Gillcup Documentation

Release 0.2.1

Petr Viktorin

2014-11-19

Contents

This is a technical documentation of the Gillcup library. For a tutorial and flashy demo, you should look at Gillcup Graphics.

Contents:

CHAPTER 1

Introduction

Gillcup is a 2D animation library.

It is intended for both scripted and interactive animations: in the former, the entire animation is known ahead of time, and rendered as fast as possible; the latter is gnerally tied to a system clock, and can be influenced by user input.

The Gillcup core is only concerned with animations; it's not tied to any particular graphics library. The gillcup_graphics package includes a more accessible demo and a tutorial.

1.1 Version warning

This is version 0.2. The API **will** change significantly in version 0.3. Please make sure you specify `gillcup < 0.3` in your setup/requirements file.

1.2 The Project

Gillcup is a MIT-licensed, Github-hosted project striving to uphold best practices of the Python craft, from PEP8 to semantic versioning. Please report any bugs, style issues and suggestions on the bug tracker!

CHAPTER 2

Module Reference

Gillcup, a Python animation library

Gillcup provides a number of modules:

2.1 gillcup.clock

Gillcup's Clock Class

In Gillcup, animation means two things: running code at specified times, and changing object properties with time.

You will notice that the preceding sentence mentions time quite a lot. But what is this time?

You could determine time by looking at the computer's clock, but that would only work with real-time animations. When you'd want to render a movie, where each frame takes 2 seconds to draw and there are 25 frames per second, you'd be stuck. That's why Gillcup introduces a flexible source of time, the Clock, which keeps track of time and schedules actions.

Time is measured in "time units". What a time unit means is entirely up to the application – it could be seconds, movie/simulation frames, etc.

class `gillcup.`**`Clock`**

Keeps track of time and schedules events.

Attributes:

`time`

The current time on the clock. Never assign to it directly; use `advance()` instead.

Animated Properties:

`speed`

Speed of the clock.

When calling update(), the interval is multiplied by this value.

The speed is an AnimatedProperty. When changing, beware that it is only checked when advance() is called or when a scheduled action is run, so speed animations will be only approximate. For better accuracy, call `advance()` with small *dt*, or schedule a periodic dummy action at small inervals.

Basic methods:

`advance`(*dt*)

Call to advance the clock's time

Steps the clock dt units to the future, pausing at times when actions are scheduled, and running them.

Attempting to move to the past (dt<0) will raise an error.

schedule (*action*, *dt=0*)
Schedule an action to be run "dt" time units from the current time

Scheduling is stable: if two things are scheduled for the same time, they will be called in the order they were scheduled.

Scheduling an action in the past (dt<0) will raise an error.

If the scheduled callable has a "schedule_callback" method, it will be called with the clock and the time it's been scheduled at.

Update function registration:

schedule_update_function (*function*)
Schedule a function to be called every time the clock advances

Then function will be called a lot, so it shouldn't be very expensive.

Only a weak reference is made to the function, so the caller should ensure another reference to it is retained as long as it should be called.

unschedule_update_function (*function*)
Unschedule a function scheduled by *schedule_update_function*

class gillcup.**Subclock** (*parent*, *speed=1*)
A Clock that advances in sync with another Clock

A Subclock advances whenever its *parent* clock does. Its *speed* attribute specifies the relative speed relative to the parent clock. For example, if speed==2, the subclock will run twice as fast as its parent clock.

Unlike clocks synchronized via actions or update functions, the actions scheduled on a parent Clock and all subclocks are run in the correct sequence, with all clocks at the correct times when each action is run.

2.2 gillcup.actions

Gillcup Actions

Although arbitrary callables can be scheduled on a Gillcup `Clock`, one frequently schedules objects that are specifically made for this purpose. Using `gillcup.Action` allows one to chain actions together in various ways, allowing the developer to create complex effects.

class gillcup.**Action** (*clock=None*, *dt=0*)
A chainable "event" designed for being scheduled.

As any callable, an Action can be scheduled on a clock, either by `schedule()`, or by chaining, or, as a shortcut, directly from the constructor with the *clock* and *dt* arguments. Each Action may only be scheduled *once*.

Other actions may be chained to an Action, that is, scheduled to run at some time after the Action is run.

Some Actions may represent a time interval or process rather than a discrete point in time. In these cases, chained actions are run after the interval is over or the process finishes.

Actions may be combined to form larger structures using *helper Action subclasses* as building blocks. As a shorthand, the following operators are available:

- + creates a `Sequence` of actions; one is run after the other.

- `|` creates a `Parallel` construct: all actions are started at once.

The `chain()` method and operators can be used with Actions, or regular callables (which are wrapped in `FunctionCaller`), or with numbers (which create corresponding `delays`), or with iterables (which get wrapped in `Process`), or with None (which coerces into a no-op Action).

chain(*action*, *dt=0*)
Schedule an action to be scheduled after this Action

The `action` argument may be a callable, number, or None, and is wrapped by an Action if necessary. See `__init__` for more details.

The `dt` argument can be given to delay the chained action by the specified time.

If this Action has already been called, the chained action is scheduled immediately *dt* units after the current time. To prevent or modify this behavior, the caller can check the `chain_triggered` attribute.

Returns the chained action.

`gillcup.Action.`**chain_triggered**
Set to true when this Action is finished, i.e. its chained actions have been triggered.

Overridable methods:

__call__()
Run this action.

Subclasses that represent discrete moments in time should call the superclass implementation when they are finished running.

Subclasses that represent time intervals (there's a delay between the moment they are called and when they trigger chained actions) should call `expire()` when they are called, and `trigger_chain()` when they're done.

Methods useful for subclasses:

expire()
Marks the Action as run.

Subclasses must call this method at the start of `__call__()`.

trigger_chain()
Schedule the chained actions.

Subclasses must call this method after the Action runs; see `__call__()`.

classmethod coerce(*value*)
Coerce value into an action.

Wraps functions in FunctionCallers, numbers in Delays, and None in a no-op.

2.2.1 Building blocks for complex actions

class `gillcup.actions.`**FunctionCaller**(*function*, **args*, ***kwargs*)
An Action that calls given *function*, passing *args* and *kwargs* to it

function can be any callable.

class `gillcup.actions.`**Delay**(*time*, ***kwargs*)
An Action that triggers chained actions after a given delay

The *kwargs* are passed to `gillcup.Action`'s initializer.

class gillcup.actions.**Sequence**(**actions*, ***kwargs*)
An Action that runs a series of Actions one after the other

Actions chained to a Sequence are triggered after the last Action in the sequence.

The *kwargs* are passed to gillcup.Action's initializer.

class gillcup.actions.**Parallel**(**actions*, ***kwargs*)
Starts the given Actions, and triggers chained ones after all are done

That is, after all the given actions have triggered their chained actions, Parallel triggers its own chained actions.

The *kwargs* are passed to gillcup.Action's initializer.

class gillcup.actions.**Process**(*iterable*, ***kwargs*)
Wraps the given iterable

When triggered, takes an item from the iterable and schedules it, then chains the scheduling of the next item, and so on. When the underlying iterator is exhausted, chained actions are run.

The items in the underlying iterable can be callables, numbers or other iterables, as for Action's + and | operators.

The *kwargs* are passed to gillcup.Action's initializer.

See process_generator() for a simple way to create Processes.

gillcup.actions.**process_generator**(*func*)
Decorator for creating Processes

Used as a decorator on a generator function, it allows writing in a declarative style instead of callbacks, with yield statements for "asynchronousness".

2.3 gillcup.properties

Gillcup's Animated Properties

To animate Python objects, we need to change values of their attributes over time. There are two kinds of changes we can make: *discrete* and *continuous*. A discrete change happens at a single point in time: for example, an object is shown, some output is written, a sound starts playing. Actions are used for effecting discrete changes.

Continuous changes happen over a period of time: an object smoothly moves to the left, or a sound fades out. These changes are made by animating special properties on objects, using Animation classes on so-called animated properties.

Under the hood, Gillcup uses Python's descriptor interface to provide efficient animated properties.

Assigment to an animated attribute causes the property to get set to the given value and cancels any running animations on it.

class gillcup.**AnimatedProperty**(*default*, *docstring=None*)
A scalar animated property

The idiomatic way to define animated properties is as follows:

```
class Tone(object):
    pitch = AnimatedProperty(440)
    volume = AnimatedProperty(0)
```

Now, Tone instances will have *pitch* and *volume* set to the corresponding defaults, and can be animated.

The **docstring** argument becomes the property's __doc__ attribute.

adjust_value(*values*)
Convert an animation's `*args` values into a property value

For scalar properties, this converts a 1-tuple into its only element

tween_values(*function*, *parent_value*, *value*)
Call a scalar tween function on two values.

class gillcup.**TupleProperty**(**default*, ***kwargs*)
A tuple animated property

Iterating the TupleProperty itself yields sub-properties that can be animated individually. The intended idiom for declaring a tuple property is:

```
x, y, z = position = TupleProperty(0, 0, 0)
```

adjust_value(*value*)
Convert an animation's `*args` values into a property value

For tuple properties, return the tuple unchanged

tween_values(*function*, *parent_value*, *value*)
Call a scalar tween function on two values.

Calls the function on corresponding pairs of elements, returns the tuple of results

class gillcup.properties.**ScaleProperty**(*num_dimensions*, ***kwargs*)
A TupleProperty used for scales or sizes in multiple dimensions

It acts as a regular TupleProperty, but supports scalars or short tuples in assignment or animation.

Instead of a default value, __init__ takes the number of dimensions; the default value will be `(1,) * num_dimensions`.

If a scalar, or a tuple with only one element, is given, the value is repeated across all dimensions. If another short tuple is given, the remaining dimensions are set to 1.

For example, given:

```
width, height, length = size = ScaleProperty(3)
```

the following pairs are equivalent:

```
obj.size = 2
obj.size = 2, 2, 2

obj.size = 2, 3
obj.size = 2, 3, 1

obj.size = 2,
obj.size = 2, 2, 2
```

Similarly, `Animation(obj, 'size', 2)` is equivalent to `Animation(obj, 'size', 2, 2, 2)`.

adjust_value(*value*)
Expand the given tuple or scalar to a tuple of len=num_dimensions

class gillcup.properties.**VectorProperty**(*num_dimensions*, ***kwargs*)
A TupleProperty used for vectors

It acts as a regular TupleProperty, but supports short tuples in assignment or animation by setting all remaining dimensions to 0.

Instead of a default value, __init__ takes the number of dimensions; the default value will be `(0,) * num_dimensions`.

For example, given:

```
x, y, z = position = VectorProperty(3)
```

the following pairs are equivalent:

```
obj.position = 2, 3
obj.position = 2, 3, 0

obj.position = 2,
obj.position = 2, 0, 0
```

Similarly, `Animation(obj, 'position', 1, 2)` is equivalent to `Animation(obj, 'position', 1, 2, 0)`.

adjust_value(*value*)
Expand the given tuple to the correct number of dimensions

2.4 gillcup.animation

Gillcup's Animation classes

Animations are `Actions` that modify `animated properties`. To use one, create it and schedule it on a Clock. Once an animation is in effect, it will smoothly change a property's value over a specified time interval.

The value is computed as a tween between the property's original value and the Animation's **target** value. The tween parameters can be set by the **timing** and **easing** keyword arguments.

The "original value" of a property is not fixed: it is whatever the value would have been if this animation wasn't applied (in other words, it's determined by the `effect` that was originally on the property). Also, if you set the **dynamic** argument to Animation, the animation's *target* becomes an `AnimatedProperty`. Animating these allows one to create very complex effects in a modular way.

class `gillcup.`**Animation**(*instance*, *property_name*, **target*, ***kwargs*)
An object that modifies an AnimatedProperty based on Clock time

Positional init arguments:

Parameters

- **instance** – The object whose property is animated
- **property_name** – Name of the animated property
- **target** – Value at which the animation should arrive (tuple properties accept more arguments, i.e. `Animation(obj, 'position', 1, 2, 3)`)

Keyword init arguments:

Parameters

- **time** – The duration of the animation
- **delay** – Delay between the time the animation is scheduled and its actual start
- **timing** – A function that maps global time to animation's time.

  Possible values:

  - `None`: normalizes time so that 0 corresponds to the start of the animation, and 1 to the end (i.e. start + *time*); clamps to [0, 1]
  - `'infinite'`: same as above, but doesn't clamp: the animation goes forever on (in both directions; it only starts to take effect when it's scheduled, but a *delay* can cause negative local times). The animation's time is normalized to 0 at the start and 1 at start + *time*.
  - `'absolute'`: the animation is infinite, with the same speed as with the 'infinite' option, but zero corresponds to the clock's zero. Useful for synchronized periodic animations.
  - *function(time, start, duration)*: apply a custom function
- **easing** – An easing function to use. Can be either a one-argument function, or a dotted name which is looked up in the `gillcup.easing` module.
- **dynamic** – If true, the **target** atribute becomes an AnimatedProperty, allowing for more complex animations.

Note: In order to conserve resources, ordinary Animations are released (replaced by a simple `ConstantEffect`) when they are "done". Arguments such as `timing`, or the `Add` or `Multiply` animation subclasses, which allow the value to be modified after the `time` elapses, turn this behavior off by setting the `dynamic` attibute to true.

When subclassing Animation, remember to do the same if your subclass needs to change its value after `time` elapses. This includes cases where the value depends on the value of the previous (parent) animation.

---

class `gillcup.animation.`**`Add`**(*instance*, *property_name*, **target*, ***kwargs*)
An additive animation: the target value is added to the original

class `gillcup.animation.`**`Multiply`**(*instance*, *property_name*, **target*, ***kwargs*)
A multiplicative animation: target value is multiplied to the original

class `gillcup.animation.`**`Computed`**(*instance*, *property_name*, *func*, ***kwargs*)
A custom-valued animation: the target is computed by a function

Pass a **func** keyword argument with the function to the constructor.

The function will get one argument: the time elapsed, normalized by the animation's *timing* function.

2.5 gillcup.effect

Effect base & helper classes

The Effect is the base class that modify an AnimatedProperty. `Animation` is Effect's most important subclass.

Each Effect can be applied to one or more properties on one or more objects. The value of these properties is then provided by the Effect's `value` property.

class `gillcup.`**`Effect`**
Object that changes an AnimatedProperty

Effects should have a *value* attribute that provides a value for the property.

`get_replacement`()
Return an equivalent effect

When it's sure that the effect's value won't change any more, this method can return a `ConstantEffect` to free resources.

`apply_to`(*instance*, *property_name*)
Apply this effect to an `instance`'s AnimatedProperty

class gillcup.**ConstantEffect**(*value*)
An Effect that provides a constant value

2.6 gillcup.easing

The easing module defines a number of functions usable in gillcup.Animation.

The functions are partly based on Robert Penner's Motion, Tweening, and Easing, and on Qt's QEasingCurve. See their pages for more background.

Each of the functions defined here can be used directly for an "ease in" animation (one that speeds up over time). For other types, use attributes: **out** (slows down over time), **in_out** (speeds up, then slows down), and **out_in** (slows down, then speeds up). The ease-in is also available in **in_**. For example, gillcup.easing.quadratic.in_out is a nice natural-looking tween.

2.6.1 Polynomial easing functions

gillcup.easing.**linear**(*t*)
Linear interpolation

t → t

gillcup.easing.**quadratic**(*t*)
Quadratic easing

t → t**2

gillcup.easing.**cubic**(*t*)
Cubic easing

t → t**3

gillcup.easing.**quartic**(*t*)
Quartic easing

t → t**4

gillcup.easing.**quintic**(*t*)
Quintic easing

t → t**5

2.6.2 Other simple easing functions

gillcup.easing.**sine**(*t*)
Sinusoidal easing

Quarter of a cosine wave

gillcup.easing.**exponential**(*t*)
Exponential easing

gillcup.easing.**circular**(*t*)
Circular easing

2.6.3 Easing factories

gillcup.easing.**elastic**(*period*, *amplitude=1*)
Elastic easing factory

gillcup.easing.**overshoot**(*amount*)
Overshoot easing factory

gillcup.easing.**bounce**(*amplitude*)
Bounce easing factory

2.6.4 Helpers for creating new easing functions

gillcup.easing.**easefunc**(*func*)
Decorator for easing functions.

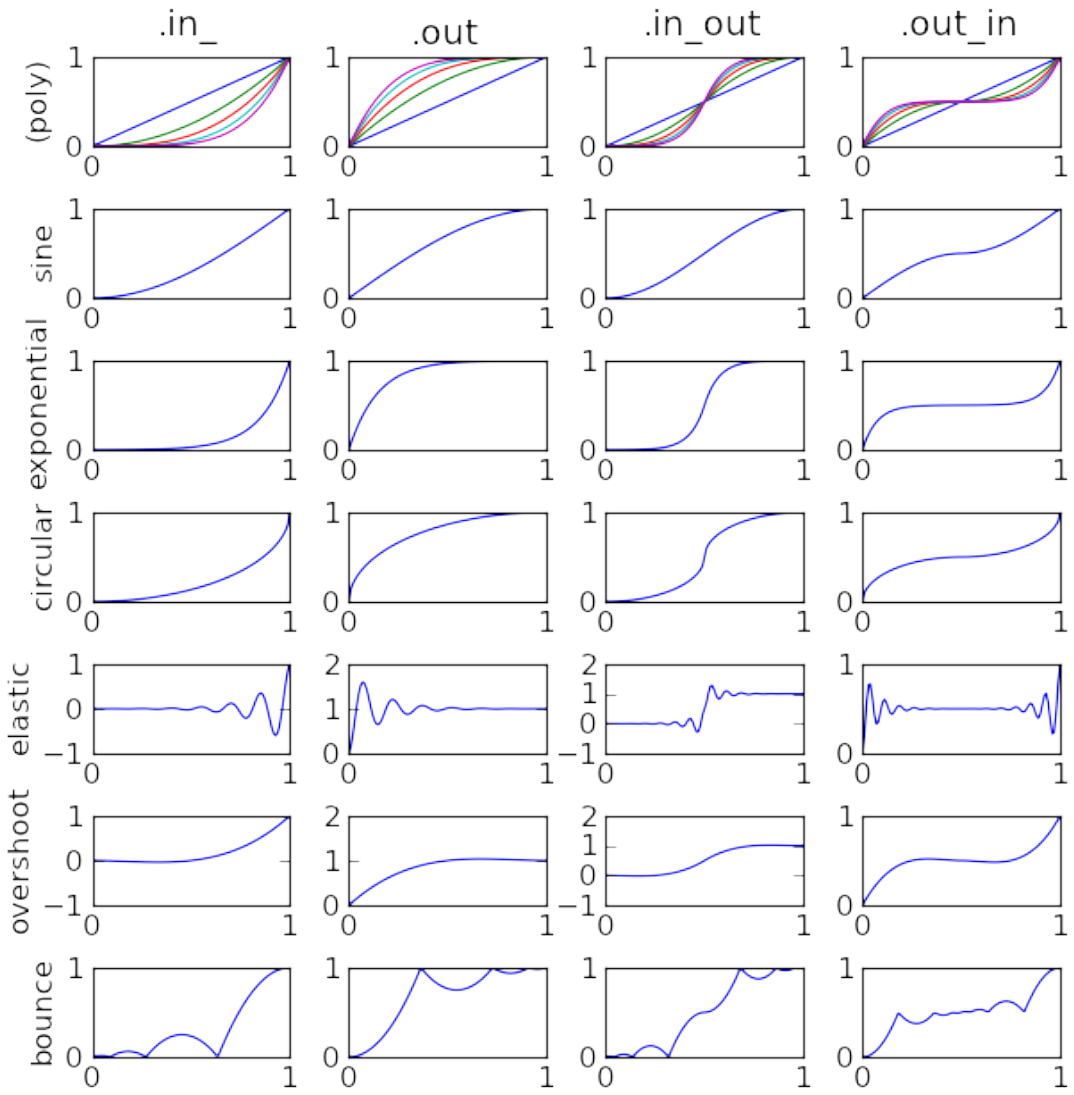
Adds the **in_**, **out**, **in_out** and **out_in** attributes to an easing function.

gillcup.easing.**normalized**(*func*)
Decorator that normalizes an easing function

Normalizing is done so that func(0) == 0 and func(1) == 1.

2.6.5 Graph

For some visual reference, here are the graphs of the various functions in this module.



The graph can be generated by running this module directly (i.e. by `python -m gillcup.easing`). If a command-line argument is given, the graph will be saved to the given file, otherwise it will be displayed. You'll need to install matplotlib to create the graph.

The most interesting classes of each module are exported directly from the gillcup package:

- `Clock` (from `gillcup.clock`)
- `Subclock` (from `gillcup.clock`)
- `Action` (from `gillcup.actions`)
- `AnimatedProperty` (from `gillcup.properties`)
- `TupleProperty` (from `gillcup.properties`)
- `Animation` (from `gillcup.animation`)

- Effect (from gillcup.effect)
- ConstantEffect (from gillcup.effect)

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

g

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

Q

S

T

U

V