify("The town bell chimes loudly in the distance.")  
 $ machine.set("bells\_rung", machine.get("bells\_rung") + 1)  
 # If 5 bells have rung in total, trigger festival event.  
 if machine.get("bells\_rung") >= 5:  
 $ date.do\_at("festival\_event", 'renpy.jump("festival\_start")', when="12:00", days=1)  
 # The festival will start tomorrow at noon.  
 else:  
 # Schedule the bell for next day again  
 $ date.do\_at("daily\_bell", 'renpy.call("bell\_ring")', when="18:00", days=1)  
 return

Here’s what happens: The first timer “daily\_bell” goes off at 18:00 of Day 1, jumping to bell\_ring. We increment the counter and check it. If it’s less than 5, we schedule the next day’s bell by calling do\_at again with days=1. This effectively re-arms the daily bell timer for the following day at 18:00. If it was the 5th ring, we instead schedule a festival event for the next day at noon by another timer “festival\_event”. (We remove or simply don’t schedule daily\_bell anymore after that, since perhaps the daily ringing stops during the festival or is no longer relevant.)

This demonstrates using do\_at in a recurring fashion and using a world state variable to keep track of something (the count of rings). The festival\_event could trigger a label that perhaps changes the location states or starts a special scene.

### 4. Manipulating World State and Schedules On the Fly

Consider a scenario where an NPC’s routine can change based on player choices – say Bob will start going to the gym every evening if the player convinces him to train.

Initially, Bob might have no evening event. Later in the game, at runtime, we can add or enable one.

**Initial**: Bob’s schedule only morning work:

$ puppeteer.add('person', 'bob', 'bob\_morning\_work',  
 status="working", location="office", x=0.5, y=0.9,  
 day=[1,2,3,4,5], interval="09:00-17:00")

No evening event yet.

**Player choice triggers training**:

menu:  
 "Invite Bob to join the gym":  
 $ machine.set("bob\_training", True)  
 "Bob": "Alright, I'll start going to the gym in the evenings."  
 $ puppeteer.add('person', 'bob', 'bob\_evening\_gym',  
 status="training", location="gym", x=0.6, y=0.8,  
 day=[1,3,5], interval="18:00-20:00")

We set a flag bob\_training in the machine (not strictly necessary, but maybe used elsewhere for story logic) and then add a new puppeteer schedule for Bob: on Mondays, Wednesdays, Fridays (days 1,3,5) from 18:00–20:00 at the gym. Now Bob has a new event. If the time is currently within that interval on one of those days, you might call puppeteer.process('person', 'gym', current\_time, current\_day) to immediately reflect it. Otherwise, it’ll naturally take effect when the time comes.

If later on Bob stops training:

$ puppeteer.remove('person', 'bob', 'bob\_evening\_gym')  
$ machine.set("bob\_training", False)

This cancels the gym schedule. Alternatively, we could have toggled it with puppeteer.set(... 'isActive', False) if we wanted to potentially re-enable it later.

### 5. Querying Schedules and State for Gameplay

Sometimes you need to query the system to decide game logic. For example, you might want to check if an NPC is currently free before assigning them a task, or find out how many days until a specific event.

Using **Puppeteer.get**, you can inspect events. E.g., to find when Alice’s next event starts or ends relative to now, you could:

$ alice\_events = puppeteer.get('person', 'alice')  
$ for eid, event in alice\_events.items():  
 if event['isActive']:  
 "[event['cfname']] is doing [event['status']] at [event['location']] until [event['interval'].split('-')[1]]."

This could output something like “alice is doing relaxing at park until 12:00.” – giving the player info on schedule.

Or perhaps you want to skip time to when a certain place opens. You might store in machine the next opening time, or if it’s daily, just call date.time\_to("09:00") for next morning.

In summary, by combining the **Time**, **Event**, **Person**, **Puppeteer**, and **Utils** (Machine) modules, you can implement rich simulation behavior: characters that move on their own schedule, events that trigger after delays, and global variables that track the evolving world.

## Best Practices

To get the most out of SOWRL and avoid common pitfalls, consider the following guidelines:

* **Unique Naming:** Ensure all your important entities and events have unique identifiers:
* Each character’s cfname should be unique (e.g., don’t have two different characters both called "alex"). Many systems (person, puppeteer, machine) use these keys to reference entities.
* Each scheduled event id per entity should be unique. It’s wise to incorporate the entity’s name into the event id (e.g., "alex\_work\_morning") so that even if different entities coincidentally have an event with the same purpose, their IDs don’t clash. Remember that timer events (do\_in/do\_at) also use a global namespace for cfname – scheduling a timer with an existing name will overwrite the old one. Use descriptive names for timers (e.g. "festival\_timer" rather than something generic like "event1").
* If you rename the main game object or any controller shortcuts, update all references accordingly. Consistency is key.
* **Controller Shortcuts:** As shown, define shortcuts like persons, puppeteer, date, machine for convenience. This makes the code more readable and avoids deep game.x.y calls throughout. You can do this via Ren’Py’s define or default after creating game. (If SOWRL didn’t already define them internally.)
* **Using do\_in vs do\_at:** Use do\_in for one-off delays when the exact target time isn’t important, just the duration (e.g., “5 minutes later do X”). Use do\_at when it matters that it happens at a specific clock time (e.g., “at midnight trigger Y”). Keep in mind that do\_at is tied to the in-game clock – if the player never reaches that time (because you ended the day early), the event might not fire unless time is advanced. Also note, as time can be skipped, both do\_in and do\_at will *not* interrupt a big skip; they’ll execute after the skip completes if their time was passed.
* **Recurring Events:** The Event timers are best for things that happen once or a limited number of times. For daily or routine events, it’s often better to use the Puppeteer schedules (since they inherently repeat each day or set of days). Alternatively, as shown above, you can reschedule a do\_at inside its own code to achieve repetition, but that can get unwieldy for many events. A rule of thumb: **use Puppeteer for regular schedules attached to characters or places** (NPC routines, shop open hours), and **use timed events for story triggers or global countdowns** (delayed quest event, periodic notification, etc.).
* **Manual Time Skips vs. Real-Time:** If your game allows the player to skip time (sleep until evening, etc.), prefer using time\_to() or a loop of add\_hours rather than setting the time abruptly. This ensures the Event module catches up correctly. After skipping, always call puppeteer.process to update NPCs. If you have multiple entity types, call it for each type or provide a way for the framework to update all (depending on implementation).
* **Keep Schedules and Data in Sync:** If you remove or deactivate a character, also remove or deactivate their scheduled events. For example, if a character leaves the game (set isActive=False or removed), call puppeteer.remove('person', cfname) or puppeteer.clean('person', persons.get()) to clear their events. Conversely, if you add a character mid-game, you can immediately assign them schedules.
* **Avoid Overlapping Events for one Entity:** It’s generally best not to schedule two events for the same person that overlap in time. The system does not inherently prevent overlapping intervals. If overlaps happen, the behavior might be undefined – one event might simply override the other or the last processed one wins. Design schedules so that a character’s events either don’t overlap or if they might, ensure one is inactive while the other is active (maybe using isActive flags or different day conditions).
* **Use isActive Flags for Temporary Control:** If you want to temporarily suspend a schedule event without removing it (say an NPC is sick and not following their routine today), you can set that event’s isActive to False via puppeteer.set. This will cause Puppeteer to ignore it until you set it True again. This is easier than removing and later re-adding the event. Alternatively, adjust the day list to exclude the current day if it’s just for today.
* **Leverage machine for global conditions:** Instead of using a bunch of separate Ren’Py variables, use machine to group related world state. For instance, all your city reputation or event flags can be keys in machine. This makes it easy to save/load as one structure (Ren’Py will include game in the save since it’s a default store object) and avoids cluttering the global namespace. It also pairs well with timers and events (which can check or set those machine values). However, don’t overuse it for trivial things that could just be local variables – focus on state that multiple parts of the game need to know about.
* **Coordinate with Ren’Py Save/Load:** All data stored in SOWRL (time, persons, schedules, machine) lives in Ren’Py store variables (like game and those default shortcuts). This means they are automatically saved and loaded with the game state, which is good. But ensure you’re not putting unserializable objects in the machine or person dictionaries. Stick to basic data types (int, str, list, dict, bool) or Ren’Py store objects. Avoid storing things like open file handles or complex Python objects that Ren’Py’s pickler might not handle.
* **Screen Updates:** If you design custom screens to show NPCs, remember to update them when events trigger. You can use the renpy.restart\_interaction() or UI functions to refresh screens after calling puppeteer.process. A common pattern is to call process at the start of a location screen or whenever the player moves to a new area/time.
* **Day/Night Visuals:** The use\_daypart flag on persons/places and how get\_image chooses images implies you might have different images for morning/afternoon/evening. Define a convention (e.g., append \_night or \_morning to filenames) and let SOWRL help pick the right one by keeping use\_daypart=True. This way, as time changes, you could simply call something like persons.set("alice", "image", persons.get\_image("alice\_"+alice\_status, dp=True)) whenever her status or daypart changes to update her sprite.
* **Testing and Debugging:** Use the get() methods to your advantage while debugging. For example, you can print out puppeteer.get('person', 'alice') in a console or log to see what events are scheduled, or check date.get() to see if time rolled over correctly, etc. This can help ensure your schedules are set up as you expect. Also, be mindful that all times are strings and 24-hour format – a common mistake is to provide like "8:00" without zero-padding (which might be interpreted as 8 minutes, not 8 hours!). Always use two-digit hours.
* **Game Design Consideration:** SOWRL adds a simulation layer to Ren’Py – but Ren’Py is still fundamentally a VN engine. Make sure the addition of open-world elements doesn’t conflict with Ren’Py’s flow. For example, if you have a lot of parallel events, decide how the player experiences them (Ren’Py can’t spontaneously jump to a new scene without your instruction even if an event triggers – you might use notifications or variables to cue the player). Timers executing code like renpy.jump will interrupt the current interaction, so use with care (possibly schedule at transition times or use renpy.notify or flags to handle events more gracefully). In other words, integrate the SOWRL-driven events with your narrative in a way that feels natural.
* **Performance:** Generally, SOWRL is lightweight (checking a bunch of events or adding a few hundred entries is fine). However, avoid extremely tight loops of time advancement with process calls every minute of game time if you have thousands of events – try to jump in larger chunks or strategically call process. Also, cleaning up unused data (characters, events) as recommended will prevent slowdowns due to accumulation.
* **Organization:** Even though SOWRL helps organize data, as your game grows, keep your schedules and world state definitions organized. Perhaps maintain a separate script file where you define all characters and base schedules (like a setup phase), and then only modify them in the game flow as needed. This way you have a clear reference of what the “normal” world behavior is.

### Naming Conventions

There is no enforced naming style in code, but here are suggestions for clarity: - Use lowercase with underscores for cfname (internal names) of entities, to distinguish from displayed names. e.g., alice, bob, old\_man, etc. This aligns with how Ren’Py often expects labels or images names. - For schedule id\_, include the entity and a brief context: <name>\_<context> or <name>\_<location>\_<time> etc., as seen earlier. e.g., alice\_afternoon\_cafe or bob\_gym\_evening. This makes it easy to identify what an event is for in debug output. - For machine variables, you can use descriptive keys, possibly grouped by feature: e.g., quest\_guild\_joined, stat\_strength, etc. This reduces risk of key collision and helps readability. - If you utilize the action field for person interactions, keep the default pattern of "person\_"+cfname or similar patterns for consistency, unless there’s a reason to do custom. That way you immediately know what label to write for an NPC interaction.

By following these best practices, you’ll keep your implementation robust and maintainable, making it easier to scale your game’s complexity.

## Known Limitations

While SOWRL greatly expands Ren’Py’s capabilities, it does have some limitations and design assumptions to be aware of:

* **Time System Limits:** The in-game time tracking does not inherently manage years or calendar months beyond what is needed for a typical game. It tracks up to the month level and week/day names, but the concept of a “month” in SOWRL may be a fixed length (e.g., 30 days) unless configured. The framework mentions that schedules are confined to one game month, meaning if your game spans multiple months and you want an event on day 45, you might need to reset or recreate events when a new month starts. Check okistore.rpy or configuration for any constants (like days per month) and adjust them if your game needs more. There isn’t a built-in year or date object that automatically progresses beyond the month – if your story needs a calendar over multiple months, you’ll have to handle the transition in logic (possibly by incrementing a “month” variable and resetting “monthday”).
* **No Native Repeat/Loop for Timers:** Timer events (do\_in, do\_at) do not repeat on their own. You must manually reschedule them if you want something to happen regularly. There’s no cron-like feature (aside from using Puppeteer for daily repeats as discussed). This is by design, keeping timers simple.
* **Concurrent Event Handling:** If two timers are set to fire at the exact same time, they will both execute, but the order is not strictly guaranteed unless you know how the internal list is sorted. It’s usually by insertion order. This typically isn’t an issue, but if order matters, consider staggering them by a minute or use one timer’s code to call multiple outcomes.
* **Threading and Real-time:** SOWRL is not real-time or multithreaded. It runs in the Ren’Py execution thread. Timed events are tied to game time, not real elapsed time. So if the player stays on a pause screen or saves and loads, in-game time doesn’t change unless your code advances it. Nothing will “happen” in the background until you advance the time or call a relevant function. This is consistent with Ren’Py’s turn-based nature.
* **UI Integration:** Out of the box, SOWRL doesn’t include on-screen widgets or menus (except any you create) to show the time, list schedules, etc. You will need to create a UI for the player to see the current time/day or to navigate locations where characters are. The “views” part of the original framework was not included as it’s a backend. This means you have freedom to design your own interface, but also an extra step. For example, to display time, you can do text "Time: [date.get('time')] Day [date.get('day')]" in a screen, updating each frame. For characters, as shown, you loop through persons.get\_all\_from\_location. None of that is automatic. The limitation is simply that SOWRL doesn’t come with a pre-made HUD or location system – it gives you the data to build one.
* **Assumptions about Scale:** The system works best for a game with a moderate number of entities and events (tens of characters with tens of events each, for example). If you try to have, say, 1000 NPCs each with complex schedules, you might run into performance issues when processing events. There’s no hardcoded cap, but consider optimizing or simplifying schedules if you have a very large world. In normal VN/sim use cases, this is not a problem.
* **Hardcoded Behavior:** Some behaviors in SOWRL are somewhat hardcoded but can be changed by editing the scripts:
* The default naming of interaction labels (person\_cfname) is an assumption; if you don’t want that, you have to manually change each character’s action or alter the code that sets it (in OkiPersonController init, presumably).
* The calculation of dayPart (morning/afternoon/evening indices) and how get\_image picks images might assume specific cutoff times (e.g., maybe 6:00 for morning, 18:00 for evening). These might be defined in a config or in code (possibly okistore.rpy). If you need a different scheme (say, four time-of-day segments or different hours), you may need to modify those definitions in the SOWRL code.
* The week structure is assumed to be 7 days (with names Monday–Sunday) by default. If your game world has a different concept of week, you’d have to adjust the code that computes weekday and weekname.
* The coordinate system for x, y assumes a normal Ren’Py window (0.0–1.0 range). If you change your game resolution or how you interpret these (absolute pixels vs relative), keep consistency.
* Transition handling (like OkiMaster’s Start and scene functions) are part of the original framework but if present in SOWRL, they may expect you to use them as intended. For example, the game.Start() method (if it exists) likely assumes you’re using the label structure they set up. It’s fine if you don’t use it, but then ensure you manually call whatever needed initialization it does (like setting up screens). In documentation above, we assumed manual control.
* **Saving/Loading Edge Cases:** Generally, SOWRL data is saved, but a potential limitation is if a timer was set to execute at a time that passes while the game is not running (e.g., the player saved at 17:50 with a timer for 18:00, then loads that save and immediately jumps to 18:10). On load, Ren’Py restores state exactly as saved (17:50, with a pending timer). If your code then immediately sets time to 18:10, the timer will fire. That’s fine. Just be aware that saving mid-countdown preserves the countdown. There’s no known issue here, just something to test in your game logic to ensure that, say, important events can still trigger if the player loads right before them.
* **Lack of Direct Support for Complex Conditions:** SOWRL schedules (Puppeteer events) are primarily time/location based. They don’t by themselves handle conditions like “only if player did X” or “if weather is Y”. You can achieve those by combining with world state variables (e.g., check a flag in your screen before showing an NPC, or use the isActive toggle on events). But the limitation is there’s no built-in conditional scheduling beyond time and day. You must implement any additional logic in your game scripts.
* **Documentation and Community Support:** As SOWRL is a custom extension, it doesn’t have the same level of community support or extensive documentation as core Ren’Py features. The documentation here (and any provided by the author) is what you have. If you encounter a bug or need a new feature, you may have to dive into the SOWRL source code. The code is Python, so it’s accessible – but this means the onus is on the developer to maintain it for their game’s needs. Known issues should be listed in the repository; if not, be prepared to troubleshoot.

In conclusion, while you should be mindful of these limitations, most can be worked around or are unlikely to affect normal use cases. SOWRL is a powerful toolkit to simulate a living world in Ren’Py, and with careful usage it can bring depth and interactivity to your visual novel that would otherwise be very difficult to implement from scratch. Happy developing, and enjoy the richer worlds you can create with it!