Quickstart

Welcome to my Quickstart Manual on how to create a mod for Mind the School.c!bp The purpose of this guide is to show you how to quickly set up a mod.

Create your Mod

Here I show you how to create a Mod for Mind the School. c!pp

First you should have Ren'Py 8.1.3 installed and should use that for testing.

Higher versions of Ren'Py don't work with my game yet.

The creation of mods is rather simple. You basically need only a rpy-file together with a single method call, put it in a folder and put that folder in the mods folder of the game. You need to make sure the file name is unique across all mods, so best use your mod key as a name.

That is basically everything you need for the game to register your mod.

The folder structure therefore should look something like this:

```
Mind the School
├ MindTheSchool.exe
└ game
  └ mods
    └ YOUR_MODFOLDER
      └ your-mod.rpy
```

I added the game exe to give you some kind of anchor point.

The rpy-file itself should look something like this:

```
init -97 python:
  register_mod(
    "your-mod", # key
    "Your Mod", # name
    "0.1", # version
    "YOUR_MODFOLDER", # path
    "This could be your mod!", # description
    "YOU" # author
  )
```

These values are necessary and the game won't register your mod, if you don't include one of those.

I have attached an example metadata-file to this page, which you can just download.

The game will also automatically register the path to your metadata-file.

That means for the mod to work consistently your mod should stay within the mod folder. That means if you set a path for an image for example, you always set the path relative to your metadata file.

If you have a file structure that looks like this for example:

```
Mind the School
├ MindTheSchool.exe
└ game
  └ mods
    └ YOUR_MODFOLDER
      ├ your-mod.rpy
      ├ images
      | ├ image 0.png
      | └ image 1.png
      └ events
        └ event.rpy
```

To reference 'image 0.png', your path should look something like this: "images/image 0.png". That is because the game automatically puts the path to your mod in front of paths you add. This is so that there are no incompatibilities when you decide to rename the mod folder and to keep the paths in the code consistent.

Beware, since renpy is doing weird stuff with identical file names, be sure that your files always have a unique name.

Now what do these values in the metadata file mean?

key

This is the key to your mod. The game uses this value to recognize your mod. You can change the name of everything (except for the metadata file), even of the mod folder. As long as the key in the metadata is the same, the game will recognize the mod as the same one.

version

This shows the version of your mod. If the mod version changes since the last time the mod has been used, the game will automatically disable your mod and your mod then has to be enabled in the game again for it to work.

name, description and author

These values are just for displaying purposes. Those will be used in the mod overview in-game to show and present your mod.

path

This is the name of your modfolder. This is used for the game to find the images and other files of your mod.

That is everything you need to know to create a mod.

In the next step, I show you how to add your own event to the game.

Creating an Event

Here I show you how to create an Event and register it for the game to use.

1. Writing the Event

Here I explain, how you write the body of an event. c!pp

Creating an Event follows multiple Step.

Step 1 ist to write the Event itself.

An Event has to follow the following structure:

```
label event_label (**kwargs):
    $ begin_event(**kwargs);

    // get values

    // get images

    // dialogue or whatever happens during the event

    // stat changes

    $ end_event('new_daytime', **kwargs)
```

So the **label** defines where the event begins.
An event always takes a **kwargs parameter. In it the game delivers all the data and information the game, the event and other functions need to work. So you need to make sure to not overwrite or change values in it haphazardly.

When using the normal value system, to insert data in the event for you to use, there will be no collision since those are stored in a dedicated dict inside the kwargs.

Next is the **begin_event** method.

begin_event(version: str = "1", **kwargs)
This method is called at the start of an event after choices and topics have been chosen in the event.

It prevents rollback to before this method and thus prevents changing choices and topics.
It also starts the Gallery_Manager if the event is not in replay which is used to track and register
the used variables and decisions in the event.

Parameters

1. version: str (Default "1")
   - The version of the event. This is used to identify the event version.

Options

1. no_gallery = True
   - Gallery_Manager will not be initiated and event will not be registered into the gallery

This one has to be called at the beginning of each event. This method does a multitude of tasks to prepare and make the event work.

It starts the Gallery_Manager, which is responsible for properly registering the event and all the used values and decisions into the replay gallery.

By adding a version, you can tell the gallery if the stored data about the event is still valid or not. If the version varies from the one used for the data already stored, the gallery wipes the data about the event from the gallery storage. This is necessary if you change the event enough for the data to not be valid anymore. This is the case for example when removing or adding values or changing the order of the values occurence. To get a more detailed explanation, you can look here.

The begin_event method also blocks the player from rolling back further than the start of an event.

At the end of each event we have the **end_event** method.

end_event(return_type: str = "new_daytime", **kwargs)

This method is called at the end of an event.
It returns to the map overview or calls a new daytime event.
In case of a replay, it returns to the journal.

Parameters:

1. return_type: str (Default "new_daytime")

- The type of the return.
- If return_type is "new_daytime", a new daytime event is called.
- If return_type is "new_day", a new day event is called.
- If return_type is "none", nothing is called.

This one is also important. This method closes the event. In case of a Composite Event, it calls the next Fragment, in case of the event being played as a replay from the gallery, it returns you to the gallery menu and otherwise it returns you to the game.

By default it returns you to the map but progresses the daytime cycle by one.

By using "new_day", you can skip the entire day and by using anything else or "none", the method simply returns and you can use another way to return to the game or do something else.

It is in your responsibility to have the event return to the game properly when not using the end_event method for returning.

In the next Page I show you how to register the event so it can be used by the game.

2. Registering the Event

Here I show you how to register your eventh{pt .}c!pp, so the game is able to call the event and integrate it into it's systems.

On the last page we created the Event **event_label**.

Now before registering, we need to decide, where the Event will be stored.

For that the game uses a class called **EventStorage**. This class registers the events and also provides them when called.
Currently there are 3 types of **EventStorage,** the EventStorage, the TempEventStorage and the FragmentStorage.
We'll ignore the FragmentStorage for now.

A more detailed explanation of the EventStorages follows in a later chapter.

Now back to the question where we want to register the Event. Tha game has multiple locations, for example the school building, the courtyard and so on. Now each of those locations has three Storages. a general one, a TempEventStorage and a dictionary of multipal EventStorages. The difference between those is, that Events registered in the TempEventStorage will only be shown once and then never again in the playthrough, while Events stored in the normal EventStorage can be shown repeatedly.

When a location is entered by the player, the game will first try to call an event from the TempEventStorage. After that it will try to open an Event from the general EventStorage, and after that it will open a Selection Menu, where the player can select the action he want's to execute. So each action is it's own EventStorage.

In this example, I will register the event for the courtyard, and as it should be shown repeatedly, we use the normal Storage.
So the registration will look like this:

```
init 1 python:
    set_current_mod('base')


    event_label_event = Event(3, "event_label")
    courtyard_events["patrol"].add_event(event_label_event)
```

So what do we see here. First we open a python block that runs when the game loads. The **1** shows the position in the load order. When registering events, the position should not be lower than 1.

In the second row we set the current mod. This is important for the game to detect from which mod the registrations come from. It is your responsibility to make sure, that when you register content into the game, that you call that method with your mod key at least once at the beginning of each init block. Make sure that the key is identical to the key you provide in the metadata file.

In row 4 we define the event we wrote in the last page of this manual. It is simply built by initializing the Event class with a priority and the renpy label it has to call.

And in row 5 we register the event. As you can see, we add the event into the EventStorage for the patrol action in the courtyard.

With that the event is sucessfully registered and will now be able to be called by the game. But since we added only the barebone of informations, it willl work, but there will be no image in the replay gallery and there is no limitation when it can be called. With that there is always a chance that it can be called.

A list of all Storages can be found here. And how to add your own storages can be found here.

In the next page we will work on how to set conditions and limitations on when an event is allowed to be called.

Event(priority: int, event: str, *options: Condition | Selector | Option | Pattern, thumbnail: str = "", register_self: bool = True, override_intro: bool = False, override_location: bool = None)

This class represents a callable game event.

This class performs a self-check when initialized and will not be registered when something is not correct. It will also throw an error message in the log stating the problem.

priority

This value represents the way an EventStorage selects the event. There are three priorities ranging from 1 to 3. Some methods that check for or call events can also use priority 0 which then represents all three priorities.

Priority 1 events are high priority, that means the EventStorage elects the first viable Priority 1 Event and calls it. After that event is fnished, the game will return to the map overview.

Priority 2 events are also high priority but lower than 1. An EventStorage will collect all viable Priority 2 events and call them one after another and then return back to the map overview.

Priority 3 events are also referred to regular events. Here the EventStorage will collect all viable Priority 3 events and then select one at random to call it. After that event, the game will return to the map overview.

event

This value is the renpy label the event opens when called.

options

This is an argument list that can contain various numbers of Conditions, Selectors, Options and Patterns. These are used to modifiy an event and to give it more detail and restrictions on when an event is allowed to be called.

After this parameter, all following parameters have to be called by their parameter name

---

register_self

This is an option to disable or enable the event from being registered in the central event collection. This value is True by default, since an event should always be registered in the central event collection since different instances use that collection to recieve the events data. But rarely it is necessary to delay that registration so this option exists.

override_intro

By default all events are blocked from being called during the introduction phase of the game, since there are two free roaming phases, that run on the normal event system. To block the event from being run during those phases, it adds a Condition to itself that blocks the use during the introduction. To prevent that blocking from happening, this parameter can be set True. That would prevent the event from adding the condition and allow the event to be called during the introduction.

override_location

Normally an event gets it's location from the EventStorage it is added to, but in some cases it is needed to override the location. The location value is only important for the replay gallery and does not have any other role that to sort the event into the correct category in the gallery.

3. Conditions for Events

Here we talk about Conditions.c!pp

In this game, Conditions are implemented in a special way. Here the conditions are represented by the Condition-Object. The Condition-Object itself doesn't do much but it serves as base for the different condition types.

The reason Conditions are done this way when fellow coders know there are already boolean operations, is because the conditions have to be checked on demand, so these condition classes help to store the condition and then check on demand if the condition has been fulfilled.

So currently there is a set of different condition types already available. You can find the overview for those here.

So now lets set a condition for our event. On the last Page we used this code to register our event:

```
init 1 python:
    set_current_mod('base')


    event_label_event = Event(3, "event_label")
    courtyard_events["patrol"].add_event(event_label_event)
```

Now lets add a TimeCondition. The TimeCondition is used to for example set the event to be available only at certain times. For example only on Mondays or only during the evening.

Now the updated code looks like this:

```
init 1 python:
    set_current_mod('base')


    event_label_event = Event(3, "event_label", TimeCondition(day = "5-10", month = "2-", daytime = "f"))
    courtyard_events["patrol"].add_event(event_label_event)
```

You see now the added TimeCondition. It has multiple input values like day, month and daytime. This condition now limits the availability of the Event to between the 5th and 10th day of the month for only January and Febuary and only during free time. So you see, you have some freedom

on what you can set for the TimeCondition. The details on what can be used for this Condition can be found below or in the Condition overview.

TimeCondition(blocking: bool = True, **kwargs: str | int)

This condition checks if the current time and date is within the set time limits.

blocking: bool = True

This value sets the visibility of the object including this condition. If blocking is set to True and the condition is not fulfilled, the parent will not be visible. That functionality is only limited for conditions used inside Journal Objects.

**kwargs: str | int

Here he time limits are set using arguments. To use it, set an argument together with the value. Here is an example:

```
TimeCondition(day= "5-10", month = "2-", year = 2024, daytime = "f")
```

Here you see the keys "day", "month", "year" and "daytime" set as the arguments together with their values. This example limits the condition to a time between the 5th and 10th on January of February 2024 during free-time.

You can use the following keys: **day**, **week**, **month**, **year**, **daytime**, **weekday**

For the keys: **day**, **week**, **month**, **year**, **weekday** and **daytime** you can simply set a value as a number or you can set a range.
To set the range, you can either use a plus-symbol at the end (**2+**) to set the value to the number and upwards. A minus-symbol (**2-**) for the number and downwards or a range between two values (**2-5**). The values themselves are always included.
You can also give a set of different values by seperating them with a comma. (**2,5,10**)

You can also mix the set together with ranges, so the following is also possible: **2,5-10,20+**
That would include the values 2, all from 5 to 10, 20 and everything after.

Please make sure to NOT put whitespaces in between.

Some keys also accept special values used to make entering certain values easier. You'll find an overview further below.

Key Overview:

Key	Description	Limit	Special Values

day	Used to limit to a certain day in the month	1 - 28	-
week	Used to limit to within a certain week in the month	1 - 4	-
month	Used to limit to a certain month in the year	1 - 12	-
year	Used to limit to a certain year	2023+	-
daytime	Used to limit to a certain time of the day	1 - 7	• "c" - The time during class (2, 4, 5) • "f" - The free-time (1, 3, 6) • "d" - The entire day except night (1-6) • "n" - The night
weekday	Use to limit to a certain day in the week	1 - 7	• "d" - Monday to Friday (1-5) • "w" - Saturday and Sunday (6, 7)

With that we have now successfully limited the Event, so that it now only shows up during certain times and dates.

On the next Page we talk about how to add images to your event.

4. Data for Events

On this page we talk about how we can generate varying datah{pt .}c!pp to the event so it can then use that data to change itself.

This is useful if for example you want to rotate between characters every time the event is called or if you want the event to play out differently after you hit a certain stat limit. Or something else.

To provide this data I created the Selector class. Like the Condition Class, the Selector class is the base from which different Variants can be created. For this example we will put in a RandomListSelector class. This one takes a random value from a set of values. If we put it in the code example from the last pages, it will look like this:

```
init 1 python:
    set_current_mod('base')


    event_label_event = Event(3, "event_label",
        TimeCondition(day = "5-10", month = "2-", daytime = "f"),
        RandomListSelector("girl", "Aona Komuro", "Lin Kato", "Luna Clark")
    )
    courtyard_events["patrol"].add_event(event_label_event)
```

Now you see the added Selector. I just put it after the Condition. The order of Conditions and Selectors doesn't matter for it to work, as the Event itself just expects a Set of Conditions or Selectors after the event label. The Event will then filter out the correct classes itself.

A Selector always starts with a key, which is used to identify the value later. In the example above it is "**girl**". For this Selector it is then followed by all values from which one will be chosen randomly.

One important thing to know is, that after an Event is called, it will then immediately reroll the values for the Selectors. So if you start an event and the Selector returns "Aona Komuro", it will then immediately reroll the value and provide it for the next time the event is called. This is to lower the performance needed for these functions.

Another important thing is, some Selector types can wok with values that have been set by other Selectors. For that to work, the Selectors can only access values from the Selectors above it, since the values are fetched in order from top to bottom.
This also applies to Conditions, since certain Condition Types access the values provided by the Selectors to check if they are fulfilled.

One example is the ValueCondition. It checks if a value for a certain key is identical to the one provided for the Condition. That would look like this:

```
init 1 python:
    set_current_mod('base')


    event_label_event = Event(3, "event_label",
        TimeCondition(day = "5-10", month = "2-", daytime = "f"),
        RandomListSelector("girl", "Aona Komuro", "Lin Kato", "Luna Clark"),
        ValueCondition("girl", "Lin Kato")
    )
    courtyard_events["patrol"].add_event(event_label_event)
```

You see it uses the key **girl** to identify the value and the checks if the value provided by the RandomListSelector is **Lin Kato**. Only then the Event would be available. As you can also see the Selector is now encased between two Conditions. That again shows that the Conditions and Selectors don't have to be bundled. The order is only important if you want to access the values of other Selectors.

This example now would also prevent the event from ever being shown, since the Event now can only be called when the RandomListSelector randomly picks **Lin Kato**. Since it only rerolls the value on the first registration and then only when the event is called, the value will never change when the event can't be called. So it is now locked. So it is your responsibility to prevent that from happening unintentionally.

RandomListSelector(key: str, *values: Any, realtime: bool = False, alt: Any = None)

This Selector-class selects a random value from a given set of values.

key: str

The key used to later identify the value in the data set provided in the gallery.

*values: Any

A set of values from which one will be selected.

The values can consist of normal values, other Selectors or tuples. Selectors provided as value for this Selector don't need a key, so the key can be an empty string.

The tuples can have multiple forms. An overview of that can be found in the table below:

Tuple	Description	Example
(float, Any)	By setting a float with a value between 0 and 1 you can give the value (**Any**) a set probability of being chosen. All other values without a set probability are spread throughout the remaining percentage. Make sure to give values a probability of at least 1%.	(0.1, "Lin Kato")
(Any, bool | Condition)	By setting a boolean condition or a condition via the Condition-class, you can set certain values (**Any**) to only be considered if the condition is fulfilled.	("Lin Kato", school_level > 2) ("Lin Kato", TimeCondition(daytime = "f"))
(float, Any, bool | Condition)	Probability and Condition can also be used together.	(0.1, "Lin Kato", school_level > 2)

realtime: bool = False

This variable sets if the Selector will roll its value on demand or in preparation for the next call.
If False, it will returned the last rolled value currently stored and then roll a new value for the next call.
If True, it will roll a new value and the return that.

If set to True, it is possible that Conditions that are based on this Selector could show unexpected behaviour.

alt: Any = None

This variable provides an alternative value in case the method is unable to select a value from the set because of unfulfilled conditions.

So we added data to the event, now we have to access it in the event itself.

For that the GalleryManager provides the methods **get_value** and **get_level.** The main method though is **get_value**.
Adding it to the event would look like this:

```
label event_label (**kwargs):
    $ begin_event(**kwargs);

    $ girl = get_value('girl', **kwargs)

    // get images
```

```
    // dialogue or whatever happens during the event

    // stat changes

    $ end_event('new_daytime', **kwargs)
```

Here I took the event from earlier and added the get_value method. This method does multiple things. For one it retrieves the value behind the key "**girl**" which we inserted into the Event with the Selector. The other thing this method does is to automatically register the value for the Replay Gallery. In case you play this event as a replay, this method also provides the value you set in the replay menu.

An important factor here is the order of loading all the values. The order of how you get the values does not have to be the order in how you put the Selector classes in the Event-definition. But the order of the get_value methods is important for the gallery. So if you decide to change the order, you have to register the change with the gallery by providing an updated event version. This tells the gallery that the value order is not valid anymore and thus removes the gallery data for that event. That would look like this:

```
label event_label (**kwargs):
    $ begin_event(version = "2", **kwargs);

    $ new_girl = get_value('new_girl', **kwargs)
    $ girl = get_value('girl', **kwargs)

    // get images

    // dialogue or whatever happens during the event

    // stat changes

    $ end_event('new_daytime', **kwargs)
```

Here you see I added another get_value in front. This would break the gallery data, since it expects the "girl" key and not "new_girl". To tell the gallery, the event has been reworked I added the version argument to **begin_event** and set the version to **2**.
By default the version is set to **1**, so it only has to be set when the event undergoes changes that change the order in which the values are loaded. For that reason you only need to change the version when the value order or the value keys change, since it's only the gallery that needs this information.

Now other than the **get_value** method, there is also the **get_level** method. This one works a little bit different. The usage is pretty much identical, but this method does not require a Selector in the Event definition. The levels of the school, parents, teachers and secretary are always provided with the data send into the event. So you just call the **get_level** method together with the correct key for each of the characters. The possible keys are: **school_level**, **parent_level**, **teacher_level** and **secretary_level**.

In the code it would look like this:

```
label event_label (**kwargs):
    $ begin_event(**kwargs);

    $ girl = get_value('girl', **kwargs)
    $ school_level = get_level('school_level', **kwargs)

    // get images

    // dialogue or whatever happens during the event

    // stat changes

    $ end_event('new_daytime', **kwargs)
```

---

get_value(key: str, alt: Any = None, **kwargs)

This methods retrieves the value from the data provided for the event.

It returns the value and registers it for the replay gallery. In case the method is called while the event is played as a replay, it only retrieves the value.

key: str

This key is used to identify the correct value. The key has to be identical to the key used in the Selector Class during the Event definition.

alt: Any = None

Here you can define a default value that will be returned in case the key cannot be found in the data provided. This default value will NOT be registered into the gallery.

**kwargs

This is the data provided for the event. It has to be the kwargs that comes with the event itself.

get_level(key: str, **kwargs)

This method returns the current level for the defined key. This value is automatically registered in the gallery.

In case the method is called while the event is played as a replay, it only retrieves the value.

key: str

This key is used to identify the correct character object from which to retrieve the level from.

The following keys are valid: **school_level**, **parent_level**, **teacher_level**, **secretary_level**

**kwargs

This is the data provided for the event. It has to be the kwargs that comes with the event itself.

With that you now have the values defined in the Event definition and are now freely usable in the event.

On the next Page we talk about how to add images to your event.

5. Images for Events

Let's add some images to the event! c!pp

Adding images is very similar to adding values. First we must provide the image for that we have to define a Pattern in the Event definition. The Pattern is define with the Pattern class, which takes a key for identifying the correct pattern and the image pattern. The image pattern is just the path starting from the game folder or in your case starting from the root folder of the mod, with the root folder being the folder where your metadata file is located. The Event definition would then for example look like this:

```
init 1 python:
    set_current_mod('base')


    event_label_event = Event(3, "event_label",
        TimeCondition(day = "5-10", month = "2-", daytime = "f"),
        RandomListSelector("girl", "Aona Komuro", "Lin Kato", "Luna Clark"),
        Pattern("main", "/images/events/school building/sb_event_4 <school_level> <girl> <step>.webp"),
        Pattern("main2", "/images/events/school building/sb_event_5 <school_level> <girl>.webp", 'school_level',
'girl'),
    )
    courtyard_events["patrol"].add_event(event_label_event)
```

Here I have added two patterns. First let's look at the Pattern with the key "main".

In Pattern I put "main", which is the key used to identify the correct pattern when showing the images and after that I put in the path, or the pattern. Now I say pattern because you can see in the latter part of the path there are keys put in <>-brackets. **school_level**, **girl_name** and **step**. Now we know **girl**, since it is defined in the Selector directly above, and we know **school_level** since it is always present in the event data. **step** is a new key we haven't seen yet. This key is one of the main functions of these image patterns. The **step** key is used to create a series of images. For example you have 3 images in an event, so you can just name your images "**image 1.webp**", "**image 2.webp**" and "**image 3.webp**". To include all of these images you can use the pattern "**/images/.../image <step>.webp**" and the method later used to access the images will check what images are available and will then provide them for usage.

In a similar way will the method also replace the keys **school_level** and **girl** with the actual values, so the path could then look like this: "**/images/events/school building/sb_event_4 1 Aona Komuro <step>.webp**". Step is still step since it will then be replaced on demand.

Let's look at the pattern with the key "**main2**". It is almost identical to the "**main**"-Pattern except for a little different path, no step key and more keys behind the path. That is another feature these pattern provide. By defining these keys you can set the pattern to also check for images that don't have a value for the corresponding key. The image file would then just have a "**#**" at the keys position instead. In this example you could have an image named: "**sb_event_5 # Aona Komuro 0.webp**" or "**sb_event_5 1 #.webp**" or even "**sb_event_5 # #.webp**". This is particulary useful if you have images where for this example you don't have any girls in the picture, so instead of having the same image three times just with the different names to conform to all possible value combinations, you can just define one image with "**#**" for the key.

But please make sure to only use that feature when it is needed. If there is no image with "**#**", the keys should not be set for ignoring. The method checks for all possible combinations and selects the best one, so while being pretty optimized, it could still have impact on performance.

Pattern(name: str, pattern: str, *alternative_keys: str)

This class provides a way to insert an image patten into the event for it to be used.

name: str

This is the key used for identification.

pattern: str

The image path or pattern.

*alternative_keys: str

(Optional) A set of keys that can be ignored by the patterns.

Now let's use the images in our event. To do that we use the **convert_pattern** or the **show_pattern** method.

The difference between these two is that you use **convert_pattern** for series of images and **show_pattern** for a single image. In our example we have both, so lets add both. Tha would then look something like this:

```
label event_label (**kwargs):
    $ begin_event(**kwargs);


    $ image = convert_pattern("main", **kwargs)
```

```
$ show_pattern("main2", **kwargs)
person "Hi I'm a dialogue text."

$ image.show(0)
person "Oh another image."

$ image.show(1)
person "Wow such simple"

// stat changes

$ end_event('new_daytime', **kwargs)
```

Here you see I used **convert_pattern** to get the pattern with the key "**main**". It is a conversion because that method converts the pattern to an ImageSeries-class. That method comes with a lot of handy functions. One of those is the show method. First things first, the **convert_pattern** method converts the pattern into an ImageSeries-class and returns it, so I then store it in the image variable so I can use it later. Since it is a series of images defined by the step key inside the pattern, I can show the image with the corresponding step number just by calling "**$ image.show(n)**" where n is the number of the image. This class can also show videos, but that comes later.

show_pattern does a similar thing, but instead of creating an ImageSeries it just fills the pattern and directly shows the image.

Maybe you already noticed, but the **get_value** methods are missing here. That is because the **convert_pattern** and the **show_pattern** methods retrieve and register those values themselves, so no need to do that again, unless you want to do other things with these values like changing the dialogue.

Another key that can be used for the pattern I haven't talked about yet is the "**variant**" key. If you add the variant key, you are able to have multiple variants of the same image and the methods will just select one of those at random. You don't have to specify anything. You only have to add the "**variant**" key to the pattern and the methods check for themselves what variants are available. The only thing you have to make sure is, that you have a series of numbers for that starting with 1, since the method starts looking for the variant "1" and then just goes up until it doesn't find a higher variant number.

If you want to override the variant, you can call the show method with the variant argument like this: "**$image.show(0, variant = 1)**".

Be aware that the the order where **convert_pattern** and **show_pattern** occur in the code and the order in which the keys in the pattern occur is also a factor for the order of data stored in the gallery.

---

convert_pattern(pattern_key: str, **kwargs)

Gets a pattern from the event data based on the key and converts it to an ImageSeries class.

pattern_key: str

The key under which the pattern is defined.

**kwargs

This is the data provided for the event. It has to be the kwargs that comes with the event itself.

show_pattern(pattern_key: str, **kwargs)

Gets a pattern from the event data based on the key and converts it to an image and then displays that image.

pattern_key: str

The key under which the pattern is defined.

**kwargs

This is the data provided for the event. It has to be the kwargs that comes with the event itself.

On the last page of this quick manual on how to create an event, I will show you how to manipulate the stats for the different characters.

6. Changing Stats in Events

Let's change some Stats! c!pp

To change stats in the Event there are two methods. one to change stats for the characters and one to change the money.

The first method to change stats for characters in general is **change_stats_with_modifier**. That one is a label though so it has to be called appropriately. That would look like this:

```
label event_label (**kwargs):
    $ begin_event(**kwargs);

    $ image = convert_pattern("main", **kwargs)

    $ show_pattern("main2", **kwargs)
    person "Hi I'm a dialogue text."

    $ image.show(0)
    person "Oh another image."

    $ image.show(1)
    person "Wow such simple"

    call change_stats_with_modifier("school",
        happiness = 2, inhibition = TINY)

    $ end_event('new_daytime', **kwargs)
```

Here you see the **change_stats_with_modifer** label. First I inserted the key "school" for the character object and after that I added a set of stats with the value on how they should be changed. The Happiness-Stat will be increased by 2 and Inhibition by a tiny amount.

To add a bit of variety I added keywords that can be used for the stat changes to change the values a little bit randomly. An overview over these keywords can be found in the method description below.

But what is that modifier in the method name? The game has a modifier system, where modifier can be created to change stat changes. This enables you to create buffs or debuffs. The income of the school is also managed by these modifiers. Documentation on how to use them can be found here.

---

change_stats_with_modifier(char_name: str, collection: str = 'default', **kwargs)

This method is used to change the stats for the different characters.

char_name: str

The key for the corresponding character.

The following keys are available: **school**, **teacher**, **parent**, **secretary**

collection: str = 'default'

The collection of modifiers to be used.

**kwargs

A set of stats together with the value changes. The values can be numbers or keywords defining a range of possible values.

When using keywords, a random value inside a specified range will be selected for the value change. An overview over the available keywords can be found in the table below:

Keyword	**Range**
TINY	0.1 - 0.3
SMALL	0.3 - 0.5
MEDIUM	0.5 - 1.0
LARGE	1.0 - 3.0
GIANT	3.0 - 7.0

By putting "**DEC_**" in front of the keywords, the values are inverted to their negative counterparts, so "**DEC_TINY**" for example represents a range from -0.3 to -0.1.

To change the money we use the method **change_money_with_modifier**. That is is much easier, you just call it and add how you want to change the value. That would look like this:

```
label event_label (**kwargs):
    $ begin_event(**kwargs);

    $ image = convert_pattern("main", **kwargs)

    $ show_pattern("main2", **kwargs)
    person "Hi I'm a dialogue text."

    $ image.show(0)
    person "Oh another image."

    $ image.show(1)
    person "Wow such simple"

    call change_money_with_modifier(100)

    $ end_event('new_daytime', **kwargs)
```

I just made it add 100 money.

---

change_money_with_modifier(value: int, collection: str = "default")

This method is used to change the money.

value: int

The amount on how much to change the money.

collection: str = 'default'

The collection of modifiers to be used.

That is basically everything you need to know to create basic events. Of course you should take a look in the renpy documentation on how to add dialogue, but you can pretty much see it in the examples above or check out the book about dialogue.

The system also has some more advanced stuff which you can find under Advanced Events.

Example

Here is an example on how a full file with the event and its definition could look like.c!pp

```
init 1 python:
    set_current_mod('mod')

    event_label_event = Event(3, "event_label",
        TimeCondition(day = "5-10", month = "2-", daytime = "f"),
        RandomListSelector("girl", "Aona Komuro", "Lin Kato", "Luna Clark"),
        Pattern("main", "/images/events/school building/sb_event_4 <school_level> <girl> <step>.webp"),
        Pattern("main2", "/images/events/school building/sb_event_5 <school_level> <girl>.webp", 'school_level',
'girl'),
    )
    courtyard_events["patrol"].add_event(event_label_event)

label event_label (**kwargs):
    $ begin_event(**kwargs);

    $ image = convert_pattern("main", **kwargs)

    $ show_pattern("main2", **kwargs)
    person "Hi I'm a dialogue text."

    $ image.show(0)
    person "Oh another image."

    $ image.show(1)
    person "Wow such simple"

    call change_stats_with_modifier("school",
        happiness = 2, inhibition = TINY)

    call change_money_with_modifier(100)

    $ end_event('new_daytime', **kwargs)
```