SOWRL (Semi-Open World Ren'Py Library) Documentation

Introduction

SOWRL (Semi-Open World Ren'Py Library) is a custom extension for Ren'Py 7.4.5+ (last tested on 7.5) that adds support for nonlinear gameplay and semi-open-world features to visual novels. It provides a framework of classes and controllers to manage game state beyond the standard Ren'Py flow. With SOWRL, developers can create free-roaming environments with multiple locations, time-of-day cycles, character scheduling, interactive objects, dynamic weather, conditional triggers, shops with items, and storyline progression systems. All components are modular – you can use the global time system or quest system independently, or combine everything to build a complex open-world visual novel.

Key Features

- **World and Locations:** Define multiple locations and navigate between them freely, simulating an open world.
- **Characters and Objects:** Manage characters (with positions and schedules) and interactive or static objects in locations.
- **Time System:** Simulate time passing, day/night cycles, and schedule events or character routines based on time and day.
- **Weather Effects:** Add dynamic weather (rain, snow, etc.) that can appear in certain locations and times.
- **Triggers and Conditions:** Set up triggers that monitor game conditions and execute actions when conditions are met.
- **Story/Quest Management:** Handle non-linear story progression via "stories" (quest lines) with parts, hints, and location restrictions for each stage.
- **Shops and Inventory:** Create in-game shops and items that players can purchase, supporting inventory management.
- **Global Variables Storage:** Use a global key-value storage to keep track of custom game variables (e.g., stats or flags) outside of Ren'Py's default store.
- **Utility Functions:** Math helpers and convenience functions for common tasks (distance calculations, time formatting, etc.).
- **Scene Management:** Seamlessly transition between free-roam (open-world) mode and traditional Ren'Py scenes (visual novel dialogues) while preserving game state.

All these systems are designed to work together but can also be used in isolation. For example, you might use the time and schedule system to create timed events in a mostly linear game, or use location and trigger systems to add optional open-world segments. The following documentation describes each module, class, and function in SOWRL with usage examples.

Setup and Initialization

To use SOWRL in your Ren'Py project, you need to initialize the main controller and call the SOWRL game loop. Typically, this is done by creating an instance of the main class and calling a special Ren'Py label at game start:

- **Game Master Initialization:** SOWRL's core is the **OkiMaster** class. In a Ren'Py script (e.g., at the top of script.rpy), instantiate it as a default global variable. For example:

  ```
  default game = OkiMaster()
  ```

  This creates a global game object that will manage all subsystems. The OkiMaster constructor will internally create various controllers (for places, characters, time, etc.) as attributes of game .

- **Starting the Framework Loop:** SOWRL uses a custom label (commonly named okigamelabel) to run the main game loop. You should call this label when you want to enter the open-world mode (usually right after the main menu or prologue). For example:

  ```
  label start:
      # ... any pre-game setup ...
      call okigamelabel
      return
  ```

  The okigamelabel will handle entering the free-roaming world, updating time, processing scheduled events, and displaying the world UI. Make sure to include the SOWRL scripts in your project so this label and the classes are defined.

Note: In Ren'Py script syntax, you call Python functions using the $ prefix or in Python blocks. The examples in this documentation assume you're executing them as Ren'Py Python statements (e.g., $ some_controller.add(...)). Ensure the SOWRL library files are placed in your game directory so that classes like OkiMaster and the controllers are available.

Main Game Controller: OkiMaster and Global Store

OkiMaster (Game Master)

The OkiMaster class orchestrates all components of SOWRL. It is the central "game" object. When you created game = OkiMaster() , it automatically created instances of all the subsystems and made them available as attributes or global variables. Notably:

- game.store : An instance of **OkiStore** , which acts as a global database for game state.
- game.place_controller (aliased as global **places**): Manages locations.
- game.person_controller (aliased as **persons**): Manages characters.
- game.clicky_controller (aliased as **clickies**): Manages interactive objects.
- game.unclicky_controller (aliased as **unclickies**): Manages static (non-interactive) objects.

- `game.weather_controller` (aliased as **`weather`**): Manages weather effects.
- `game.trigger_controller` (aliased as **`triggers`**): Manages conditional triggers.
- `game.shop_controller` (aliased as **`shops`**): Manages shops.
- `game.product_controller` (aliased as **`products`**): Manages shop products/items.
- `game.story_controller` (aliased as **`story`** or **`stories`**): Manages storylines/quests.
- `game.date_controller` (aliased as **`date`** or integrated in time functions): Manages time, date, and scheduling.
- `game.machine_controller` (aliased as **`machine`**): Manages custom global variables.
- `game.extra` : An instance of **`OkiExtra`** with utility functions. *(You may also do `default extra = game.extra` for convenience.)*

The `OkiMaster` provides a few high-level methods to control scenes and visuals:

`Start(transition='fade')` – *Start the game's free-roam mode.*
This method is automatically called in the `okigamelabel` at the beginning of the game loop. It handles the transition from the main menu into the game world. You typically **do not** need to call this manually (it's set up in the SOWRL scripts). It accepts an optional Ren'Py transition for entering the game (default is a fade).

`Scene()` – *Enter a visual novel scene (hide free-roam UI).*
Use this when you want to switch from the open-world interface to a traditional Ren'Py scene (for example, when starting a dialogue or cutscene). Calling `game.Scene()` will: - Take a screenshot of the current world screen (so you have a background if needed). - Hide the free-roam interface (world map, HUD, etc.), pausing the open-world loop.

Example:

```
# The player clicks on an NPC, we enter a dialogue scene
$ game.Scene()
# (Now show dialogue or other scene contents)
```

`noScene(transition=None)` – *Exit a VN scene and return to free-roam.*
Call this when your dialogue or cutscene ends and you want to return to the open-world mode. This will restore the world UI (and optionally use a transition when returning to gameplay). You can pass a transition effect that will be applied when closing the scene.

Example:

```
# After the dialogue ends:
$ game.noScene(dissolve)
```

This would dissolve back into the free-roaming world, showing all the location screens and interactive elements again.

`ScreenshotScene()` – *Capture the current game screen as an image.*
This utility method takes a screenshot of the game's viewport and displays that image on the scene layer. SOWRL calls this internally when you use `Scene()` or `noScene()`, to preserve the last state of

the world behind menus or transitions. You typically don't call this directly unless you want to manually capture the screen for a special effect.

ColorMatrix(colormatrix=None, timepiece=0.5, layer='master', rst=True) – *Apply a color filter to the screen.*
This method lets you apply a color matrix filter to the game's rendering, which can be used to simulate effects like nighttime tint, sepia tone, etc. If `colormatrix` is `None`, it resets to the default (identity matrix with no color change). You can also pass one of Ren'Py's predefined matrix objects (e.g., `renpy.display.matrix.BrightnessMatrix(0.5)` for a dim effect) or a custom list of 16 values.

- `timepiece` : Duration in seconds for a smooth transition between the current filter and the new one (default 0.5s).
- `layer` : Which display layer to apply the filter to (default `'master'` which affects the master scene layer – basically the whole game scene).
- `rst` : If `True` , resets to identity matrix when done (you can usually leave this True to ensure a clean state after transitions).

Example – simulate dusk:

```
# Gradually apply a dim, orange tint over half a second
$ game.ColorMatrix(SepiaMatrix(tint="#704214"), timepiece=0.5)
```

(This assumes you have the Ren'Py color matrix objects available; you might need to import or prefix them appropriately.)

ShowImg(name, path, x, y) – *Show an arbitrary image at given coordinates in the world.*
This is a convenience for adding images to the current scene dynamically via code. It will display the image from `path` on the game scene. Parameters:
- `name` : a unique identifier name for the image (you can later manipulate or remove it using Ren'Py image functions if needed).
- `path` : the file path to the image to display (e.g., `"images/props/boat.png"`).
- `x, y` : The position on screen to place the image. Coordinates can be absolute pixels or relative (0.0–1.0) if using fractions. (Do not mix types for x and y in one call.)

Example:

```
# Display a prop image on the scene at 400px, 300px
$ game.ShowImg("boat", "images/props/boat.png", 400, 300)
```

refreshClick() – *Reset the "active click" memory.*
Internally, the framework might track which object is currently being interacted with (for example, to highlight it or prevent multiple overlaps). `refreshClick()` clears the record of any object being clicked or hovered. If you notice interactive objects not responding because the game still thinks you're interacting with something, you can call this to reset the state.

refreshScreen() – *Force-refresh the UI screens.*
If you have changed some game state via code (such as toggling an object's visibility or moving a

character) and the change isn't reflected on-screen, `game.refreshScreen()` will manually tell Ren'Py to re-render the screens. Use this after significant manual updates to ensure the display catches up.

`OkiStore` (Global Data Store)

The **OkiStore** class (accessible as `game.store`) functions as a simple in-memory database for the game world. It holds data structures and default settings used by other controllers. Key roles of `OkiStore` include:

- Storing dictionaries of all entities (places, characters, items, etc.). Each controller (places, persons, etc.) writes to `game.store` under the hood. For example, when you add a new place, its data dictionary is saved in `game.store` (you typically don't manipulate `OkiStore` directly; you use the controllers).
- Defining default paths and parameters. For instance, `OkiStore` might contain base directory paths for images of places, characters, weather effects, etc., and configuration like what hours count as morning/day/night (used for day-part image selection). These defaults are set when `OkiMaster` is initialized.

You usually will not call methods on `game.store` directly. Instead, use the provided controllers (which interact with `game.store` as needed). However, you can read and write generic attributes in `game.store` for simple global flags if you want. For example, `game.store.money = 100` could be used to track player currency (as seen in triggers examples below). It's essentially a Python object you can attach attributes to. The library uses it for structured data, but you can also utilize it for custom global data if careful not to overwrite keys used by SOWRL.

Example:

```
# Accessing the player's current location stored in game.store
$ current_loc = game.store.location
# Setting a custom global flag
$ game.store.found_secret_room = True
```

(Note: `game.store.location` is a special variable typically used to track the player's current location key. The `places` controller has helper methods to manipulate it, as described next.)

Location Management: `OkiPlaceController` (Locations)

Locations (places) represent distinct areas or screens in your game world (rooms, regions, etc.). The **`OkiPlaceController`** manages all location entries. By default, an instance is accessible as the global **`places`** object (which is `game.place_controller` internally).

Each location is stored as a dictionary with several properties. When you add a location, SOWRL keeps track of things like the location's name and an identifier key. Locations can also have an image or background associated, and the framework can handle different images for different times of day if configured.

Location Properties: (each location's data dictionary contains)
- **`name`** – The display name of the location (e.g. `"Bedroom"`).

- **cfname** – The unique key identifier for the location (e.g. "bedroom"). This is used as the reference in code and for image lookup.
- **arts** – (optional) An alternate key for the location's art assets. If provided, the system will use this string instead of cfname when looking up location background images. (Use this if your image folder name differs from the location key.)
- **location** – *(For locations, this may not be used or could serve a special purpose. Generally, location fields appear in other entities to indicate where they are. In a location's own dict, this might remain empty or serve advanced purposes like linking parent locations. Typically you won't use this field.)*
- **x**, **y** – Coordinates for the location's default position in a world map screen (if applicable). These might represent where a clickable hotspot for this location appears on an overworld map. If your game doesn't have a world map screen with clickable location icons, these might be unused.
- **isActive** – Boolean (default True). If False, the location is considered "inactive" or not present in the game world – it might be hidden or inaccessible. Inactive locations can be ignored by travel functions or the UI. Set to False to remove a location without deleting its data.
- **image** – The current image path used for this location's background or icon. Initially None or a default path based on the location key. Usually the system will set this automatically using the cfname and current time of day (if day-part images are used).
- **text** – Extra text field (for description or notes). Not used by the framework logic directly, but you can store a description or narrative for the location if needed (e.g., to show on a tooltip).
- **status** – A string indicating the location's current state, if the scheduling system is used on this location. For example, if you have scheduled events that move or change the location's environment, status might hold something like "unavailable" or a scenario name. By default it's an empty string unless events update it.
- **use_daypart** – Boolean (default True). Determines if the framework should automatically choose the location's image based on the current part of day. If True, and you have images named or organized by day parts (morning/afternoon/evening/night), the system will pick the appropriate variant. If False, it will use the base image regardless of time of day.

Managing Locations: Use the places controller's methods to create and manipulate locations.

- **places.add(name, cfname, arts=None)** – Add a new location.
- *name*: Human-readable name of the location (appears in UI or prints).
- *cfname*: Key identifier (no spaces, typically lowercase). Also assumed to be the folder or basename for images.

- *arts*: (optional) Alternate key for art assets if different from cfname. If you have a location where images are stored under a different name, provide that here. Otherwise, omit or use None.
  Example:

  ```
  $ places.add("Beach", "beach")
  ```

  This creates a new location called "Beach" with key "beach". The system will expect to find its background images under a path using "beach" (unless overridden). Coordinates will default to (0,0) or a neutral position unless you set them via events or directly.

- **places.get(cfname, keyname=None)** – Retrieve information about a location.

- *cfname*: The key of the location you want to query.

• *keyname*: (optional) If provided, returns only that specific property of the location's dict. If omitted, returns the full dictionary for that location. If the location doesn't exist, returns `False` .
Example:

```
# Get the full dictionary for "beach"
$ beach_data = places.get("beach")
# Or get a specific field, e.g., is it active?
$ is_beach_open = places.get("beach", "isActive")
```

• **`places.set(cfname, keyname, val)`** – Modify a property of a location.

• *cfname*: The key of the location to modify.
• *keyname*: The property name to change (e.g., `"isActive"` , `"name"` , `"image"` , etc.).

• *val*: The new value to assign to that property.
Example:

```
# Deactivate the beach location temporarily
$ places.set("beach", "isActive", False)
# Change the display name of "beach" to "Sunny Beach"
$ places.set("beach", "name", "Sunny Beach")
```

• **`places.remove(cfname)`** – Remove a location from the database.
This deletes the location's entry entirely. Use with caution, as any references to it (character locations, etc.) might become invalid. In many cases, setting `isActive` `False` is safer for temporary removal. If `cfname` is not given or is `None` , this could remove *all* locations (depending on implementation), so always specify the key.
Example:

```
$ places.remove("beach")
```

• **`places.get_location()`** – Get the player's current location key.
This returns the `cfname` of the location where the player currently is. Under the hood, it likely reads `game.store.location` . It's useful for triggers or logic that need to check "where am I now?".
Example:

```
$ current_place = places.get_location()
if current_place == "hospital":
    "You are standing in the hospital lobby."
```

• **`places.set_location(location_key)`** – Change the current location.
This method moves the player to the specified location. It will update the internal current location tracker and typically trigger the world UI to switch to that place's background and

objects. After calling this, the new location's screen and entities will become active.
Example:

```
# Move the player to the hospital
$ places.set_location("hospital")
```

You might call this in response to a click on a travel icon or as part of a story event. For instance, an interactive object (like a door) could have an action that calls `places.set_location("hospital")` to simulate moving the player to the hospital location.

Locations are foundational – other systems (characters, clickables, weather) often reference location keys to determine where they appear. Once you have defined your locations with `places.add()`, you can start populating them with characters and objects.

Character Management: `OkiPersonController` (Characters)

The **`OkiPersonController`** (global **`persons`**) manages all characters in your game's world. These are not the same as Ren'Py dialogue Character objects; rather, they represent entities in the open world (like NPCs on the screen with positions). You can still use Ren'Py's Character for dialogue name and voice, but OkiPersonController handles where the character is, their sprite, and their schedule in the world.

Each character is stored as a dictionary with properties: name, surname, unique id, current location, coordinates, etc., as well as some interaction settings.

Character Properties:
- **`name`** – First name or given name of the character (for display).
- **`surname`** – Last name or family name of the character (optional, for display or uniqueness).
- **`cfname`** – Unique key identifier for the character (e.g., `"jane"`). This is used as the reference in code and for image file naming. Sprites for the character are expected under this key (unless `arts` is used).
- **`arts`** – Sprite assets key (if different from `cfname`). If provided, the system will use this for locating the character's image files. For example, you might have a character key "villager1" but share sprites with another, so you could set `arts` to use a common folder. Usually leave this None to use `cfname`.
- **`location`** – The key (string) of the location where the character currently is, or a list of location keys if the character can appear in multiple places. Typically this is a single location string indicating where to render them now. (The schedule system can automatically change this as time progresses.)
- **`x`**, **`y`** – Coordinates of the character on the location's screen (in pixels relative to a default window size). This determines where the character's sprite is drawn in the scene. (0,0 would be top-left; as Ren'Py positions, the default coordinate system might be 0 at left for x and 0 at top for y). You should set this to place the character at the desired spot in the background.
- **`isActive`** – Boolean (default True). If False, the character is not considered present in the world. They will be ignored and typically not rendered. (This could be used if a character is "out of town" or not yet introduced, without removing them entirely.)
- **`scale`**, **`hue`**, **`brightness`**, **`saturation`** – Visual effects for when the player hovers over or interacts with the character's sprite. These default to `1.0` (no scale change), `3.0` (hue shift amount), `0.15` (brightness increase), `1.4` (saturation). They define how the character's image will visually respond on hover: e.g., slightly bigger, highlighted color. You can tweak these if needed to alter the highlight effect on characters.

- **action** – A string naming a Ren'Py label or code snippet to run when this character is clicked. By default, if not set, the framework will assume a label name of the form "person_<cfname>" . For example, a character with cfname = "jane" will, by default, call label person_jane when clicked (if such a label exists). You can override this by setting action to a different label or even a direct Python call string. If you want clicking the character to open dialogue, you might write a custom label to handle it.
- **image** – The current image path being used for the character's sprite. Initially None or auto-determined. The framework will automatically choose an image based on the character's cfname (or arts) and their current status and time of day. For example, if Jane is in "idle" status at night, it might look for an image like images/jane/idle_night.png . This field updates as events or time change.
- **text** – Extra text field for any notes or description related to the character. Not used by the system logic, but can store additional info (perhaps biography or state that's displayed somewhere).
- **status** – A string representing the character's current status or activity, especially if using the scheduling system. For example, status might be "sleeping" or "working" , which could correspond to different sprite sets or positions. Scheduled events can automatically update a character's status at certain times, and the sprite image can change accordingly if you have images named after statuses.
- **use_daypart** – Boolean (default True). Similar to locations, this flag tells the system whether to automatically adapt the character's sprite to the current time of day. If True, the framework looks for different images for morning/day/evening/night (assuming your file structure or naming differentiates them). If False, it will use one image regardless of time.

Managing Characters: Use the persons controller methods:

- **persons.add(name, surname, cfname, location, x, y, arts=None)** – Create a new character in the world.
- *name*: First name of the character (string).
- *surname*: Last name of the character (string).
- *cfname*: Unique key for the character (used in code and file names).
- *location*: The initial location key where this character starts. This can be a single location or a list of possible locations. Usually it's one location (e.g., "room").
- *x, y*: Initial coordinates where the character's sprite will appear in the location.
- *arts*: (optional) Alternate art key if sprite assets use a different name.
  Example:

```
# Add a character Jane Doe at position (500, 500) in the "room" location
$ persons.add("Jane", "Doe", "jane", "room", 500, 500)
```

  This will register Jane in the world. By default she is active and her action on click will look for a label person_jane (unless changed). Her sprite should be placed at x=500, y=500 in the "room" background.

- **persons.get(cfname, keyname=None)** – Retrieve a character's data.
- *cfname*: The character's key.
- *keyname*: (optional) If provided, returns only that property from the character's data (e.g., "location" or "status"). If omitted, returns the full dictionary for the character. Returns

`False` if the character is not found.
Example:

```
$ jane_data = persons.get("jane")          # full info dictionary
$ jane_location = persons.get("jane", "location")
$ jane_status = persons.get("jane", "status")
```

- **`persons.set(cfname, keyname, val)`** – Modify a character's property.

- *cfname*: Character key.
- *keyname*: Field to change (e.g., `"x"`, `"location"`, `"isActive"`, etc.).

- *val*: New value.
Example:

```
# Move Jane to a new location "park"
$ persons.set("jane", "location", "park")
# Temporarily hide Jane from the world
$ persons.set("jane", "isActive", False)
# Change Jane's action to a custom label
$ persons.set("jane", "action", "meet_jane_event")
```

- **`persons.get_all_from_location(location='')`** – Get all characters currently at a location.

- *location*: The location key to filter by. Returns a list of character dictionaries for those whose `location` matches. If `location` is an empty string or not provided, it may return all characters (or none). Typically, use it with a specific location to find out who is there.
Example:

```
$ people_in_room = persons.get_all_from_location("room")
# This might return [ {Jane's dict}, {Billy's dict}, ... ] if Jane and
Billy are in "room".
```

- **`persons.remove(cfname)`** – Remove a character from the database.
Completely deletes the character's entry. After this, the character will no longer be tracked or appear. Use only if you permanently want to destroy that character's data (for example, if a character is entirely written out of the story). In most cases, setting `isActive` `False` or moving them offscreen is enough.
Example:

```
$ persons.remove("jane")
```

Characters often have scheduled routines or triggers associated with them. The scheduling system (explained later) can automatically update their location and status based on time of day, which creates life-like NPC schedules in an open world. You can also script movements or changes manually using the above methods in events or dialogues.

Using Characters in Scenes: Even though `persons.add()` creates world characters, you can still leverage Ren'Py's dialogue system. For example, you might have a Ren'Py Character named `define J = Character("Jane")` for dialogues. Clicking the Jane sprite in the world could trigger a label that does `game.Scene()`, then uses `J "Some dialogue..."` for a conversation, and finally calls `game.noScene()` to return to the world. The open-world character data (position, etc.) remains in `persons` while the conversation plays out. This way SOWRL and Ren'Py's narrative system work together.

Interactive Objects: `OkiClickyController` (Clickables)

Interactive objects, or "clickies", are in-world objects the player can click on (apart from characters). These might be items, doors, points of interest, etc., that trigger some action when interacted with. The **`OkiClickyController`** (global **`clickies`**) manages these entities.

A clicky is similar to a character in that it has a position and can be clicked, but it generally doesn't have a name or surname. Instead, it's identified just by a key and perhaps an optional display name. Clickies can also have special visual effects on hover and an action on click.

Interactive Object Properties:

- **`cfname`** – The unique key for the object (e.g., `"door_to_shop"`). This is used as its ID and for image lookup (unless overridden by `arts`).
- **`arts`** – Sprite asset key for the object. If None, it will use `cfname` to find its images. If you have multiple objects sharing sprites, you can set their `arts` to the same value.
- **`location`** – The location key or list of location keys where this object exists. You can place a clicky in multiple locations if needed (e.g., a generic item appearing in different places). Typically it's one location string.
- **`x`**, **`y`** – Coordinates of the object within its location screen (in pixels). Determines where the object's image is drawn.
- **`isActive`** – Boolean (default True). If False, the object is not present/interactive in the world. You can toggle this to show/hide objects without removing them entirely.
- **`scale`**, **`hue`**, **`brightness`**, **`saturation`** – Hover effects (same idea as for characters). Defaults: 1.0, 3.0, 0.15, 1.4 respectively. These control how the object appears when the player hovers the cursor: it might slightly enlarge or glow, indicating it can be clicked.
- **`action`** – A string for the action to perform on click. By default, if you don't set this, the framework will assume a label named `"clicky_<cfname>"`. For example, an object with `cfname="door_to_shop"` will look for `label clicky_door_to_shop` when clicked. You can override `action` with a different label name or even inline Python code (like `"places.set_location('shop')"` to immediately change location).
- **`image`** – Current image path of the object. Initially None or auto-set. The system will determine which image file to use based on `cfname` / `arts`, `status`, and daypart. For example, if an object's `cfname` is "chest" and status is "open", it may try `images/chest/open.png`.
- **`text`** – Extra text/description field for the object. Not used by the engine logic, but you might use it for an on-hover tooltip or description of the object.
- **`name`** – An optional display name for the object. By default, it's set equal to `cfname` (so you don't have to provide one). If you want a nicer name for UI purposes, you can set this after creating the object

(e.g., set `"name"` property to "Mysterious Chest").
- **`status`** – A string representing the object's current state if using schedules (like characters). For instance, an object could have status `"broken"` or `"active"` which could correspond to different images or whether it's clickable.
- **`use_daypart`** – Boolean (default True). If True, the object's image can change with time of day. For example, streetlights might have different sprites at night vs day. If False, the same image is used all the time.

Managing Interactive Objects (Clickies):

- **`clickies.add(cfname, location, x, y, arts=None)`** – Add a new interactive object to the world.
- *cfname*: The key/ID for the object. (Also used for default action label and image filenames.)
- *location*: The location key (or list of keys) where this object exists.
- *x, y*: Coordinates on the screen where the object will appear.
- *arts*: (optional) Alternate key for the object's images if not using `cfname`.
  Example:

```
# Add an info kiosk object present in three locations (with a sprite at
200,100 in each)
$ clickies.add("info_kiosk", ["room", "hospital", "beach"], 200, 100)
```

This creates a clickable object "info_kiosk" that appears in the Room, Hospital, and Beach locations at coordinates (200,100) on each. By default, clicking it will attempt to call label `clicky_info_kiosk` unless you specify a different `action`.

- **`clickies.get(cfname, keyname=None)`** – Get data about an interactive object.

- *cfname*: The object's key.

- *keyname*: (optional) Specific property to retrieve. If omitted, returns the full dictionary for that object, or `False` if not found.
  Example:

```
$ door_data = clickies.get("door_to_shop")
$ door_status = clickies.get("door_to_shop", "status")
```

- **`clickies.set(cfname, keyname, val)`** – Change a property of an interactive object.

- *cfname*: The object's key.
- *keyname*: Property to set (e.g., `"isActive"`, `"action"`, `"status"`).

- *val*: New value.
  Example:

```
# Change what happens when the player clicks the info kiosk
$ clickies.set("info_kiosk", "action", "show_info_screen")
```

```
# Disable the info kiosk at night by setting it inactive (perhaps via a
trigger)
$ clickies.set("info_kiosk", "isActive", False)
```

- **`clickies.get_all_from_location(location='')`** – Get all interactive objects at a given location.

- *location*: The location key to filter by. Returns a list of object dictionaries present there. (If a clicky's `location` list includes that key, it counts.)
  Example:

```
$ objects_in_room = clickies.get_all_from_location("room")
```

  If "room" had an info_kiosk and perhaps a door object, you'd get their data here.

- **`clickies.remove(cfname)`** – Remove an interactive object from the world.
  This deletes its data entry entirely. After removal, it will no longer appear. Use it if an object is permanently removed or consumed.
  Example:

```
$ clickies.remove("info_kiosk")
```

Using Clickies: Typically, you place clickies for things like exits and interactive props. For example, you might add a clicky for a door that leads from the hospital to the shop. You'd then set its `action` to some code or label that changes the location. Using the example above:

```
# Add a door clicky in "hospital" location at (300,100)
$ clickies.add("door_to_market", "hospital", 300, 100)
# Set the door's action to actually move the player
$ clickies.set("door_to_market", "action", "places.set_location('market')")
```

Now, when the player clicks the door in the hospital, the `places.set_location('market')` code will run, transporting them to the market location. You could also have the `action` call a label if you need more complex logic (like playing a sound, or checking if the door is unlocked).

The framework will automatically handle highlighting the object when hovered (using the scale, hue, etc. parameters) to indicate it can be clicked, and will call the specified action when clicked.

Static Objects: `OkiUnclickyController` (Unclickables)

Static objects, or "unclickies", are similar to clickies but purely decorative or non-interactive. They appear in the world but do not respond to clicks. Use unclickies for background elements that might move or change but aren't meant to be clicked by the player (for example, a bird flying in the sky, or a signpost that's just part of the scenery).

The **OkiUnclickyController** (global **unclickies**) manages these. Unclickies have almost the same data structure as clickies, minus the action handling.

Static Object Properties:
- **cfname** – Unique key for the object (e.g., "tree01").
- **arts** – Asset key for images (if different from cfname).
- **location** – Location key or list of keys where this object appears.
- **x**, **y** – Coordinates on the screen for the object's image.
- **isActive** – Boolean (default True). If False, the object won't be displayed.
- **scale**, **hue**, **brightness**, **saturation** – (These might not be heavily used for unclickies since they're not interactive, but if the framework shares code with clickies, they could exist. Typically, you won't see hover effects on unclickies because they don't get hovered for interaction.)
- **action** – (Usually not applicable for unclicky objects; they have no click action. This might be None or simply unused.)
- **image** – Current image path used for this object's sprite (auto-managed similar to clickies).
- **text** – Extra text/description (if needed for reference).
- **name** – Display name (mostly unused for static objects, but available for consistency; often just same as cfname).
- **status** – State name if used in scheduling (e.g., a fountain might have status "on"/"off").
- **use_daypart** – Boolean (default True) to toggle using time-of-day specific imagery.

Managing Static Objects:

- **unclickies.add(cfname, location, x, y, arts=None)** – Add a new static object. Parameters mirror clickies.add except there's no interactive aspect.
  Example:

  ```
  # Add a decorative statue at (500,500) in the "park" location
  $ unclickies.add("park_statue", "park", 500, 500)
  ```

- **unclickies.get(cfname, keyname=None)** – Retrieve a static object's data (same usage as clickies.get).

- **unclickies.set(cfname, keyname, val)** – Modify a static object's property. You might use this to change its isActive or maybe swap an image via status.
- **unclickies.get_all_from_location(location)** – List all static objects in a given location.
- **unclickies.remove(cfname)** – Remove a static object entirely.

Example:

```
# Place a non-clickable bird that appears in the room at (100,50)
$ unclickies.add("bird", "room", 100, 50)
# Later, hide the bird (maybe it flew away)
$ unclickies.set("bird", "isActive", False)
```

Unclickies are optional, but they're useful to enrich the environment. They can also be controlled by the schedule system or triggers (for instance, perhaps at noon each day, you use a trigger to show or hide a

certain unclicky). Since they share a lot of structure with clickies, you can schedule them to appear/disappear or change status just like any entity.

Weather System: `OkiWeatherController` (Weather Effects)

The **`OkiWeatherController`** (global **`weather`**) allows you to create and control weather effects in your game, such as rain, snow, falling leaves, etc. Weather effects are essentially particle systems that can be active or inactive depending on conditions, and tied to specific locations.

Each weather effect is stored with properties defining how it behaves (like intensity of particles, direction, etc.). By default, when you add a weather effect, it is **not active** – you must activate it to see it.

Weather Effect Properties:
- **`name`** – An optional name/description of the weather effect (e.g., `"Light Rain"`). This is mostly for your reference or UI; the system doesn't require it except to store as extra info.
- **`cfname`** – Unique key for the weather effect (e.g., `"rain"` or `"snow"`). This key is used to identify the effect in code and also for finding its particle image assets (unless `arts` is specified).
- **`arts`** – The folder/name for particle sprites. If None, it will look for images using `cfname`. If you want to use a different sprite sheet or particle images for this effect, provide an alternate name here. For example, you might have multiple rain sprite variations – you could use `arts="heavy_rain"` to use a different set.
- **`intensity`** – Number of particles or elements in the effect. For rain, this might be the number of raindrop sprites generated; for snow, number of snowflakes. Default is 10. Higher values mean denser effect (more drops or flakes).
- **`xforce`** – Horizontal force applied to particles (wind effect). Positive or negative values push particles to the right or left. For example, `10` might mean a breeze blowing particles to the right; `-10` would blow to the left. `0` means no horizontal movement.
- **`yforce`** – Vertical force (gravity or upward force). Positive values push particles down (like gravity for rain/snow), negative would push up (like something rising). For typical weather like rain, you'd use a positive number to fall downward.
- **`rotate`** – Rotation setting for particles. If set to a number (in degrees), particles will spawn with that fixed angle. If set to `True`, the system will automatically rotate each particle based on its movement direction (so raindrops might angle in the wind). If `False`, no rotation (or 0 degrees). Default is `0` (which likely is treated as no rotation unless `True` is used).
- **`location`** – Which location(s) this weather effect applies to. (Not in the initial list above, but the usage example suggests you can assign weather to specific locations via `weather.set`). This is likely stored similarly to objects: it can be a string or list of location keys where the weather occurs. If the player is in one of those locations and the effect is active, it will be visible.
- **`isActive`** – Boolean indicating if the effect is currently active (visible). By default, after adding, this will be False. You can either manually set it True or use the `weather.show()` method to activate it. When True, and if the player's current location is one in the effect's location list, the particles will be rendered. If the player leaves those locations or `isActive` is False, the effect will not show.
- **`image`** – The current image (or sprite) path used for the particles (if applicable). Possibly the system picks the appropriate particle image based on `arts` and daypart. Usually, you'll have a sprite or image sequence for the particles (like a PNG of a raindrop or snowflake).
- **`text`** – Extra field for any notes (not used in logic).
- **`use_daypart`** – Boolean (default might be False for weather as per doc snippet). This would determine if the weather effect's sprites change by time of day. For example, maybe at night you use slightly darker raindrop images. Often this isn't needed, so it might default to False.

Adding and Controlling Weather: Use the `weather` controller's methods:

- **`weather.add(cfname, intensity=10, xforce=10, yforce=50, rot=0, arts=None)`** – Create a new weather effect definition.
- *cfname*: Key for the weather (and default sprite name).
- *intensity*: Number of particles (default 10).
- *xforce*: Horizontal wind force (default 10).
- *yforce*: Vertical force (default 50, meaning particles fall downward fairly quickly).
- *rot*: Rotation mode (0 by default, use `True` for auto-rotate with direction).
- *arts*: Alternate sprite key, if not using `cfname`.
  Example:

```
# Add a snow effect with 20 flakes, slight wind to left, auto-rotating
flakes
$ weather.add("snow", intensity=20, xforce=-5, yforce=30, rot=True,
arts="snowflake")
```

This defines a snow weather named "snow" that will use sprites from the "snowflake" set. It has 20 snowflakes, wind pushing left (xforce -5), gravity down (yforce 30), and each flake rotates according to its movement.

- **`weather.get(cfname, keyname=None)`** – Get information about a weather effect.
- *cfname*: Weather effect key.
- *keyname*: (optional) Specific field to retrieve (e.g., `"isActive"` or `"intensity"`). If omitted, returns the full dictionary for that weather effect, or False if not found.
  Example:

```
$ snow_info = weather.get("snow")
$ is_snowing = weather.get("snow", "isActive")
$ snow_particles = weather.get("snow", "intensity")
```

- **`weather.set(cfname, keyname, val)`** – Modify a weather effect's property.
- *cfname*: Weather key.
- *keyname*: Property to set (e.g., `"isActive"`, `"location"`, `"intensity"`).
- *val*: New value.
  Example:

```
# Assign the snow effect to the "park" and "forest" locations
$ weather.set("snow", "location", ["park", "forest"])
# Activate the snow effect (start snowing)
$ weather.set("snow", "isActive", True)
```

```
# Change intensity on the fly (maybe a blizzard starts)
$ weather.set("snow", "intensity", 50)
```

- **`weather.remove(cfname)`** – Remove a weather effect definition entirely.
  If you no longer need a weather effect (perhaps it was a one-time event), you can remove it. This will stop it and clear its data.
  Example:

```
$ weather.remove("snow")
```

In addition, the weather controller provides convenience methods to show or hide effects with optional transitions (like fade-in/out of particles):

- **`weather.show(cfname, dp=True, transition=None)`** – Manually display a weather effect immediately.
- *cfname*: The weather effect key to show.
- *dp*: Day-part auto-detect (True by default). If True, the effect may choose a variant of its sprite for the current time of day (if applicable). If False, it will not consider daypart for sprite selection.
- *transition*: (optional) a Ren'Py transition to use when the weather appears (e.g., fade). If provided, the effect might fade in using that transition.
  Example:

```
# Start rain effect with a fade-in transition
$ weather.show("rain", transition=dissolve)
```

  This will activate the "rain" effect and dissolve it in. Essentially, it sets `isActive` to True for that effect and ensures it's rendered.

- **`weather.hide(cfname, transition=None)`** – Hide a weather effect.

- *cfname*: Weather effect key to hide.
- *transition*: (optional) transition for fading it out.
  Example:

```
# Stop the rain with a fade-out
$ weather.hide("rain", transition=fade)
```

  This will gradually remove the rain effect using a fade transition and set it inactive.

Usage Tips:
Define weather effects early (e.g., at init, using `weather.add()` for each type you plan on). Assign each effect to relevant locations via `weather.set(..., "location", ...)` . Then you can control when they're active either by triggers or story events. For example, you might have a trigger that turns on rain at 6 PM each day (using the time system to detect 6 PM and calling `weather.set("rain", "isActive", True)`).

In our earlier example usage:

```
$ weather.add("test1", 5, 100, 100, True)          # adds a weather effect
'test1'
$ weather.add("test2", 5, -100, -100, True, "snowflake")
$ weather.set("test1", "location", ["room", "hospital"])
$ weather.set("test1", "arts", "rain")             # use 'rain' sprite set
for test1
$ weather.set("test2", "location", "room")
$ weather.set("test1", "isActive", True)
$ weather.set("test2", "isActive", True)
```

This example (from the original notes) creates two test weather effects and activates them. `test1` uses arts "rain" and appears in room and hospital, `test2` uses arts "snowflake" and appears in room. Both are activated (so if the player is in room or hospital, they'd see rain; in room specifically, maybe both rain and snowflake effect, which might be a mixed example).

Time and Scheduling: `OkiDateController` (Game Time & Event Scheduler)

One of SOWRL's core features is a game clock and scheduling system, handled by the **`OkiDateController`** (accessible via `game.date_controller`, often used indirectly through time functions). This system keeps track of in-game time (hours, minutes, days) and allows you to fast-forward time, schedule future events (timers), and define routines for characters or other entities that happen at certain times.

Game Time Tracking

The game's current time is typically stored in `game.store` (for example, as hours and minutes, or a combined time string). The date controller provides methods to manipulate time:

- **`date.set_time(h, m)`** – Set the current in-game time to hour `h` and minute `m`.
  For example, `date.set_time(8, 30)` would set time to 08:30. You might call this at game start or after a jump in time. Changing time can immediately trigger time-based events (more on that below).

- **`date.add_minutes(m)`**, **`date.add_hours(h)`**, **`date.add_days(d)`** – Increment the current time by a given amount.
  These functions advance the clock conveniently. They handle overflow (e.g., adding 120 minutes will increment hours/days appropriately).
  Example:

  ```
  $ game.date_controller.add_hours(1)    # advance time by 1 hour
  $ game.date_controller.add_minutes(30) # advance time by 30 minutes
  ```

  If it was 8:30, after add_hours(1) it becomes 9:30; add_minutes(30) would then make it 10:00.

- **`date.time_to(target_time, days=0)`** – Fast-forward to a specific time of day (with optional day offset) and return the number of in-game minutes that passed.

- *target_time*: a string in "HH:MM" format that you want to skip to (e.g., `"18:00"` for 6:00 PM).

- *days*: how many days forward to jump as well. Default 0 (same day). If days=1 and target_time is earlier than current time, it effectively goes to that time on the next day.
  This method will advance the clock to the specified time (and additional days if specified). It calculates how many minutes elapsed during that skip (which it returns). Also, importantly, if you have scheduled timers (see below), those that fall between the start and end time might get executed or marked appropriately. The method ensures that if you fast-forward past a timer, that timer will fire after the skip (unless it was supposed to happen after the skip window).
  Example:

  ```
  # Skip ahead to 22:00 (10 PM) tonight
  $ minutes_passed = game.date_controller.time_to("22:00")
  ```

  If it was 18:00, this moves to 22:00 (4 hours later) and `minutes_passed` would be 240. If any timers were set for times between 18:00 and 22:00, they would trigger appropriately during this call.

- **Time Interval Checks:** There are helper functions to check if a time is within an interval:

- **`date.is_in_interval(time, interval_str)`** – Returns True if a given time (format "HH:MM") lies within a time interval (format `"HH:MM-HH:MM"`). Example: `date.is_in_interval("13:30", "12:00-14:00")` returns True.
- **`date.is_minutes_between(dm, dms, dme)`** – A lower-level check: given a minute count `dm` and a start minute `dms` and end minute `dme`, returns True if `dm` is in [dms, dme]. (This is used internally for interval checks after converting times to total minutes of day.)

These checks can be useful if you manually need to see if the current time is within a certain range (though triggers and schedule can automate a lot of that).

The date controller also defines constants for day length or allowed times (by default likely 24h clock, 0–23 hours, etc., defined in OkiStore).

Timer Events (Delayed Actions)

SOWRL allows scheduling one-time code executions in the future via “timers”:

- **`date.do_in(cfname, do, timepiece=0, sort='minutes')`** – Schedule a one-time event to occur after a certain amount of time passes.
- *cfname*: A unique name for this timer (string). If you use a name that already exists, it will overwrite the old timer with this new one.
- *do*: A string of Python code to execute when the time is up. (You can call a Ren’Py label by using a Python call like `renpy.call("some_label")` or trigger dialogue with `renpy.say` in this code string, etc. Make sure the code is valid Python in Ren’Py’s context.)
- *timepiece*: The time value for the delay, interpreted according to *sort*. For example, if sort is 'minutes', timepiece is the number of minutes from now to wait.

- *sort*: The unit of the delay – `'minutes'` , `'hours'` , or `'days'` . (Default is 'minutes'.)
  Effect: The timer starts counting immediately. When the specified time elapses in the game's clock, the code in *do* will run. If the game is fast-forwarded beyond the timer's target time using `time_to` , the timer will fire immediately after the time jump (because the scheduled time was surpassed).
  Example:

```
# In 60 in-game minutes, give the player a reminder
$ game.date_controller.do_in("reminder", "renpy.notify('An hour has
passed!')", 60, sort='minutes')
```

  This sets a timer named "reminder" to call Ren'Py's notify function after 60 minutes of game time.

- `date.do_at(cfname, do, when, days=0)` – Schedule a one-time event at an exact future time.

- *cfname*: Unique name for the timer (like above).
- *do*: Python code string to execute.
- *when*: Target time as a string "HH:MM" when it should execute. This can be today or a future day.
- *days*: How many days in the future to schedule it (0 means today at that time if that time is still upcoming; if the time has already passed today, 0 days would schedule for next day same time? It's safer to use days=1 if you mean tomorrow).
  Effect: Creates a timer to run code at a specific clock time (and optional day offset). If you skip time or time naturally reaches that moment, the code triggers.
  Example:

```
# Schedule store closing event at 21:00 (9 PM) today
$ game.date_controller.do_at("store_close", "shops.open('store');
renpy.say(narrator, 'The store is now closed.')", "21:00", days=0)
```

  This example (for demonstration) at 9 PM will execute the code to open a screen or something and narrate that the store closed (just as an illustration; `shops.open` wouldn't normally be used to close, but we could imagine code to handle closure).

Both `do_in` and `do_at` create an entry in a timer list. The framework checks these timers regularly (likely each game loop tick or each time increment) and runs them when due. Timers, once executed, are either removed or marked so they don't repeat (all timers created with these are one-shot by default).

You can manage timers with:

- `date.set_timetask(cfname, keyname, value)` – Edit a scheduled timer's property.
  If you need to adjust a timer (e.g., change its `do` code or postpone it), you can modify it.
  Properties of a timer include:
- `'cfname'` – the name
- `'typetask'` – whether it was created by do_in or do_at
- `'do'` – the code to run
- `'executed'` – bool if it has executed (True once done, such timers may be cleared)

- `'day'` – the target day for do_at tasks
- `'dayMinutes'` – the target minute of day to execute (for do_at tasks)
- `'text'` – a text field for notes
  In practice, you might rarely use this, but it's there if needed. For example, you could disable a timer by setting its `executed` to True prematurely or change its `dayMinutes`.
  Example: *(Hypothetical)*

  ```
  # Suppose we want to delay the reminder by another 30 minutes
  $ game.date_controller.set_timetask("reminder", "dayMinutes",
  game.date_controller.get("reminder", "dayMinutes") + 30)
  ```

  (This assumes a `get` method exists to fetch timer info; if not, one would access internal storage. Usually simpler: remove and create a new timer.)

- **`date.remove_timetask(cfname)`** – Cancel a scheduled timer by name.
  Example:

  ```
  $ game.date_controller.remove_timetask("reminder")
  ```

  This would cancel the "reminder" timer if it hasn't executed yet, so it will not trigger.

Timer Properties Recap: Each timer's data might look like:

```
{
   'cfname': "reminder",
   'typetask': "do_in",                    # or "do_at"
   'do': "renpy.notify('An hour has passed!')",
   'executed': False,                      # becomes True after running
   'day': 0,                               # target day offset (for do_at), 0 means
current day
   'dayMinutes': 1234,                     # the absolute minute of the day to execute
(e.g., 20:34 is 1234 if dayMinutes counts from 0:00 as 0)
   'text': ""                              # any notes (unused by logic)
}
```

You likely don't need to directly see this structure; using the provided methods suffices.

Scheduled Routines for Entities (Character/Place Schedule)

Beyond one-time timers, SOWRL includes a **schedule system for entities** like characters, places, and objects. This allows defining repeating events or routines, such as "Character A is in the park from 9:00 to 12:00 on Mondays and Wednesdays" or "The shop is open at certain hours".

The schedule system is quite powerful: you define events for a specific entity by kind (person, clicky, unclicky, place) and give each event an ID, time interval, days, and effect on the entity's status/location. The system will then automatically "process" these events as time changes, updating the entity's

`location` , `status` , etc., according to the schedule. This can remove a lot of manual work in moving characters around.

Scheduled Event Properties: Each scheduled event for an entity is essentially a dict with keys like:
- **`id`** – an identifier for the event (unique per entity).
- **`status`** – a label/name for the event or state during that event (often used to derive image folder or sprite, e.g., "sleep" or "work"). This might change the entity's `status` field and thereby which image is shown (assuming images named by status).
- **`location`** – the location key where the entity should be during this event.
- **`x`**, **`y`** – coordinates where the entity should be placed during this event. (For example, at work they might be at one position, at home at another position in that background.)
- **`day`** – the day or list of days when this event occurs. Use a list of day numbers (depending on how you count days in your game, e.g., 0-6 for weekdays or 1-30 if your game's calendar resets each month). If single number, wrap it in a list. Example: `[1,3,5]` for Monday, Wednesday, Friday if you designate those numbers, or `[15]` for 15th of the month. (The framework's default might consider day numbering within a month.)
- **`interval`** – a string `"HH:MM-HH:MM"` representing the daily time window when this event is active. For example, `"09:00-17:00"` means from 9 AM to 5 PM. The entity will be in the specified location and status during that interval on the specified days.
- **`isActive`** – Boolean flag for the event itself. If False, the event is ignored. This allows toggling schedule events on or off (for example, a character might normally follow a routine, but if an event is disabled, the routine won't happen and you can manually control them). If *no* active event is currently affecting a character, then you (or the player) can manually change their position/status; otherwise, the schedule overrides their location/status.
- **`text`** – Extra text field for the event (could describe it, e.g., "Alice at work in cafe"). Not used by logic but may be displayed in a schedule UI for reference.

Adding Scheduled Events:
Use the `date` (or schedule) functions to define these events:

- **`date.add(kind, cfname, id, status, location, x, y, day, interval, arts=None)`** – Add a scheduled event for an entity.
- *kind*: The type of entity – must be one of `'person'` , `'clicky'` , `'unclicky'` , or `'place'` . This tells the scheduler which controller to affect.
- *cfname*: The key name of the specific entity (character or object) this event is for.
- *id* (or *id_*): A unique ID for this event (string). You choose this name to reference or modify the event later. e.g., `"morning_job"` .
- *status*: The status name to set during this event (affects image lookup as well). e.g., `"working"` or `"closed"` .
- *location*: The location key where the entity will be during the event.
- *x, y*: Coordinates for the entity at that location during the event.
- *day*: The day or list of days when this event occurs. Use a list even for one day. e.g., `[2]` for day 2 of the month, or `[0,1,2,3,4]` for Monday-Friday if your game counts those, etc. (The system likely treats day numbers modulo some cycle, possibly 0-6 for a week or 1-30 for a month; default says "within a game month". If your game doesn't have months, consider day as day-of-week or similar. You may need to experiment with what constitutes day 0 vs 1 depending on your design.)
- *interval*: Time range string "start-end". Must be within 00:00 to 23:59. If the current time falls in this interval on the correct day, the event is considered active.

- *arts*: (optional) Alternative image folder name if different from the entity's `cfname` when in this event. If not None, the scheduler will temporarily treat the entity's art key as this for the duration (useful if, say, a character has a different outfit at work, stored under different sprite folder).
Example:

```
# Schedule: Jane works at the cafe (location 'cafe') from 09:00-17:00 on
weekdays (let's say days 1-5)
$ game.date_controller.add('person', 'jane', 'work_shift', 'working',
'cafe', 300, 200, [1,2,3,4,5], "09:00-17:00")
```

In this example, we add an event for person 'jane' with ID 'work_shift'. When active, Jane's status will be "working", she'll be at location 'cafe' at coordinates (300,200). It occurs Monday-Friday (if we treat day 1-5 as weekdays) during 09:00–17:00 each of those days. If we have sprites for Jane's "working" status (maybe in a uniform), we ensure those exist. We could also specify an `arts` if her working outfit sprites are in a different folder.

- **`date.get(kind, cfname, id=None, keyname=None)`** – Retrieve scheduled events.

- If *id* is provided, returns that event's dictionary for the entity. If *id* is None, returns a dictionary of all events for that entity. The structure might be `{ event_id: {event_data}, ... }` .

- *keyname*: If provided, returns just that property from the event dict (only works if id is specified).
Example:

```
# Get all schedule events for Jane
$ jane_events = game.date_controller.get('person', 'jane')
# Get the interval of Jane's 'work_shift' event
$ work_interval = game.date_controller.get('person', 'jane',
'work_shift', 'interval')
```

- **`date.set(kind, cfname, id, keyname, val)`** – Modify a property of a scheduled event.
This lets you change an event after creation. For example, you could extend the hours or change location.
Example:

```
# Change Jane's work shift to end at 18:00 instead of 17:00
$ game.date_controller.set('person', 'jane', 'work_shift', 'interval',
"09:00-18:00")
# Deactivate a particular event (maybe a day off)
$ game.date_controller.set('person', 'jane', 'work_shift', 'isActive',
False)
```

Setting `isActive False` on a schedule event will effectively pause that routine, allowing you to manually control the character during that time if needed.

- **`date.remove(kind, cfname=None, id=None)`** – Remove scheduled events.

- If you specify both an entity (`cfname`) and an event ID, it will remove that one event from the schedule.
- If you specify an entity but `id=None` , it will remove **all** events for that entity.
- If you even omit `cfname` (and `id`), it may remove all events of that kind for all entities (use with extreme caution!).
  Example:

```
# Remove Jane's work_shift schedule (perhaps she quit her job)
$ game.date_controller.remove('person', 'jane', 'work_shift')
# Remove all schedules for Jane
$ game.date_controller.remove('person', 'jane')
```

- **Internal Processing:** You don't manually call the event processing in normal use; the framework calls `date.process(kind, location, day_time, month_day)` under the hood each time the time or location changes (likely every few real-time seconds or every in-game minute tick). But for completeness:
  `date.process(kind, location, day_time, month_day)` – Checks and applies schedule events for a given location and time.

- *kind*: 'person', 'clicky', 'unclicky', or 'place' — the category to process.
- *location*: the current location to consider.
- *day_time*: current time string (like "14:30").

- *month_day*: current day-of-month number. (The schedule system assumes a repeating month cycle for days, unless you treat it differently.)
  This will activate any events whose interval contains `day_time` and whose day list contains `month_day` , for entities of that kind, *and at that location* presumably. It will update those entities (moving them there, setting status, etc.), and deactivate events that are no longer current.
  Normally, you do not need to use this because SOWRL's loop calls it appropriately. Only advanced users might manually trigger a processing (for example, right after a large time skip, to immediately refresh positions — though `time_to` likely calls it internally as well).

- `date.clean(kind, puppetdict)` – Clean up schedule data by removing events for entities that no longer exist.

- *puppetdict*: the dictionary of existing entities (for the given kind) to compare against. For example, `persons` dictionary for kind 'person'. This method will remove any scheduled events referencing an entity that isn't present in the given dictionary (which prevents orphan events from causing errors).
  Typically, this is called automatically if you remove entities. You might call it if you dynamically remove characters or places mid-game to ensure no schedule tries to access them. Usually internal use.

The SOWRL framework likely provides a UI screen for schedules (the docs mentioned a schedule window in the views). There are helper functions:

- `date.show_schedule(kind, cfname)` – Opens a schedule viewer UI for the given entity.
  This might be a debugging or admin feature to visually see that entity's schedule. (It's mentioned

as "opens schedule window for the entity"). Not needed for gameplay except for maybe a journal feature or dev debugging.

- **`date.hide_schedule()`** – Closes the schedule window.

Using scheduled routines effectively: Once you've defined events with `add`, the framework does the rest. For example, if it's 10:00 AM on Monday, the earlier "Jane work_shift" event will be active, so SOWRL will automatically: set Jane's `location` to "cafe", her `status` to "working", move her to (300,200), and if you have a sprite at `images/jane/working.png` (or perhaps in a subfolder if arts specified), it will display that sprite at the cafe. At 17:00, the interval ends, and SOWRL will mark that event inactive, potentially leaving no active event for Jane — at which point, since no other event for Jane is active and she's free, you could have another schedule that starts at 17:00 (maybe goes home), or if none, Jane will remain where she was until something moves her (one might typically define another event for the evening or set her isActive false after work).

If no schedule event is active for an entity (and `isActive` in all events is False or none match current time), the engine considers them "free". The note in the documentation: *"If there are no active events waiting or executing, then you can manually set the position or change other data."* That means outside their scheduled times, you can move them via code or let the player interact to move them.

This schedule system can be applied to places as well (for example, maybe a place has an event where it's "closed" during certain hours — you could set a place's status to "closed" at night and perhaps a clicky that tries to enter it checks that status).

Example scenario using schedule:

```
# Create a place 'market' and by default, it's open.
$ places.add("Market", "market")
# Add a schedule for market to be closed at night
$ game.date_controller.add('place', 'market', 'closed_night', 'closed',
'market', 0,0,
                    day=list(range(1,8)), interval="21:00-06:00")
# (Assume days 1-7 cover all days of week here, and interval wraps past
midnight)
```

The above event sets the market's status to "closed" from 9PM to 6AM every day. In practice, since 21:00 to 06:00 spans midnight, the implementation might require splitting into two events (one from 21:00-23:59 and another 00:00-06:00). The system may or may not support an interval that wraps around midnight directly. If not, you could schedule closed_night for 21:00-23:59 and closed_early for 00:00-06:00. Anyway, during those intervals, the place's `isActive` might remain True (the place exists) but maybe you check its `status` to decide if the player can go or not. You could also tie it with triggers to prevent travel if status is closed.

Which leads us to triggers next:

Trigger System: `OkiTriggerController` (Conditional Events)

Triggers in SOWRL allow you to evaluate arbitrary conditions continuously and perform actions when those conditions become true. They are a powerful way to create dynamic events that depend on game

state — for example, “if the player’s money hits 1000, trigger a special event” or “when it’s midnight and player is at the graveyard, show a ghost”.

The **OkiTriggerController** (global **triggers**) manages these conditional checks.

Each trigger has:
- **name** – An optional descriptive name (could be same as cfname or different) for reference.
- **cfname** – Unique key for the trigger (identifier string).
- **case** – A condition expression (string of Python code) that will be evaluated. It should return True or False. Often this involves game.store variables, controller queries, etc. e.g., "game.store.money >= 1000 and places.get_location() == 'bank'" . Because it’s a string, you can write complex logic.
You have access to all Python objects in the game state when it runs.
- **do** – The action to perform (string of Python code) when the condition is True. For example: "renpy.call('bonus_event')" , or "renpy.say(narrator, 'You feel a chill down your spine.')" , or direct calls to controller methods. This code executes right when the trigger condition is detected as True.
- **text** – Extra text field (not used by logic, could be a description for your own logging or a message perhaps).
- **isActive** – Not explicitly listed in the snippet, but implied by examples. Likely a boolean that can temporarily disable a trigger without removing it. If isActive is False, the trigger will be ignored even if its condition is true. By default triggers are probably active upon creation (True). You can set this via triggers.set if you want to turn a trigger off for a while.

Managing Triggers:

- **triggers.add(cfname, case, do)** – Create a new trigger.
- *cfname*: Unique name for the trigger.
- *case*: Condition string (Python expression to evaluate).

- *do*: Action string (Python code to execute when condition holds).
  Example:

```
# Add a trigger that checks if money is 12345 and player is in park,
then plays a scene.
$ triggers.add("test_trigger", "game.store.money == 12345 and
places.get_location() == 'park'",
            "renpy.say(narrator, 'This is it!')")
```

  In this example (inspired by earlier notes), the trigger named "test_trigger" will continuously monitor if game.store.money equals 12345 **and** the current location is 'park'. The moment both are true, it will make the narrator say "This is it!". (In practice, you might call a label or do a more elaborate event, but this illustrates running multiple conditions and an action.)

- **triggers.get(cfname, keyname=None)** – Get info about a trigger.

- *cfname*: Trigger key.

- *keyname*: (optional) If provided, returns only that property from the trigger’s dict (e.g., "case" or "isActive"). If omitted, returns the full dictionary for that trigger, or False if not found.

Example:

```
$ cond = triggers.get("test_trigger", "case")
$ active_flag = triggers.get("test_trigger", "isActive")
```

- **triggers.set(cfname, keyname, val)** – Modify a trigger's property.

- *cfname*: Trigger key.
- *keyname*: Field to change, e.g. "case" , "do" , or "isActive" .

- *val*: New value.
Example:

```
# Temporarily disable the trigger without removing it
$ triggers.set("test_trigger", "isActive", False)
# Change the action to call a label instead of say
$ triggers.set("test_trigger", "do", "renpy.call('special_event')")
```

- **triggers.remove(cfname)** – Remove a trigger entirely.
Example:

```
$ triggers.remove("test_trigger")
```

How Triggers Work: Once added (and active), triggers are evaluated automatically, likely every tick of the game loop or whenever relevant state changes. The `case` expression is `eval`'d in Python. If it returns True and the trigger is active, its `do` code is executed immediately at that moment. Depending on the design, the trigger might then be deactivated or removed automatically after firing (if it's meant to be a one-time event) or it might continue to fire repeatedly each time the condition is True. In many frameworks, triggers are one-shot by default — they execute once and then either mark themselves inactive or need removal. The presence of an `executed` flag in timers but not explicitly in triggers suggests triggers might not auto-remove. You may need to remove or disable it manually inside its own action if you only want it once. For example, your `do` code could include something like `triggers.remove('mytrigger')` or `triggers.set('mytrigger','isActive',False)` if you want it to run just once.

Alternatively, design triggers such that the condition becomes false after the action (e.g., trigger checks if a flag is False, and the action sets that flag True). That way it won't fire again because the condition is no longer satisfied.

Example Use Cases:
- Trigger story events when player reaches certain stats or items: e.g., `case="machine.get('fame') > 50"` and `do` calls a scene.
- Trigger location-based events at specific times: e.g., `case="places.get_location()=='graveyard' and game.store.time == '00:00'"` do some ghost encounter.
- Trigger tutorial popups when entering an area for the first time (with a flag to not repeat).

- Toggle environment changes: e.g., if a global variable `power_on` becomes False, do triggers that turn off lights (set some images).

The triggers system essentially allows game logic reactivity without having to constantly check conditions manually in your script – you declare them and SOWRL will handle checking them in the background.

Story/Quest System: `OkiStoryController` (Storylines)

Building nonlinear games often means managing multiple plot lines or quests. The **`OkiStoryController`** (global **`story`** or sometimes referred to as **`stories`**) is designed to help track the player's progress in various story arcs and enforce certain game flow constraints (like locking locations until a certain part of the story, or providing hints to the player about what to do next).

Think of each "story" as a quest or plot thread that can be enabled/disabled and has multiple parts (stages).

Each story has:
- **`name`** – The title or name of the story (for display or reference, e.g., "Main Quest" or "Alice's Romance").
- **`cfname`** – The unique key identifier for the story (used in code).
- **`part`** – The current part number of the story (an integer, starting from 0 or 1). This indicates progress. For a quest, part 0 might mean not started, part 1 first phase, etc. You increment or set this as the player advances.
- **`isActive`** – Boolean indicating if the story is currently active. Inactive could mean the quest is not available yet or has been completed/paused. Active means it's in progress and the game should enforce its restrictions or show its hints. By default, a new story is inactive (`False`) until you enable it.
- **`parthint`** – A dictionary mapping part numbers to hint text. This stores hint messages for each part of the story, which can be shown to the player (for example, in a journal or hint screen).
- **`partloc`** – A dictionary mapping part numbers to restricted locations and an action. This defines location restrictions for each story part: basically "while the story is at this part, these locations are off-limits; and if the player tries, perform this action". You use this to prevent the player from going somewhere until a certain story progress, e.g., lock the city gate until part 2 of main quest.
- **`text`** – Extra field for any notes or description of the story.

Managing Stories:

- **`story.add(name, cfname, part=0)`** – Create a new story entry.
- *name*: The story's title.
- *cfname*: Key for the story.

- *part*: Starting part number (default 0).
  When you add a story, it's recorded with the given starting part and isActive set to False by default.
  Example:

```
$ story.add("The Lost Amulet", "lost_amulet", part=0)
```

This sets up a quest "The Lost Amulet" with key `"lost_amulet"` , currently at part 0 (not yet begun). You can now configure hints or restrictions and then enable it when appropriate.

- **`story.get(cfname, keyname=None)`** – Retrieve story information.

- *cfname*: Story key.

- *keyname*: (optional) If given, returns that specific property (e.g., `"part"` or `"isActive"`). If omitted, returns the whole story dict, or False if not found.
  Example:

```
$ current_part = story.get("lost_amulet", "part")
$ active_flag = story.get("lost_amulet", "isActive")
```

- **`story.set(cfname, keyname, val)`** – Modify a story property.

- *cfname*: Story key.
- *keyname*: Field to set (e.g., `"part"` or `"isActive"`).
- *val*: New value.
  Example:

```
# Manually set the story part (not typical, usually use set_part, but
possible)
$ story.set("lost_amulet", "part", 1)
# Mark story as active (again, enable() is the method provided, but this
is underlying)
$ story.set("lost_amulet", "isActive", True)
```

There are also specialized methods for common story operations:

- **`story.enable(cfname)`** – Activate a story.
  Sets `isActive` to `True` for that story. You should call this when the player starts or discovers the quest. When active, any part-specific restrictions (`partloc`) will be enforced and you might present hints.
  Example:

```
$ story.enable("lost_amulet")
```

- **`story.disable(cfname)`** – Deactivate a story.
  Sets `isActive` to `False`. You might call this when a quest is completed or temporarily paused. Inactive means its restrictions are lifted (the player can freely travel again if locations were locked) and you probably won't show its hint.
  Example:

```
$ story.disable("lost_amulet")
```

- **story.get_part(cfname)** – Get the current part number of the story.
  Example:

```
$ part_num = story.get_part("lost_amulet")
```

- **story.set_part(cfname, number)** – Set the story's part to a specific number.
  This is used to advance (or rewind) the story progress. It doesn't automatically do anything else except update the number; you might script additional effects when part changes (like enabling triggers or changing NPC schedules).
  Example:

```
$ story.set_part("lost_amulet", 1)
```

  Perhaps after the player accepts the quest, you set part to 1.

- **story.part_up(cfname, number=1)** – Increase the story's part by a certain amount (default 1).
  This is a convenient way to move to the next part. It essentially does `set_part` to current part + number.
  Example:

```
$ story.part_up("lost_amulet")  # increment from 0 to 1
```

- **story.part_down(cfname, number=1)** – Decrease the story's part by a certain amount.
  Possibly used if you want to revert progress or handle branching where a quest might move backwards.
  Example:

```
$ story.part_down("lost_amulet")  # maybe drop from part 2 back to 1 if
something failed
```

- **story.set_parthint(cfname, part, hint)** – Define a hint text for a specific part of the story.

- *part*: The part number to attach this hint to.

- *hint*: A string of text giving a hint or objective for that part.
  These hints can be displayed to the player via a "hints" or "journal" screen to guide them.
  Example:

```
$ story.set_parthint("lost_amulet", 1, "Find the old map in the
library.")
```

```
$ story.set_parthint("lost_amulet", 2, "Go to the forest marked on the
map and search for clues.")
```

- **`story.get_hint(cfname, part)`** – Retrieve the hint text for a given part of a story. Returns the hint string, or False if none is set.
  Example:

```
$ current_hint = story.get_hint("lost_amulet",
story.get_part("lost_amulet"))
if current_hint:
    narrator current_hint
```

This would display the current part's hint to the player.

- **`story.set_partloc(cfname, part, location, do)`** – Set a location restriction for a given story part.

- *part*: The part number during which this restriction applies.
- *location*: A location key or list of location keys that are **not allowed** during this part. If the player tries to travel to any of these locations while the story is at that part, the trigger will fire.
- *do*: Python code (as string) to execute when the player attempts to go to those locations during this part. Typically, you might want to show a message or redirect them. For example, if the city gates are closed, you might do `renpy.say(narrator, "The gates are locked. I can't leave yet.")` . Or you could automatically send them back.
  Example:

```
# During part 0 of lost_amulet, prevent entering "forest" and "cave"
$ story.set_partloc("lost_amulet", 0, ["forest", "cave"],
"renpy.say(narrator, 'I should not venture there yet.')")
```

In this case, until the story progresses beyond part 0, if the player tries to go to forest or cave (maybe by clicking a travel button), the game will instead execute that renpy.say line and presumably not actually change the location. (Under the hood, the framework likely checks when you use `places.set_location` or similar; if the target location is restricted by an active story, it triggers this code and cancels the move.)

- **`story.set_hints_background(img_path)`** – Set a background image for the hints window. If the framework includes a built-in hints screen (where it lists your quests and hints), this function lets you customize its background. Provide the path to an image file. If not set, it might default to a plain gray or some default.
  Example:

```
$ story.set_hints_background("images/ui/hints_bg.png")
```

- **story.show_hints(transition=None)** – Open the hints (story journal) screen.
  This likely displays a UI listing active stories and their hints (maybe using the parthint texts). You can call this to show the player their current objectives. Optionally provide a transition for how the screen appears or disappears.
  Example:

  ```
  # When player clicks a "Notes" button, you might do:
  $ story.show_hints()
  ```

  There should be a corresponding close or perhaps the hints screen has a close button. You could also possibly hide with hide_hints() if it existed, but likely the UI handles it.

Using the story system ensures that, while a story is active, any specified location locks are enforced and players have guidance via hints. When a story is completed (you might mark it as inactive or move to a final part), you can free up everything.

Typical quest flow example:
- Add story "lost_amulet" as above. Initially part 0, inactive.
- When the player hears about the amulet, call story.enable("lost_amulet") and maybe story.set_part("lost_amulet", 1) to start the quest at part 1, and show a hint ("Find the old map in the library.").
- The player goes to library, finds map – you detect this via trigger or event, then do story.part_up("lost_amulet") to part 2, update hint to "Go to the forest", and maybe adjust location locks (perhaps previously the forest was locked until they had the map, which you might have set via set_partloc for part 1). Now in part 2 allow forest, but maybe lock something else or no locks.
- When they complete the quest, you might do story.disable("lost_amulet") and maybe trigger some reward. The location restrictions for that story are lifted since it's not active, and you could remove its hints from the hints screen.

The story controller helps manage multiple such quests simultaneously. You can have several stories active, each with their own part and restrictions. The player's free-roaming is the intersection of all active stories' restrictions (i.e., a location is locked if *any* active story says it is locked for its current part).

Shops and Inventory: OkiShopController & OkiProductController

No open-world game is complete without the possibility of buying or finding items. SOWRL provides a basic shop system for managing shops (stores) and products (items for sale).

Shops (OkiShopController, global shops)

A "shop" in SOWRL represents a store or vending location in the game. Each shop can have multiple products associated with it (managed by the product controller). The shop controller also manages the UI for the shop interface (how items are listed and displayed when a shop is open).

Shop Properties:
- **name** – Display name of the shop (e.g., "Joe's Antiques"). This is shown in the shop UI title likely.
- **cfname** – Key identifier for the shop (e.g., "joes_antiques"). Used in code and for image lookups

(storefront image or icon).
- **arts** – Art key for shop images. If None, uses cfname . If you have a special icon or background for the shop under a different name, set it here.
- **isActive** – Boolean (default True). If False, the shop is considered not present in the world (maybe closed permanently or not yet introduced). Inactive shops will likely be ignored or not openable via the UI. You might set a shop inactive if, for example, the shopkeeper is gone. The shop's open method will refuse to open if isActive is False.
- **image** – Path to the shop's current image. Perhaps an icon or a background used for the shop screen. On creation, a default image path is set (maybe something like images/shops/<cfname>.png). You can override this if needed.
- **text** – Extra description or notes about the shop. Not used by core logic, but you could use it to store a shop description to show the player.
- **use_daypart** – Boolean (default False for shops). If True, the shop's image might change based on time of day (e.g., a shop icon that lights up at night). Usually shops might not need daypart-specific images, hence default False. If you set it True, ensure you have different images (like shop_night vs shop_day etc.).

Additionally, the shop controller defines **UI layout properties** that affect how the shop screen is rendered (so you can adjust the appearance of the item grid):

- **row_sort** – Layout direction, 'vertical' or 'horizontal' . This likely decides whether products are listed in rows downwards or in a horizontal carousel. Default 'vertical' .
- **max_items** – How many products to show per page (default 6). If a shop has more items than this, it might paginate.
- **slip** – Overall vertical offset for the list of items (default 170 pixels). Adjust if your UI needs shifting.
- **image_x** , **image_y** – The display size (width and height) of each product's image icon in the shop UI. Defaults 70x70. So product icons will be 70px square by default.
- **v_gap** – Vertical gap between item entries (default 10).
- **h_gap** – Horizontal gap between items (for horizontal layout; default 50).
- **width_slip** – Relative horizontal offset for elements (default 0.2, meaning items are offset a bit from center).
- **text_color** – Color for text in the shop UI (default "#000" black).
- **text_outlines** – Outline style for text (default a list [(3,"#000",0,0), (2,"#fff",0,0)] which likely means a black outline of width3 and a white outline of width2, giving a readable stroke effect).

These properties can be tweaked via shops.set if needed to customize the look of the shop screen (font colors, spacing, etc.).

Managing Shops:

- **shops.add(name, cfname, arts=None)** – Create a new shop.
- *name*: Display name of the shop.
- *cfname*: Key for the shop (used as ID and base for assets).
- *arts*: (optional) alternate key for shop art assets if not using cfname .
  Example:

```
$ shops.add("Jizz Mart", "jizz")
```

(From the notes, they had `shops.add('Jizz', 'jizz')` . That presumably created a shop named "Jizz" with key "jizz".) In our example, "Jizz Mart" with key "jizz". The shop will now exist, and you can add products to it.

- **`shops.get(cfname, keyname=None)`** – Retrieve shop data.

- *cfname*: Shop key.

- *keyname*: (optional) property to fetch. If omitted, you get the whole shop dictionary, or False if it doesn't exist.
  Example:

```
$ shop_info = shops.get("jizz")
$ is_open = shops.get("jizz", "isActive")
$ shop_title = shops.get("jizz", "name")
```

- **`shops.set(cfname, keyname, val)`** – Modify a shop property.

- *cfname*: Shop key.
- *keyname*: Field to change (could be a UI layout property or something like "isActive").

- *val*: New value.
  Example:

```
# Temporarily deactivate the shop (maybe it closes for renovation)
$ shops.set("jizz", "isActive", False)
# Change the shop's display name
$ shops.set("jizz", "name", "Jizz Mart Deluxe")
# If we wanted to enable day/night images for the shop:
$ shops.set("jizz", "use_daypart", True)
```

- **`shops.get_image(obj_arts, dp=True)`** – Utility to get the path of an image for a given art key, optionally considering daypart.
  This is similar to the one in places or persons. It will return the image file path that corresponds to `obj_arts` . If dp (daypart) is True, it picks according to current time of day; if False, just the base image. This might be used internally to show the correct shop image. As a developer, you might not need to call it often except maybe to manually fetch an image path.
  Example:

```
$ icon_path = shops.get_image("jizz", dp=False)
```

This might return something like `"images/shops/jizz.png"` .

- **`shops.open(shop='', transition=None)`** – Open the shop UI for a given shop.

- *shop*: The key of the shop to open. If empty or not provided, perhaps it opens a default or last used shop, but generally you should pass the shop's cfname.
- *transition*: optional Ren'Py transition to use when closing the shop screen. The shop likely opens as a screen overlay (pausing the game scene but not using Scene/noScene since it's a UI). When the player exits the shop, this transition will apply.

  What it does: It brings up the shop interface (likely a screen showing the shop name, maybe an image, and a list/grid of products with their prices and possibly a buy button for each). While this screen is up, normal gameplay is paused. The player can purchase items or exit.

  The note in docs: *"this method does not allow opening a shop in the game scene, but does not forbid using it method"* suggests `shops.open` always opens as a screen (not on the scene layer), meaning you can't have the shop UI appear while still in free-roam mode – it covers it or uses its own layer. This is typical; you wouldn't want the player moving around while a shop menu is open.

  Example:

  ```
  # Player interacts with a shopkeeper NPC, you call:
  $ shops.open("jizz")
  ```

  This will bring up the shop interface for the "jizz" shop, showing all products added to that shop, etc. The player can then select items to buy (how the buying is handled – probably automatically deducting some currency variable – depends on how you integrate it, possibly with triggers or additional code).

Ensure that you have defined products for the shop (using `products.add`, see below) before opening, or the shop might be empty.

Products (`OkiProductController`, global `products`)

Products are the items that can be sold in shops. The **`OkiProductController`** manages all items in all shops. Each product has a price and some properties.

Product Properties:
- **`name`** – Name of the product/item (e.g., "Health Potion").
- **`cfname`** – Key identifier for the product (e.g., `"potion_health"`). This is used as ID and for finding the item's image.
- **`shop`** – The key of the shop where this product is sold. (Each product belongs to one shop; you could list an item in multiple shops by adding two products entries with same name but different shop keys.)
- **`price`** – Cost of the item (an integer value, presumably in some currency units your game uses).
- **`arts`** – Asset key for the item's sprite/icon. If None, uses `cfname` to find an image. If your image file is named differently, set arts accordingly.
- **`buyed`** – Boolean indicating if the item has been bought (the term "buyed" is presumably used as "purchased"). If `True`, it means the player already bought this item, and it will likely be removed from the shop list (assuming one-time items). This field helps avoid showing already purchased unique items. By default for new products this should be False (available). When the player buys the item via the UI, the system should set this to True (and indeed, there's a method to get all not-yet-bought items). If your design allows buying multiple of the same item, you might not use this flag, or you might treat each product as a single stock.
- **`isActive`** – Boolean (default True). If False, the item is not available in the shop (perhaps out of stock or disabled). This is another way to hide items without removing them. The `get_not_buyed_list` method can filter by this flag as well.

- **image** – Path to the item's image (icon). Initially None or auto-set to a default based on cfname or arts . The system likely expects an image in an items folder or such. E.g., images/products/potion_health.png . It will fill this when displaying the shop UI.
- **text** – Extra description or notes for the item (e.g., "Restores 50 HP"). You could use this to show item descriptions in the shop UI or in an inventory.
- **use_daypart** – Boolean (default True for products). If True, the item icon can vary with time of day (though that's unusual for an item – maybe not needed unless the item appearance changes somehow). Possibly always True by default but doesn't matter if no alt icons exist.

Managing Products:

- **products.add(name, cfname, shop, price, arts=None)** – Add a new product to a shop.
- *name*: Item name.
- *cfname*: Item key.
- *shop*: Shop key where it's sold.
- *price*: Price of the item (int).

- *arts*: (optional) alternate key for the item's image if not using cfname.
  Example:

```
$ products.add("Health Potion", "potion_health", "jizz", 300)
$ products.add("Mana Potion", "potion_mana", "jizz", 250)
```

  This adds two products to the "jizz" shop: Health Potion (cost 300) and Mana Potion (cost 250). Their keys are potion_health and potion_mana; we assume we have corresponding icons named accordingly.

- **products.get(shop, cfname=None, keyname=None)** – Retrieve product info.

- *shop*: If you pass a shop key and no other arguments, you get a dictionary of all products in that shop. Likely structured as { product_key: {product_data}, ... } .
- *cfname*: If provided, returns that specific product's data (dict) from the given shop.

- *keyname*: If provided along with cfname, returns just that field of that product (e.g., price or isActive).
  Example:

```
# Get all products in "jizz" shop
$ all_items = products.get("jizz")
# Get data for the health potion in that shop
$ potion_info = products.get("jizz", "potion_health")
# Get just the price of health potion
$ price = products.get("jizz", "potion_health", "price")
```

- **products.set(shop, cfname, keyname, val)** – Modify a product's property.

- *shop*: Shop key where the product is.

- *cfname*: Product key.
- *keyname*: Field to set (e.g., `"price"` , `"isActive"` , `"buyed"`).
- *val*: New value.
  Example:

```
# Put a product on sale by changing price
$ products.set("jizz", "potion_health", "price", 150)
# Mark the health potion as purchased (so it won't show again)
$ products.set("jizz", "potion_health", "buyed", True)
```

- **`products.get_not_buyed_list(shop, isActive=True)`** – Get a list of all products in a shop that have not been bought yet, filtered by active status.

- *shop*: Shop key.

- *isActive*: If True (default), only consider products that are active. If False, you might get only inactive ones or something – usually you keep default.
  Returns: A list of product dictionaries (or maybe just keys) for all items where `buyed == False` and, if isActive filter is True, also `isActive == True` . Essentially, this is the list of items to display in the shop for sale.
  The shop UI likely uses this to show what's available.
  Example:

```
$ available_items = products.get_not_buyed_list("jizz")
```

  If health potion was marked buyed, it would be excluded from this list now, meaning it won't appear in the shop menu.

- *(Possibly)* **`products.remove(cfname)`** – If implemented, would remove a product entirely. It was not explicitly listed, but logically consistent. If needed, you can remove an item (maybe if it's out of stock permanently). If not available, you can simulate removal by marking `isActive=False` or `buyed=True` to hide it.

Using Shops and Products:
Typically, you will define all your shops and products during initialization (or when they become relevant). For example:

```
init python:
    # Create shop
    shops.add("General Store", "general_store")
    # Add products to it
    products.add("Apple", "apple", "general_store", 5)
    products.add("Iron Sword", "sword_iron", "general_store", 100)
    products.add("Health Potion", "potion_health", "general_store", 30)
```

During gameplay, when the player interacts with the shop (maybe through a clicky, or a character in that location), you would call `shops.open("general_store")`. The shop screen appears, listing "Apple", "Iron Sword", "Health Potion" with their prices. The player buys something (the framework likely has a built-in buying mechanism: e.g., clicking an item might automatically deduct some currency variable like `game.store.money` and mark the item as bought if one-of-a-kind).

You should ensure you have a way to track player currency. The system doesn't declare one explicitly, but in triggers examples they used `game.store.money`. So perhaps you manage money as a variable in `game.store` or `machine`.

If an item is not meant to be bought again, once purchased, mark it `buyed=True` (the shop UI might even do that for you automatically). Next time the shop opens, that item will not appear (because get_not_buyed_list will filter it out).

Shops can also be closed by making them inactive or by not calling `shops.open`. The `shops.open()` is typically invoked from some in-world trigger (like clicking a shop entrance).

Inventory: Note that SOWRL's shop system covers buying items, but it doesn't explicitly mention a player inventory system. You would likely manage inventory yourself (e.g., by adding purchased items to a list of owned items, which could be stored in `game.store` or `machine`). For example, after a purchase, you might do `machine.add("has_sword", True)` or maintain `game.store.inventory = ["sword_iron", ...]`. The integration of purchase effects is up to your game logic (you might use triggers for "when item bought, do X").

Now that all components are described, you can combine them to create complex gameplay. For instance, you can have a trigger that listens for a story part completion to unlock a shop, or a scheduled event that moves a merchant (character) to a shop location at 9:00, and triggers enabling the shop only when he's present.

Global Variables Storage: `OkiMachineController` (`Machine`)

To facilitate custom game variables and flags (such as stats, quest flags, relationship points, etc.), SOWRL provides the **`OkiMachineController`** (global **`machine`**) as a simple key-value storage. You can think of it as a dictionary that persists in game state, separate from Ren'Py's persistent or store, meant for in-game usage.

Use the machine to store anything that doesn't naturally fit into other controllers. For example, player stats like money, reputation, or a counter of secrets found.

Machine functions:

- **`machine.add(varname, value)`** – Create a new variable or key in the machine and set its value.
- *varname*: The name of the variable (string key).
- *value*: Any value (number, string, bool, even list or dict) to store.
  If the key already exists, this will overwrite it (so it can function as a set as well).
  Example:

```
$ machine.add("money", 100)
$ machine.add("fame", 0)
$ machine.add("guild_member", False)
```

This initializes three variables in the machine: money = 100, fame = 0, guild_member = False.

- **`machine.get(varname=None)`** – Retrieve values from machine.

- If *varname* is provided, returns the value associated with that key, or False if the key is not found.

- If *varname* is None, returns the entire dictionary of all machine variables.
  Example:

```
$ gold = machine.get("money")
$ all_stats = machine.get()
```

After our previous add, `gold` would be 100, and `all_stats` might be `{"money":100, "fame":0, "guild_member":False}` .

- **`machine.set(varname, value)`** – Update the value of an existing machine variable.

- *varname*: Key to update.

- *value*: New value.
  Important: This will **not** create a new variable if the key doesn't exist. It only works for existing keys. (If the key isn't there, it returns an error or does nothing.) So use `add` for new keys, and `set` for changing known keys.
  Example:

```
$ machine.set("money", 150)    # player earned or cheated money
$ machine.set("guild_member", True)  # player joined the guild
```

If "money" existed, now it's 150. If "guild_member" existed, now True. If you tried `machine.set("new_var", 123)` without adding "new_var" first, it would not create it — you'd use add for that.

- **`machine.remove(varname)`** – Delete a variable from the machine.

- *varname*: Key to remove.
  Example:

```
$ machine.remove("fame")
```

This would remove the "fame" entry entirely from the machine. After this, `machine.get("fame")` would return False.

Machine storage is essentially a way to avoid cluttering Ren'Py's global store with lots of custom variables and to keep them organized. Triggers and other systems can easily read these machine variables (as seen: triggers using `game.store.money` in examples; you could similarly do `machine.get("money")` or if you alias the machine in triggers environment, possibly `machine['money']`).

Internally, `game.store` and `game.machine_controller` might both be accessible, so you might see triggers using `game.store.somevar` or directly `machine.get(...)` . Decide on one and stick to it. You could put "money" in `game.store.money` or in `machine` . If using `machine` , maybe use triggers like `machine.get('money')` in conditions. Possibly simpler is just to do `game.store.money` because it behaves like a normal attribute.

However, the machine controller ensures you have methods to manipulate variables systematically, which might be beneficial for saving/loading or consistency.

Use machine for: - Player stats (money, health, etc. if not tied to a particular entity).
- Global flags (took key item, met character X, solved puzzle Y).
- Anything that multiple triggers or story conditions need to check.

Because it's just a Python dict under the hood, it will be saved in Ren'Py save files as part of game state. There's no separate persistence logic needed beyond Ren'Py's normal save for the store objects.

Utility Functions: `OkiExtra` (Math and Helpers)

The **`OkiExtra`** class (available as `game.extra` or you can alias `default extra = game.extra`) contains miscellaneous utility methods that can be handy during game development. These functions perform common calculations that might be used for gameplay or visual effects.

Key functions in OkiExtra include:

- **`extra.ceil(a, b)`** – Divide number `a` by `b` and round **up** to the nearest integer. (Mathematical ceiling of a/b.) For example, `ceil(5,2) = 3` because 5/2 = 2.5 rounds up to 3. Use case: maybe calculating how many pages for X items per page, etc.
- **`extra.floor(n)`** – Floor of a number (round down to nearest integer). E.g., `floor(2.9) = 2`.
- **`extra.mean(numbers)`** – Takes a list of numbers and returns their arithmetic mean (average). E.g., `mean([1,3,8]) = 4`.
- **`extra.cos2D(ax, ay, bx, by)`** – Compute the cosine of the angle between two vectors (from origin to points (ax,ay) and (bx,by)). This is a common math helper for angle calculations in 2D. Returns a float between -1 and 1. (Useful if you want to check orientation or similarity of directions.)
- **`extra.arccos(x)`** – Return the arccosine (inverse cosine) of x (probably in radians or maybe degrees – likely radians). Use to get an angle from a cosine.
- **`extra.distance(x1, x2)`** – Return the distance between two one-dimensional points (basically absolute difference `|x1 - x2|`).
- **`extra.distance2D(x1, y1, x2, y2)`** – Distance between two points in 2D space. Calculates √ ((x2-x1)^2 + (y2-y1)^2).

- **extra.time_to_minutes(h, m)** – Convert hours and minutes to total minutes. For example, (2, 30) -> 150 minutes. Useful to convert a time into a single number (like minutes since start of day).
- **extra.minutes_to_time(m)** – Convert a total minutes count to a time string "HH:MM". E.g., 150 -> "02:30". This likely mod 24h, etc.
- **extra.parse_time(tm)** – Take a time string "HH:MM" and return a list of two ints [HH, MM]. E.g., "07:05" -> [7,5]. Convenience for splitting times.
- **extra.parse_intervals(interval_str)** – Take a time interval string "HH:MM-HH:MM" and return a list of four ints [H1, M1, H2, M2] representing the start and end. E.g., "21:30-06:00" -> [21,30, 6,0]. This is useful to then compute things like event durations or check intervals.

These helpers are mainly to support the systems above (schedule uses parse_intervals, triggers might use time conversion, etc.), but you can also use them in your own scripts if needed.

Examples:

```
$ avg_score = extra.mean([10, 15, 20])          # avg_score = 15.0
$ dist = extra.distance2D(0,0, 3,4)             # dist = 5.0 (3-4-5
triangle)
$ angle_cos = extra.cos2D(1,0, 0,1)             # angle_cos = 0 (because
vectors (1,0) and (0,1) are perpendicular, cos90=0)
$ time_val = extra.time_to_minutes(2, 45)       # time_val = 165
$ time_str = extra.minutes_to_time(165)         # time_str = "02:45"
$ parts = extra.parse_intervals("09:00-17:00")  # parts = [9,0, 17,0]
```

Putting It All Together

SOWRL's components are designed to interoperate. Here are some general guidelines and possibilities for using them together:

- **Integration of Location, Characters, and Time:** You can create a living world by adding characters (persons.add) and giving them schedules (date.add events). As time advances (via date.add_hours automatically in a loop or via triggers like sleeping), characters will move around according to schedules. The player can move between places (places.set_location via clicking on clickies representing exits), and they will find different characters in different places at different times.

- **Using Triggers with Machine and Story:** Triggers (triggers.add) can watch for story progression or variable changes. For example, you could have triggers.add("quest_start", "story.get_part('lost_amulet') == 1", "renpy.notify('Quest started!')") , so when that story's part becomes 1, a notification pops. Or triggers could monitor machine variables: triggers.add("rich", "machine.get('money') >= 1000", "story.enable('secret_ending')") to unlock a secret story when rich.

- **Shop Availability & Story:** You might use the story or triggers to control shops. E.g., only open the Blacksmith shop after a certain quest: set shop.isActive=False initially, and triggers.add("enable_smith", "story.get_part('blacksmith_story') >= 2", "shops.set('smithy','isActive',True)") .

- **Weather and Time:** Weather effects can be tied to time or random triggers. Possibly use triggers: `triggers.add("night_rain", "game.store.time == '20:00'", "weather.set('rain','isActive', True)")` and maybe another to stop in morning. Or use the schedule system if weather can be scheduled by day/time as events (though that system is mostly for entities, not sure if weather is integrated there; likely easier via triggers or direct time checks).

- **Hints Screen:** The story hints can be shown via `story.show_hints()` . You might present this as a menu option (like a button "Quest Journal" calls this function). The `parthint` you set for each story part will appear, possibly along with story name.

- **Example Workflow:** At game start, define world:

```
$ places.add("Home", "home")
$ places.add("Market", "market")
$ persons.add("Alice", "Smith", "alice", "home", 400, 300)
$ persons.add("Bob", "Johnson", "bob", "market", 500, 300)
$ shops.add("General Store", "general_store")
$ products.add("Key", "key_item", "general_store", 50)
$ weather.add("rain", intensity=15, xforce=5, yforce=60)
$ weather.set("rain", "location", "market")
$ story.add("Tutorial", "tutorial", part=0)
$ story.set_parthint("tutorial", 0, "Go outside your home.")
$ story.set_parthint("tutorial", 1, "Reach the market.")
$ story.set_partloc("tutorial", 0, "market", "renpy.say(narrator, 'Maybe
I should check my house first.')")
$ story.enable("tutorial")
$ story.set_part("tutorial", 0)
$ triggers.add("left_home", "places.get_location() == 'market' and
story.get_part('tutorial') == 0", "story.set_part('tutorial',1)")
```

This simplistic setup:

- Adds Home and Market.
- Alice at Home, Bob at Market.
- A shop at Market with one item.
- Rain in Market location (could activate via some time or event).
- A Tutorial story that restricts going to Market until part 1. Starting part0 with a hint to "Go outside your home." Trying to go to Market at part0 triggers a block message.
- A trigger "left_home" detects when the player actually goes to Market (which implies they overcame restriction - maybe we later disable restriction after some action). Here we cheat by saying if location is market and story part is 0, then we set story part to 1 (meaning tutorial progressed).

The pieces interplay as: - The player sees hint to go outside. Initially, going to market gives a message because story part0 restricts it. - Perhaps the player does something at home (like examine something) then we decide to lift restriction: `story.disable("tutorial")` or `story.set_part("tutorial", 1)` and maybe remove the restriction for part1 (or set_partloc for part1 none). - Then they travel to Market, trigger runs, sets part1 and hint updates to "Reach the market." Actually, maybe the trigger itself progressed it. - In a real scenario, we'd refine logic, but it shows how story and triggers coordinate.

Conclusion: With SOWRL, you have a toolkit to create an open-world feel in Ren'Py. You define the static structure (places, characters, items) and dynamic rules (time schedule, triggers, story progression). The library automates a lot: characters moving on their own schedule, weather appearing, triggers watching conditions. This frees you to focus on writing the narrative and designing interactions, rather than coding the underlying state management from scratch.

Each part of the library can be used independently. If your game only needs a day/night cycle and nothing else, you could use just the `OkiDateController` to manage time and maybe trigger events. If you only need an inventory without a free-roam world, you might still use the shop system in a linear VN to present a store interface. Conversely, if you want a free-roam sandbox, you'll find all pieces working together.

Use this documentation as a reference while building your game: refer back to each section for the syntax of methods and their effects. Happy developing your Ren'Py semi-open-world adventure!
