Keeping track of time

pyglet's `clock` module allows you to schedule functions to run periodically, or for one-shot future execution.

Calling functions periodically

As discussed in the The application event loop section, pyglet applications begin execution by entering into an application event loop:

```
pyglet.app.run()
```

Once called, this function doesn't return until the application windows have been closed. This may leave you wondering how to execute code while the application is running.

Typical applications need to execute code in only three circumstances:

- A user input event (such as a mouse movement or key press) has been generated. In this case the appropriate code can be attached as an event handler to the window.
- An animation or other time-dependent system needs to update the position or parameters of an object. We'll call this a "periodic" event.
- A certain amount of time has passed, perhaps indicating that an operation has timed out, or that a dialog can be automatically dismissed. We'll call this a "one-shot" event.

To have a function called periodically, for example, once every 0.1 seconds:

```
def update(dt):
    # ...

pyglet.clock.schedule_interval(update, 0.1)
```

The *dt*, or *delta time* parameter gives the number of "wall clock" seconds elapsed call of this function, (or the time the function was scheduled, if it's the first perio latency, load and timer imprecision, this might be slightly more or less than the requested

interval. Please note that the *dt* parameter is always passed to scheduled functions, so be sure to expect it when writing functions even if you don't need to use it.

Scheduling functions with a set interval is ideal for animation, physics simulation, and game state updates. pyglet ensures that the application does not consume more resources than necessary to execute the scheduled functions on time.

Rather than "limiting the frame rate", as is common in other toolkits, simply schedule all your update functions for no less than the minimum period your application or game requires. For example, most games need not run at more than 60Hz (60 times a second) for imperceptibly smooth animation, so the interval given to `schedule_interval()` would be `1/60.0` (or more).

If you are writing a benchmarking program or otherwise wish to simply run at the highest possible frequency, use *schedule*. This will call the function as frequently as possible (and will likely cause heavy CPU usage):

```
def benchmark(dt):
    # ...

pyglet.clock.schedule(benchmark)
```

Note

By default pyglet window buffer swaps are synchronised to the display refresh rate, so you may also want to disable vsync if you are running a benchmark.

For one-shot events, use `schedule_once()`:

```
def dismiss_dialog(dt):
    # ...

# Dismiss the dialog after 5 seconds.
pyglet.clock.schedule_once(dismiss_dialog, 5.0)
```

To stop a scheduled function from being called, including cancelling a periodic function, use `pyglet.clock.unschedule()`. This could be useful if you want to start running a function on schedule when a user provides a certain input, and then unschedule it when another input is received.

Sprite movement techniques

As mentioned above, every scheduled function receives a *dt* parameter, giving the actual "wall clock" time that passed since the previous invocation. This parameter can be used for numerical integration.

For example, a non-accelerating particle with velocity `v` will travel some distance over a change in time `dt`. This distance is calculated as `v * dt`. Similarly, a particle under constant acceleration `a` will have a change in velocity of `a * dt`.

The following example demonstrates a simple way to move a sprite across the screen at exactly 10 pixels per second:

```
sprite = pyglet.sprite.Sprite(image)
sprite.dx = 10.0

def move_sprite(dt):
    sprite.x += sprite.dx * dt

pyglet.clock.schedule_interval(move_sprite, 1/60.0)  # update at 60Hz
```

This is a robust technique for simple sprite movement, as the velocity will remain constant regardless of the speed or load of the computer.

Some examples of other common animation variables are given in the table below.

Animation parameter	Distance	Velocity
Rotation	Degrees	Degrees per second
Position	Pixels	Pixels per second
Keyframes	Frame number	Frames per second

Displaying the frame rate

A simple way to profile your application performance is to display the frame rate while it is running. Printing it to the console is not ideal as this will have a severe impact on performance. pyglet provides the `FPSDisplay` class for displaying the frame rate with very little effort:

```
fps_display = pyglet.window.FPSDisplay(window=window)

@window.event
def on_draw():
    window.clear()
    fps_display.draw()
```

By default the frame rate will be drawn in the bottom-left corner of the window in a semi-translucent large font. See the `FPSDisplay` documentation for details on how to customise this, or even display another clock value (such as the current time) altogether.

User-defined clocks

The default clock used by pyglet uses the system clock to determine the time (i.e., `time.time()`). Separate clocks can be created, however, allowing you to use another time source. This can be useful for implementing a separate "game time" to the real-world time, or for synchronising to a network time source or a sound device.

Each of the `clock_*` functions are aliases for the methods on a global instance of `Clock`. You can construct or subclass your own `Clock`, which can then maintain its own schedule and framerate calculation. See the class documentation for more details.