

4. Data for Events

On this page we talk about how we can generate varying data 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 work 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.

Revision #4

Created 30 September 2024 13:47:06 by SulT-Ji

Updated 1 October 2024 14:49:55 by SulT-Ji