# asciimatics Documentation

***Release 1.13.1***

**Peter Brittain**

**Apr 23, 2022**

# Contents

Contents:

# CHAPTER 1

# Introduction

Asciimatics is a package to help people create simple ASCII animations on any platform. It is licensed under the Apache Software Foundation License 2.0.

## 1.1 Why?

Why not? It brings a little joy to anyone who was programming in the 80s... Oh and it provides a single cross-platform Python class to do all the low-level console function you could ask for, including:

- Coloured/styled text - including 256 colour terminals
- Cursor positioning
- Keyboard input (without blocking or echoing)
- Mouse input (terminal permitting)
- Detecting and handling when the console resizes
- Screen scraping

In addition, it provides some simple, high-level APIs to provide more complex features including:

- Anti-aliased ASCII line-drawing
- Image to ASCII conversion - including JPEG and GIF formats
- Many animation effects - e.g. sprites, particle systems, banners, etc.
- Various widgets for text UIs - e.g. buttons, text boxes, radio buttons, etc.

Currently this API has been proven to work on CentOS 6 & 7, Raspbian (i.e. Debian wheezy), Ubuntu 14.04, Windows 7, 8 & 10 and OSX 10.11, though it should also work for any other platform that provides a working curses implementation.

## 1.2 Installation

Asciimatics supports Python versions 2 & 3. For a list of the precise list of tested versions, see here.

To install asciimatics, simply install with *pip*. You can get it from here and then just run:

```
$ pip install asciimatics
```

This should install all your dependencies for you. If you don't use pip or it fails to install them, you can install the dependencies directly using the packages listed in requirements.txt. Additionally, Windows users will need to install *pywin32*.

## 1.3 Quick start guide

Once you have installed asciimatics as per the instructions above, simply create a `Screen`, put together a `Scene` using some `Effect` objects and then get the Screen to play it. An Effect will typically need to display some pre-formatted text. This is usually provided by a `Renderer`. For example:

```
from asciimatics.screen import Screen
from asciimatics.scene import Scene
from asciimatics.effects import Cycle, Stars
from asciimatics.renderers import FigletText

def demo(screen):
    effects = [
        Cycle(
            screen,
            FigletText("ASCIIMATICS", font='big'),
            screen.height // 2 - 8),
        Cycle(
            screen,
            FigletText("ROCKS!", font='big'),
            screen.height // 2 + 3),
        Stars(screen, (screen.width + screen.height) // 2)
    ]
    screen.play([Scene(effects, 500)])

Screen.wrapper(demo)
```

# CHAPTER 2

# Contributing

## 2.1 Getting started

So you want to join in? Great! There's a few ground rules…

1. Before you do anything else, read up on the design.
   - You should find all general background in these 4 classes: `Screen`, `Scene`, `Effect` and `Renderer`.
   - You will find more details on TUIs in these 3 classes: `Frame`, `Layout` and `Widget`.
2. If writing a new Effect, consider why it can't be handled by a combination of a new Renderer and the `Print` Effect. For example, dynamic Effects such as `Snow` depend on the current Screen state to render each new image.
3. Go the extra yard. This project started on a whim to share the joy of someone starting out programming back in the 1980s. How do you sustain that joy? Not just by writing code that works, but by writing code that other programmers will admire.
4. Make sure that your code is PEP-8 compliant. Tools such as flake8 and pylint or editors like pycharm really help here.
5. Please run the existing unit tests against your new code to make sure that it still works as expected. I normally use nosetests to do this. In addition, if you are adding significant extra function, please write some new tests for your code.

If you're not quite sure about something, feel free to join us at https://gitter.im/asciimatics/Lobby and share your ideas.

When you've got something you're happy with, please feel free to submit a pull request at https://github.com/peterbrittain/asciimatics/issues.

## 2.2 Building The Documentation

Install the dependencies and build the documentation with:

```
$ pip install -r requirements/dev.txt
$ cd doc && cp source/conf_orig.py source/conf.py
$ ./build.sh
```

You can then view your new shiny documentation in the `build` folder.

## 2.3 Running The Tests

Install the dependencies and run the tests with the following:

```
$ pip install -r requirements/dev.txt
$ nosetests
```

On most systems this will avoid running tests that require a Linux TTY. If you are making changes to the Screen, you must also run the TTY tests. You can force that on a Linux box using the following:

```
$ FORCE_TTY=Y nosetests
```

The reason for this split is that you only typically get a TTY on a live interactive connection to your terminal. This means you should always be able to run the full suite manually. However, many CI systems do not provide a valid TTY and so these tests regularly fail on various build servers. Fortunately, Travis provides a working TTY and so we enable the full suite of tests on any check-in to master.

CHAPTER 3

# Basic Input/Output

## 3.1 Creating a Screen

The starting point for any asciimatics program is the `Screen` object. It can most easily be obtained from the `wrapper()` static method. This will handle all the necessary initialization for your environment and pass the constructed Screen into the specified function. For example:

```
from asciimatics.screen import Screen
from time import sleep

def demo(screen):
    screen.print_at('Hello world!', 0, 0)
    screen.refresh()
    sleep(10)

Screen.wrapper(demo)
```

You can also use the `ManagedScreen` class as a function decorator to achieve the same thing as the above. For example:

```
from asciimatics.screen import ManagedScreen
from asciimatics.scene import Scene
from asciimatics.effects import Cycle, Stars
from asciimatics.renderers import FigletText

@ManagedScreen
def demo(screen=None):
    screen.print_at('Hello world!', 0, 0)
    screen.refresh()
    sleep(10)

demo()
```

Or you can also use it as a context manager (i.e. using the *with* keyword). For example:

```
from asciimatics.screen import ManagedScreen
from asciimatics.scene import Scene
from asciimatics.effects import Cycle, Stars
from asciimatics.renderers import FigletText

def demo():
    with ManagedScreen() as screen:
        screen.print_at('Hello world!', 0, 0)
        screen.refresh()
        sleep(10)

demo()
```

If you need more control than this allows, you can fall back to using `open()`, but then you have to call `close()` before exiting your application to restore the environment.

## 3.2 Output

Once you have a Screen, you probably want to ensure that it is clear before you do anything. To do this call `clear()`. Now that it's blank, the simplest way to output text is using the `print_at()` method. This allows you to place a string at a desired location in a specified colour. The coordinates are zero-indexed starting at the top left of the screen and move down and right, so the example above displays *Hello world!* at (0, 0) which is the top left of the screen.

### 3.2.1 Colours

There is a long history to terminals and this is no more obvious than when it comes to colours. Original terminals had limited colours, and so used attributes to change the format, using effects like bold, underline and reverse video. As time wore on, more colours were added and you can get full 24 bit colour on some terminals.

For now, asciimatics limits itself to a maximum of the 256 colour palette. You can find how many colours your terminal supports by looking at the `colours` property. These days most terminals will support a minimum of 8 colours. These are defined by the *COLOUR_xxx* constants in the Screen class. The full list is as follows:

```
COLOUR_BLACK = 0
COLOUR_RED = 1
COLOUR_GREEN = 2
COLOUR_YELLOW = 3
COLOUR_BLUE = 4
COLOUR_MAGENTA = 5
COLOUR_CYAN = 6
COLOUR_WHITE = 7
```

These should always work for you as background and foreground colours (even on Windows). For many systems you can also use the attributes (see later) to double the number of foreground colours.

If you have a display capable of handling more than these (e.g. 256 colour xterm) you can use the indexes of the colours for that display directly instead. For a full list of the colour indeces, look here.

When creating effects that use these extra colours, it is recommended that you also support a reduced colour mode, using just the 8 common colours. For an example of how to do this, see the `Rainbow` class.

Finally, some terminals have the concept of a default colour. These can often have special attributes that are otherwise impossible to set in a terminal - e.g. transparency. If your terminal supports these you can use the `COLOUR_DEFAULT` setting to use them. If not supported, asciimatics will treat them as a black background and white foreground.

### 3.2.2 Attributes

Attributes are a way of modifying the displayed text in some basic ways that early hardware terminals supported before they had colours. Most systems don't use hardware terminals any more, but the concept persists in all native console APIs and so is also used here.

Supported attributes are defined by the *A_xxx* constants in the Screen class. The full list is as follows:

```
A_BOLD = 1
A_NORMAL = 2
A_REVERSE = 3
A_UNDERLINE = 4
```

Most systems will support bold (a.k.a bright), normal and reverse attributes. Others are capable of more, but you will have difficulties using them in a cross-platform manner and so they are deprecated. The attribute is just another parameter to *print_at*. For example:

```
# Bright green text
screen.print_at('Hello world!', 0, 0, COLOUR_GREEN, A_BOLD)
```

### 3.2.3 Multicoloured strings

If you want to do something more complex, you can use the `paint()` method to specify a colour map for each character to be displayed. This must be a list of colour/attribute values (tuples or lists) that is at least as long as the text to be displayed. This method is typically used for displaying complex, multi-coloured text from a Renderer. See *Animation* for more details.

### 3.2.4 Unicode support

As of V1.7, asciimatics is officially misleadingly named! It has support for unicode input and output. Just use a unicode literal where you would previously have used a string. For example:

```
# Should have a telephone at the start...
screen.print_at(u' Call me!', 0, 0, COLOUR_GREEN, A_BOLD)
```

If your system is configured to support unicode, this should be output correctly. However, not all systems will work straight out of the box. See *Unicode characters are not working* for more details on how to fix this.

### 3.2.5 Clearing the Screen

Once you have started your application, you will likely want to clear parts, or all, of the Screen at times. The recommended way to do that is using `clear_buffer()`. This prevents the flicker that you will see if you tried using the previously mentioned *clear* method instead.

## 3.3 Refreshing the Screen

Just using the above methods to output to screen isn't quite enough. The Screen maintains a buffer of what is to be displayed and will only actually display it once the `refresh()` method is called. This is done to reduce flicker on the display device as new content is created.

Applications are required to re-render everything that needs to be displayed and then call refresh when all the new content is ready. Note that the `play()` and `draw_next_frame()` methods will do this for you automatically at the end of each frame, so you don't need to call it again inside your animations.

## 3.4 Input

To handle user input, use the `get_event()` method. This instantly returns the latest key-press or mouse event, without waiting for a new line and without echoing it to screen (for keyboard events). If there is no event available, it will return *None*.

The exact class returned depends on the event. It will be either `KeyboardEvent` or `MouseEvent`. Handling of each is covered below.

If you wish to wait until some input is available, you can use the `wait_for_input()` method to block execution and then call `get_event()` to retrieve the input.

### 3.4.1 KeyboardEvent

This event is triggered for any key-press, including auto repeat when keys are held down. `key_code` is the ordinal representation of the key (taking into account keyboard state - e.g. caps lock) if possible, or an extended key code (the `KEY_xxx` constants in the Screen class) where not.

For example, if you press 'a' normally `get_event()` will return a KeyboardEvent with `key_code` 97, which is `ord('a')`. If you press the same key with caps lock on, you will get 65, which is `ord('A')`. If you press 'F7' you will always get `KEY_F7` irrespective of the caps lock.

The control key (CTRL) on a keyboard returns control codes (the first 31 codes in the ASCII table). You can calculate the control code for any key using the `ctrl()` method. Note that not all systems will return control codes for all keys, so this function can return None if asciimatics doesn't believe the key will work. For best system compatibility, stick to the control codes for alphabetical characters - i.e. "A" to "Z".

As of V1.7, you can also get keyboard events for Unicode characters outside the ASCII character set. These will also return the ordinal representation of the unicode character, just like the previous support for ASCII characters.

If you are seeing random garbage instead, your system is probably not correctly configured for unicode. See *Unicode characters are not working* for how to fix this.

### 3.4.2 MouseEvent

This event is triggered for any mouse movement or button click. The current coordinates of the mouse on the Screen are stored in the `x` and `y` properties. If a button was clicked, this is tracked by the `buttons` property. Allowed values for the buttons are `LEFT_CLICK`, `RIGHT_CLICK` and `DOUBLE_CLICK`.

**Warning:** In general, Windows will report all of these straight out of the box. Linux will only report mouse events if you are using a terminal that supports mouse events (e.g. xterm) in the terminfo database. Even then, not all terminals report all events. For example, the standard xterm function is just to report button clicks. If you need your application to handle mouse move events too, you will need to use a terminal that supports the additional extensions - e.g. the xterm-1003 terminal type. See *Mouse support not working* for more details on how to fix this.

## 3.5 Screen Resizing

It is not possible to change the Screen size through your program. However, the user may resize their terminal or console while your program is running. Asciimatics will continue to run as best as it can within its original dimensions, or you can tell it to re-create the Screen to the new size if desired.

In a little more detail, you can read the Screen size (at the time of creation) from the *dimensions* property. If the user changes the size at any point, you can detect this by calling the *has_resized()* method. In addition, you can tell the Screen to throw an exception if this happens while you are playing a Scene by specifying `stop_on_resize=True`.

Once you have detetected that the screen size has changed using one of the options above, you can either decide to carry on with the current Screen or throw it away and create a new one (by simply creating a new Screen object). If you do the latter, you will typically need to recreate your associated Scenes and Effects to run inside the new boundaries. See the bars.py demo as a sample of how to handle this.

## 3.6 Scraping Text

Sometimes it is useful to be able to read what is already displayed on the Screen at a given location. This is often referred to as screen scraping. You can do this using the *get_from()* method. It will return the displayed character and attributes (as a 4-tuple) for any single character location on the Screen.

```
# Check we've not already displayed something before updating.
current_char, fg, attr, bg = screen.get_from(x, y)
if current_char != 32:
    screen.print_at('X', x, y)
```

**Warning:** Some languages use double-width glyphs. When scraping text for such glyphs, you will find that `get_from` returns the character for both of the 2 locations containing the glyph. For example, if you printed at `(0, 0)`, you would find that asciimatics returns this value for both `(0, 0)` and `(0, 1)`. For more details on which languages (and hence unicode characters) are affected by this see, here and here.

## 3.7 Drawing shapes

The Screen object also provides some anti-aliased line drawing facilities, using ASCII characters to represent the line. The *move()* method will move the drawing cursor to the specified coordinates and then the *draw()* method will draw a straight line from the current cursor location to the specified coordinates.

You can override the anti-aliasing with the `char` parameter. This is most useful when trying to clear what was already drawn. For example:

```
# Draw a diagonal line from the top-left of the screen.
screen.move(0, 0)
screen.draw(10, 10)

# Clear the line
screen.move(0, 0)
screen.draw(10, 10, char=' ')
```

If the resulting line is too thick, you can also pick a thinner pen by specifying `thin=True`. Examples of both styles can be found in the Clock sample code.

In addition, there is the *fill_polygon()* method which will draw a filled polygon in the specified colour using a set of points passed in to define the required shape. This uses the scan-line algorithm, so you can cut holes inside the shape by defining one polygon inside another. For example:

```
# Draw a large with a smaller rectangle hole in the middle.
screen.fill_polygon([[(60, 0), (70, 0), (70, 10), (60, 10)],
                     [(63, 2), (67, 2), (67, 8), (63, 8)]])
```

## 3.8 Unicode drawing

The drawing methods covered above are unicode aware and will default to the correct character set for your terminal, using unicode block characters where possible and falling back to pure ASCII text if not.

CHAPTER 4

# Advanced Output

## 4.1 Rendering

When you want to create an animation, you typically need a sequence of multi-coloured text images to create the desired effect. This is where a `Renderer` object comes into play.

A Renderer is simply an object that will return one or more text strings and associated colour maps in a format that is suitable for display using the `paint()` method. This collation of text string and colour map is referred to as the rendered text. It might vary in complexity from a single, monochrome string through to many frames from an ASCII rendition of a colour video or animated GIF.

All renderers must implement the API of the abstract `Renderer` class, however there are 2 basic variants.

1. The `StaticRenderer` creates pre-rendered sequences of rendered text. They are usually initialized with some static content that can be calculated entirely in advance. For example:

```
# Pre-render ASCIIMATICS using the big Figlet font
renderer = FigletText("ASCIIMATICS", font='big')
```

2. The `DynamicRenderer` creates the rendered text on demand. They are typically dependent on the state of the program or the Screen when rendered. For example:

```
# Render a bar chart with random bars formed of equals signs.
def fn():
    return randint(0, 40)
renderer = BarChart(10, 40, [fn, fn], char='=')
```

Once you have a Renderer you can extract the next text to be displayed by calling `rendered_text()`. This will cycle round the static rendered text sequentially or just create the new dynamic rendered text and return it (for use in the Screen paint method). Generally speaking, rather than doing this directly with the Screen, you will typically want to use an Effect to handle this. See *Animation* for more details.

There are many built-in renderers provided by asciimatics. The following section gives you a quick run through of each one by area. For more examples of Renderers, see the asciimatics samples folder.

### 4.1.1 Image to ASCII

Asciimatics provides 2 ways to convert image files (e.g. JPEGs, GIFs, etc) into a text equivalent:

- `ImageFile` - converts the image to grey-scale text.
- `ColourImageFile` - converts the image to full colour text (using all the screen's palette).

Both support animated GIFs and will cycle through each image when drawn.

### 4.1.2 Animated objects

Asciimatics provides the following renderers for more complex animation effects.

- `BarChart` - draws a horizontal bar chart for a set of data (that may be dynamic in nature).
- `Fire` - simulates a burning fire.
- `Plasma` - simulates an animated "plasma" (think lava lamp in 2-D).
- `Kaleidoscope` - simulates a 2 mirror kaleidoscope.

### 4.1.3 Text/colour manipulation

The following renderers provide some simple text and colour manipulation.

- `FigletText` - draws large FIGlet text
- `Rainbow` - recolours the specified Renderer in as a Rainbow
- `RotatedDuplicate` - creates a rotated duplicate of the specified Renderer.

### 4.1.4 Boxes

The following renderers provide some simple boxes and boxed text.

- `Box` - draws a simple box.
- `SpeechBubble` - draws a speech bubble around some specified text.

## 4.2 Static colour codes

When creating static rendered output, it can be helpful to define your colours inline with the rest of your text. The `StaticRenderer` class supports this through the ${n1,n2,n3} escape sequence, where *n** are digits.

Formally this sequence is defined an escape sequence ${c,a,b} which changes the current colour tuple to be foreground colour 'c', attribute 'a' and background colour 'b' (using the values of the Screen COLOUR and ATTR constants). The attribute and background fields are optional.

These tuples create a colour map (for input into `paint()`) and so the colours will reset to the defaults passed into *paint()* at the start of each line. For example, this code will produce a simple Xmas tree with coloured baubles when rendered (using green as the default colour).

```
StaticRenderer(images=[r"""
       ${3,1}*
      / \
     /${1}o${2}  \
    /_   _\
     /   \${4}b
    /     \
   /   ${1}o${2}   \
  /__      __\
  ${1}d${2} / ${4}o${2}   \
   /       \
  / ${4}o     ${1}o${2}.\
 /___________\
      ${3}|||
      ${3}|||
"""])
```

## 4.3 Experimental

A Renderer can also return a plain text string representation of the next rendered text image. This means they can be used outside of a Screen. For example:

```
# Print a bar chart with random bars formed of equals signs.
def fn():
    return randint(0, 40)
renderer = BarChart(10, 40, [fn, fn], char='=')
print(renderer)
```

# CHAPTER 5

# Animation

## 5.1 Scenes and Effects

The asciimatics package gets its name from a storyboard technique in films (‘animatics’) where simple animations and mock-ups are used to get a better feel for the planned film. Much like these storyboards, you need two key elements for your animation.

1. One or more `Scene` objects that encompass the key stages of your animation.
2. One or more `Effect` objects in each Scene that actually display something on the Screen.

An Effect is basically an object that encodes something to be displayed on the Screen. It can be anything from `Print` that just displays some rendered text at a specific location for a certain time to `Snow` that adds dynamically generated falling snow to the Scene. These are the building blocks of your animation and will be rendered in the strict order that they appear in the Scene, so most of the time you want to put foreground Effects last to ensure they overwrite anything else.

There is no hard and fast rule of how to divide up your Scenes, though there is normally a natural cut where you want to move between effects or clear the Screen, much like you’d need to move to a different cell in a comic strip. These cuts are where you should consider creating a new Scene.

Once you have built up a set of Effects into a list of one or more Scenes, you can pass this list to `play()` which will run through the Scenes in order, or stop playing if the user exits by pressing ‘q’ (assuming you use the default key handling).

## 5.2 Timing Effects

When playing animations, asciimatics will try to redraw the Screen 20 times a second. Each iteration of the loop produces a new frame (no relation to the widget class *Frame*) and increments the frame counter.

This counter is passed as the *frame_no* parameter on `update()` to every *Effect* amd so an be used to time the animation. For example, if you only want the Effect to do something every half a second, you could wait for *frame_no* to increase by 10 before doing the next update.

This is also the counter that determines when to start/stop an *Effect* based on the *start_frame* and *stop_frame* properties on each *Effect*. Specifying non-zero values will delay the start of the *Effect* until, or stop drawing it at, the specified frame count in the *Scene*.

See the credits sample for an example of how to use these properties.

## 5.3 Sprites and Paths

A *Sprite* is a special Effect designed to move some rendered text around the Screen, thus creating an animated character. As such, they work like any other Effect, needing to be placed in a Scene and passed to the Screen (through the `play()` method) to be displayed. They typically take:

- a set of Renderers to animate the motion of the character when moving in any direction
- a default Renderer (to be used when standing still)
- a path to define where the Sprite moves.

Much like Renderers, the paths come in 2 flavours:

1. A *Path* is a pre-defined path that can be fully determined at the start of the program. This provides 4 methods - `jump_to()`, `wait()`, `move_straight_to()` and `move_round_to()` - to define the path. Just decide on the path and script it by chaining these methods together.
2. A *DynamicPath* which depends on the program state and so can only be calculated when needed - e.g. because it depends on what key the user is pressing. These provide an abstract method - `process_event()` - that must be overridden to handle any keys and Update the current coordinates of the Path, to be returned the next time the Sprite asks for an update.

The full declaration of a Sprite is therefore something like this.

```
# Sample Sprite that plots an "X" for each step along an elliptical path.
centre = (screen.width // 2, screen.height // 2)
curve_path = []
for i in range(0, 11):
    curve_path.append(
        (centre[0] + (screen.width / 4 * math.sin(i * math.pi / 5)),
         centre[1] - (screen.height / 4 * math.cos(i * math.pi / 5))))
path = Path()
path.jump_to(centre[0], centre[1] - screen.height // 4),
path.move_round_to(curve_path, 60)
sprite = Sprite(
    screen,
    renderer_dict={
        "default": StaticRenderer(images=["X"])
    },
    path=path,
    colour=Screen.COLOUR_RED,
    clear=False)
```

For more examples of using Sprites, including dynamic Paths, see the samples directory.

## 5.4 Particle Systems

A *ParticleEffect* is a special Effect designed to draw a particle system. It consists of one or more *ParticleEmitter* objects which in turn contains one or more *Particle* objects.

The ParticleEffect defines a chain of ParticleEmitters that spawn one or more Particles, each with a unique set of attributes - e.g. location, direction, colour, etc. The ParticleEffect renders a frame by rendering each of these Particles and then updating them following the rules defined by the ParticleEmitter.

It all sounds a bit convoluted, doesn't it? Let's try a concrete example to clarify it... Consider the *StarFirework* effect. This is constructed as follows.

1. The StarFirework constructs a Rocket. This is a ParticleEmitter that has just one Particle that shoots vertically up the Screen to hit a pre-defined end point.
2. When this Particle hits its end-point, it expires and spawns a StarExplosion. This is a ParticleEmitter that spawns many Particles in such a way that they are explode outwards radially from where the Rocket expired.
3. In turn, each of these Particles from the StarExplosion spawns a StarTrail on each new frame. These are ParticleSystems that spawn a single Particle which just hovers for a few frames and fades away.

Putting this all together (by playing the Effect) you have a classic exploding firework. For more examples, see the other Effects in the particles and fireworks samples.

## 5.5 CPU Considerations

Many people run asciimatics on low-power systems and so care about CPU. However there is a trade-off between CPU usage and responsiveness of any User Interface or the slickness of any animation. Asciimatics tries to handle this for you by looking at when each Effect next wants to be redrawn and only refreshing the Screen when needed.

For most use-cases, this default should be enough for your needs. However, there are a couple of cases where you might need more. The first is very low-power (e.g. SOC) systems where you need to keep CPU usage to a minimum for a widget-based UI. In this case, you can use the reduce_cpu parameter when constructing your *Frame*.

The other case, is actually the opposite problem - you may find that asciimatics is being too conservative and you need to refresh the Screen before it thinks you need to do so. In this case, you can simply force its hand by calling *force_update()*, which will force a full refresh of the Screen next time that *draw_next_frame()* is called.

## 5.6 Using async frameworks

If you cannot allow asciimatics to schedule each frame itself, e.g. because you are using an asynchronous framework like gevent, asyncio or twisted, that's fine. Asciimatics is designed to run in tiny time slices that are ideal for such a framework. All you need to do is call *set_scenes()* to set up your scenes and *draw_next_frame()* (every 1/20 of a second) to draw the next frame.

For example, here is how you can run inside an asyncio event loop.

```python
import asyncio
from asciimatics.effects import Cycle, Stars
from asciimatics.renderers import FigletText
from asciimatics.scene import Scene
from asciimatics.screen import Screen


def update_screen(end_time, loop, screen):
    screen.draw_next_frame()
    if loop.time() < end_time:
        loop.call_later(0.05, update_screen, end_time, loop, screen)
```

(continues on next page)

(continued from previous page)

```
    else:
        loop.stop()


# Define the scene that you'd like to play.
screen = Screen.open()
effects = [
    Cycle(
        screen,
        FigletText("ASCIIMATICS", font='big'),
        screen.height // 2 - 8),
    Cycle(
        screen,
        FigletText("ROCKS!", font='big'),
        screen.height // 2 + 3),
    Stars(screen, (screen.width + screen.height) // 2)
]
screen.set_scenes([Scene(effects, 500)])

# Schedule the first call to display_date()
loop = asyncio.new_event_loop()
end_time = loop.time() + 5.0
loop.call_soon(update_screen, end_time, loop, screen)

# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
screen.close()
```

# CHAPTER 6

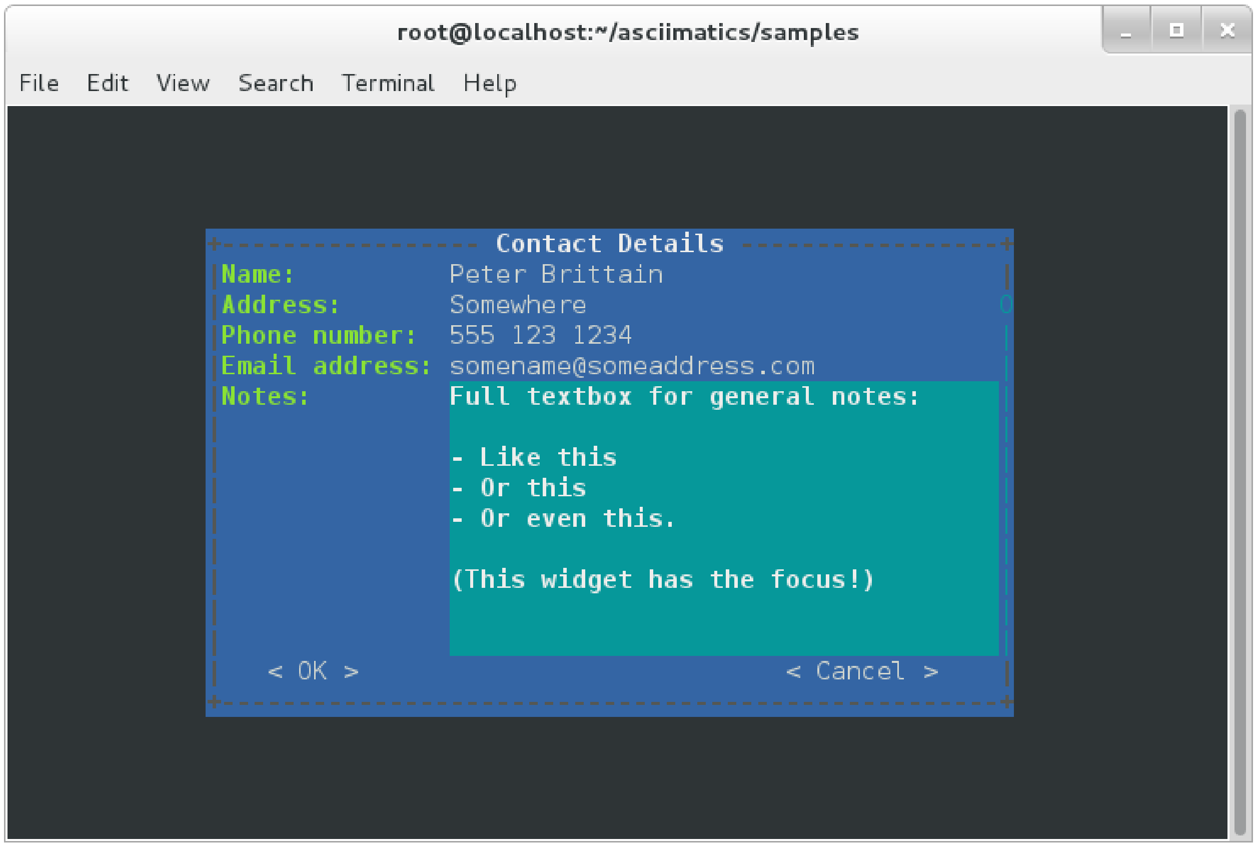
# User Interfaces

## 6.1 Introduction

Asciimatics provides a *widgets* sub-package that allows you to create interactive text user interfaces. At its heart, the logic is quite simple, reusing concepts from standard web and desktop GUI frameworks.

1. The basic building block for your text UI is a *Widget*. There is a set of standard ones provided by asciimatics, but you can create a custom set if needed. The basic set has strong parallels with simple web input forms - e.g. buttons, check boxes, etc.
2. The *Widgets* need to be arranged on the *Screen* and rearranged whenever it is resized. The *Layout* class handles this for you. You just need to add your *Widgets* to one.
3. You then need to display the *Layouts*. To do this, you must add them to a *Frame*. This class is an *Effect* and so can be used in any *Scene* alongside any other *Effect*. The *Frame* will draw any parts of the *Layouts* it contains that are visible within its boundaries. The net result is that it begins to look a bit like a window in GUI frameworks.

And that's it! You can set various callbacks to get triggered when key events occur - e.g. changes to values, buttons get clicked, etc. - and use these to trigger your application processing. For an example, see the contact_list.py sample provided - which will look a bit like this:


root@localhost:~/asciimatics/samples
File Edit View Search Terminal Help

```
+---------------- Contact Details ----------------+
|Name:          Peter Brittain                   |
|Address:       Somewhere                        |
|Phone number:  555 123 1234                     |
|Email address: somename@someaddress.com         |
|Notes:         Full textbox for general notes:  |
|                                                |
|               - Like this                      |
|               - Or this                        |
|               - Or even this.                  |
|                                                |
|               (This widget has the focus!)     |
|                                                |
|                                                |
|   < OK >                         < Cancel >    |
+------------------------------------------------+
```

## 6.1.1 Common keys

When navigating around a Frame, you can normally use the following keys.

| Key | Action |
|---|---|
| Tab | Move to the next Widget in the Frame |
| Backtab (shift+tab) | Move to the previous Widget in the Frame |
| Up arrow | Move to the nearest Widget above the current focus. |
| Down arrow | Move to the nearest Widget below the current focus. |
| Left arrow | Move to the nearest Widget to the left of the current focus. |
| Right arrow | Move to the nearest Widget to the right of the current focus. |
| Space or Return | Select the current Widget - e.g. click a Button, or pop-up a list of options. |

**Warning:** Please note that asciimatics will not allow you to navigate to a disabled widget. Instead it will select the next enabled widget when traversing the Frame.

However, some widgets (e.g. text editing widgets) have their own logic for handling the cursor key actions, which override the common navigation. In such cases, tab/backtab will still navigate out of the Widgets.

In addition you can also use the following extra keys inside text editing widgets.

| Key | Action |
|---|---|
| Home/End | Move to the start/end of the current line. |
| Delete | Delete the character under the cursor. |
| Backspace | Delete the character before the cursor. |

## 6.2 Model/View Design

Before we jump into exactly what all the objects are and what they do for you, it is important to understand how you must put them together to make the best use of them.

The underlying Screen/Scene/Effect design of asciimatics means that objects regularly get thrown away and recreated - especially when the Screen is resized. It is therefore vital to separate your data model from your code to display it on the screen.

This split is often (wrongly) termed the MVC model, but a more accurate description is Separated Presentation. No matter what term you use, the concept is easy: use a separate class to handle your persistent data storage.

In more concrete terms, let's have a closer look at the contact_list sample. This consists of 3 basic classes:

1. *ContactModel*: This is the model. It stores simple contact details in a sqlite in-memory database and provides a simple create/read/update/delete interface to manipulate any contact. Note that you don't have to be this heavy-weight with the data storage; a simple class to wrap a list of dictionaries would also suffice - but doesn't look as professional for a demo!

**Show/hide code**

```
class ContactModel(object):
    def __init__(self):
        # Create a database in RAM
        self._db = sqlite3.connect(':memory:')
        self._db.row_factory = sqlite3.Row

        # Create the basic contact table.
        self._db.cursor().execute('''
            CREATE TABLE contacts(
                id INTEGER PRIMARY KEY,
                name TEXT,
                phone TEXT,
                address TEXT,
                email TEXT,
                notes TEXT)
        ''')
        self._db.commit()

        # Current contact when editing.
        self.current_id = None

    def add(self, contact):
        self._db.cursor().execute('''
            INSERT INTO contacts(name, phone, address, email, notes)
            VALUES(:name, :phone, :address, :email, :notes)''',
                                  contact)
        self._db.commit()

    def get_summary(self):
```

(continues on next page)

(continued from previous page)

```
        return self._db.cursor().execute(
            "SELECT name, id from contacts").fetchall()

    def get_contact(self, contact_id):
        return self._db.cursor().execute(
            "SELECT * from contacts where id=?", str(contact_id)).fetchone()

    def get_current_contact(self):
        if self.current_id is None:
            return {"name": "", "address": "", "phone": "", "email": "", "notes": ""}
        else:
            return self.get_contact(self.current_id)

    def update_current_contact(self, details):
        if self.current_id is None:
            self.add(details)
        else:
            self._db.cursor().execute('''
                UPDATE contacts SET name=:name, phone=:phone, address=:address,
                email=:email, notes=:notes WHERE id=:id''',
                                      details)
            self._db.commit()

    def delete_contact(self, contact_id):
        self._db.cursor().execute('''
            DELETE FROM contacts WHERE id=:id''', {"id": contact_id})
        self._db.commit()
```

2. *ListView*: This is the main view. It queries the *ContactModel* for the list of known contacts and displays them in a list, complete with some extra buttons to add/edit/delete contacts.

**Show/hide code**

```
class ListView(Frame):
    def __init__(self, screen, model):
        super(ListView, self).__init__(screen,
                                       screen.height * 2 // 3,
                                       screen.width * 2 // 3,
                                       on_load=self._reload_list,
                                       hover_focus=True,
                                       title="Contact List")
        # Save off the model that accesses the contacts database.
        self._model = model

        # Create the form for displaying the list of contacts.
        self._list_view = ListBox(
            Widget.FILL_FRAME,
            model.get_summary(), name="contacts", on_select=self._on_pick)
        self._edit_button = Button("Edit", self._edit)
        self._delete_button = Button("Delete", self._delete)
        layout = Layout([100], fill_frame=True)
        self.add_layout(layout)
        layout.add_widget(self._list_view)
        layout.add_widget(Divider())
        layout2 = Layout([1, 1, 1, 1])
        self.add_layout(layout2)
        layout2.add_widget(Button("Add", self._add), 0)
```

(continues on next page)

(continued from previous page)

```
        layout2.add_widget(self._edit_button, 1)
        layout2.add_widget(self._delete_button, 2)
        layout2.add_widget(Button("Quit", self._quit), 3)
        self.fix()

    def _on_pick(self):
        self._edit_button.disabled = self._list_view.value is None
        self._delete_button.disabled = self._list_view.value is None

    def _reload_list(self):
        self._list_view.options = self._model.get_summary()
        self._model.current_id = None

    def _add(self):
        self._model.current_id = None
        raise NextScene("Edit Contact")

    def _edit(self):
        self.save()
        self._model.current_id = self.data["contacts"]
        raise NextScene("Edit Contact")

    def _delete(self):
        self.save()
        self._model.delete_contact(self.data["contacts"])
        self._reload_list()

    @staticmethod
    def _quit():
        raise StopApplication("User pressed quit")
```

3. *ContactView*: This is the detailed view. It queries the *ContactModel* for the current contact to be displayed when it is reset (note: there may be no contact if the user is adding a contact) and writes any changes back to the model when the user clicks OK.

**Show/hide code**

```
class ContactView(Frame):
    def __init__(self, screen, model):
        super(ContactView, self).__init__(screen,
                                          screen.height * 2 // 3,
                                          screen.width * 2 // 3,
                                          hover_focus=True,
                                          title="Contact Details")
        # Save off the model that accesses the contacts database.
        self._model = model

        # Create the form for displaying the list of contacts.
        layout = Layout([100], fill_frame=True)
        self.add_layout(layout)
        layout.add_widget(Text("Name:", "name"))
        layout.add_widget(Text("Address:", "address"))
        layout.add_widget(Text("Phone number:", "phone"))
        layout.add_widget(Text("Email address:", "email"))
        layout.add_widget(TextBox(5, "Notes:", "notes", as_string=True))
        layout2 = Layout([1, 1, 1, 1])
        self.add_layout(layout2)
```

(continues on next page)

(continued from previous page)

```
        layout2.add_widget(Button("OK", self._ok), 0)
        layout2.add_widget(Button("Cancel", self._cancel), 3)
        self.fix()

    def reset(self):
        # Do standard reset to clear out form, then populate with new data.
        super(ContactView, self).reset()
        self.data = self._model.get_current_contact()

    def _ok(self):
        self.save()
        self._model.update_current_contact(self.data)
        raise NextScene("Main")

    @staticmethod
    def _cancel():
        raise NextScene("Main")
```

## 6.3 Displaying your UI

OK, so you want to do something a little more interactive with your user. The first thing you need to decide is what information you want to get from them and how you're going to achieve that. In short:

1. What data you want them to be able to enter - e.g. their name.
2. How you want to break that down into fields - e.g. first name, last name.
3. What the natural representation of those fields would be - e.g. text strings.

At this point, you can now decide which Widgets you want to use. The standard selection is as follows.

| Widget type | Description |
|---|---|
| `Button` | Action buttons - e.g. ok/cancel/etc. |
| `CheckBox` | Simple yes/no tick boxes. |
| `DatePicker` | A single-line widget for selecting a date (using a pop-up list). |
| `Divider` | A spacer between widgets (for aesthetics). |
| `DropdownList` | A single-line widget that pops up a list from which the user can select a single value. |
| `FileBrowser` | A multi-line widget for listing the local file system. |
| `Label` | A label for a group of related widgets. |
| `ListBox` | A list of possible options from which users can select one value. |
| `MultiColumnListBox` | Like a ListBox, but for displaying tabular data. |
| `RadioButtons` | A list of radio buttons. These allow users to select one value from a list of options. |
| `Text` | A single line of editable text. |
| `TextBox` | A multi-line box of editable text. |
| `TimePicker` | A single-line widget for selecting a time (using a pop-up list). |
| `VerticalDivider` | A vertical line divider - useful for providing a visual marker between columns in a Layout. |

**Note:** You can use the *hide_char* option on Text widgets to hide sensitive data - e.g. for passwords.

Asciimatics will automatically arrange these for you with just a little extra help. All you need to do is decide how many columns you want for your fields and which fields should be in which columns. To tell asciimatics what to do you create a *Layout* (or more than one if you want a more complex structure where different parts of the screen need differing column counts) and associate it with the *Frame* where you will display it.

For example, this will create a Frame that is 80x20 characters and define 4 columns that are each 20 columns wide:

```
frame = Frame(screen, 80, 20, has_border=False)
layout = Layout([1, 1, 1, 1])
frame.add_layout(layout)
```

Once you have a Layout, you can add Widgets to the relevant column. For example, this will add a button to the first and last columns:

```
layout.add_widget(Button("OK", self._ok), 0)
layout.add_widget(Button("Cancel", self._cancel), 3)
```

If you want to put a standard label on all your input fields, that's fine too; asciimatics will decide how big your label needs to be across all fields in the same column and then indent them all to create a more aesthetically pleasing layout. For example, this will provide a single column with labels for each field, indenting all of the fields to the same depth:

```
layout = Layout([100])
frame.add_layout(layout)
layout.add_widget(Text("Name:", "name"))
layout.add_widget(Text("Address:", "address"))
layout.add_widget(Text("Phone number:", "phone"))
layout.add_widget(Text("Email address:", "email"))
layout.add_widget(TextBox(5, "Notes:", "notes", as_string=True))
```

If you want more direct control of your labels, you could use the `Label` widget to place them anywhere in the Layout as well as control the justification (left, centre or right) of the text.

Or maybe you just want some static text in your UI? The simplest thing to do there is to use the `Label` widget. If you need something a little more advanced - e.g. a pre-formatted multi-line status bar, use a `TextBox` and disable it as described below.

In some cases, you may want to have different alignments for various blocks of Widgets. You can use multiple Layouts in one Frame to handle this case.

For example, if you want a search page, which allows you to enter data at the top and a list of results at the bottom of the Frame, you could use code like this:

```
layout1 = Layout([100])
frame.add_layout(layout1)
layout1.add_widget(Text(label="Search:", name="search_string"))

layout2 = Layout([100])
frame.add_layout(layout2)
layout1.add_widget(TextBox(Widget.FILL_FRAME, name="results"))
```

### 6.3.1 Disabling widgets

Any widget can be disabled by setting the `disabled` property. When this is `True`, asciimatics will redraw the widget using the 'disabled' colour palette entry and prevent the user from selecting it or editing it.

It is still possible to change the widget programmatically, though. For example, you can still change the `value` of a disabled widget.

This is the recommended way of getting a piece of non-interactive data (e.g. a status bar) into your UI. If the disabled colour is the incorrect choice for your UI, you can override it as explained in *Custom widget colours*. For an example of such a widget, see the top.py sample.

## 6.3.2 Layouts in more detail

If you need to do something more complex, you can use multiple Layouts. Asciimatics uses the following logic to determine the location of Widgets.

1. The *Frame* owns one or more *Layouts*. The *Layouts* stack one above each other when displayed - i.e. the first *Layout* in the *Frame* is above the second, etc.
2. Each *Layout* defines some horizontal constraints by defining columns as a proportion of the full *Frame* width.
3. The *Widgets* are assigned a column within the *Layout* that owns them.
4. The *Layout* then decides the exact size and location to make each *Widget* best fit the visible space as constrained by the above.

For example:

```
+-------------------------------------------------------------------------+
|Screen...................................................................|
|.........................................................................|
|...+-----------------------------------------------------------------+...|
|...|Frame                                                            |...|
|...|+---------------------------------------------------------------+|...|
|...||Layout 1                                                       ||...|
|...|+---------------------------------------------------------------+|...|
|...|+-------------------------------+-------------------------------+|...|
|...||Layout 2                       |                               ||...|
|...|| - Column 1                    | - Column 2                    ||...|
|...|+-------------------------------+-------------------------------+|...|
|...|+-------------+----------------------------------+--------------+|...|
|...||Layout 3     | < Widget 1 >                     |              ||...|
|...||             | ...                              |              ||...|
|...||             | < Widget N >                     |              ||...|
|...|+-------------+----------------------------------+--------------+|...|
|...+-----------------------------------------------------------------+...|
|.........................................................................|
+-------------------------------------------------------------------------+
```

This consists of a single *Frame* with 3 *Layouts*. The first is a single, full-width column, the second has two 50% width columns and the third consists of 3 columns of relative size 25:50:25. The last actually contains some Widgets in the second column (though this is just for illustration purposes as we'd expect most Layouts to have some Widgets in them).

## 6.3.3 Filling the space

Once you've got the basic rows and columns for your UI sorted, you may want to use some strategic spacing. At the simplest level, you can use the previously mentioned `Divider` widget to create some extra vertical space or insert a visual section break.

Moving up the complexity, you can pick different sizes for your Frames based on the size of your current Screen. The Frame will be recreated when the screen is resized and so you will use more or less real estate appropriately.

Finally, you could also tell asciimatics to use an object to fill any remaining space. This allows for the sort of UI like you'd see in applications like top where you have a fixed header or footer, but then a variably sized part that contains the data to be displayed.

You can achieve this in 2 ways:

1. You can tell a Layout to fill any remaining space in the Frame using *fill_frame=True* on construction.
2. You can tell some Widgets to fill any remaining space in the Frame using a height of *Widget.FILL_FRAME* on construction.

These two methods can be combined to tell a Layout to fill the Frame and a Widget to fill this Layout. See the ListView class in the contact_list demo code.

**Warning:** Note that you can only have one Layout and/or Widget that fills the Frame. Trying to set more than one will be rejected.

### 6.3.4 Full-screen Frames

By default, asciimatics assumes that you are putting multiple Frames into one Scene and so provides defaults (e.g. borders) to optimize this type of UI. However, some UIs only need a single full-screen Frame. This can easily be achieved by declaring a Frame the full width and height of the screen and then specifying *has_border=False*.

### 6.3.5 Large forms

If you have a very large form, you may find it is too big to fit into a standard screen. This is not a problem. You can keep adding your Widgets to your Layout and asciimatics will automatically clip the content to the space available and scroll the content as required.

If you do this, it is recommended that you set *has_border=True* on the Frame so that the user can use the scroll bar provided to move around the form.

### 6.3.6 Colour schemes

The colours for any Widget are determined by the *palette* property of the Frame that contains the Widget. If desired, it is possible to have a different palette for every Frame, however your users may prefer a more consistent approach.

The palette is just a simple dictionary to map Widget components to a colour tuple. A colour tuple is simply the foreground colour, attribute and background colour. For example:

```
(Screen.COLOUR_GREEN, Screen.A_BOLD, Screen.COLOUR_BLUE)
```

The following table shows the required keys for the *palette*.

| Key | Usage |
|---|---|
| "background" | Frame background |
| "borders" | Frame border and Divider Widget |
| "button" | Buttons |
| "control" | Checkboxes and RadioButtons |
| "disabled" | Any disabled Widget |
| "edit_text" | Text and TextBox |
| "field" | Value of an option for a Checkbox, RadioButton or Listbox |
| "focus_button" | Buttons with input focus |
| "focus_control" | Checkboxes and RadioButtons with input focus |
| "focus_edit_text" | Text and TextBox with input focus |
| "focus_field" | As above with input focus |
| "invalid" | The widget contains invalid data |
| "label" | Widget labels |
| "scroll" | Frame scroll bar |
| "selected_control" | Checkboxes and RadioButtons when selected |
| "selected_field" | As above when selected |
| "selected_focus_control" | Checkboxes and RadioButtons with both |
| "selected_focus_field" | As above with both |
| "title" | Frame title |

In addition to the default colour scheme for all your widgets, asciimatics provides some other pre-defined colour schemes (or themes) that you can use for your widgets using `set_theme()`. These themes are as follows.

| Name | Description |
|---|---|
| "monochrome" | Simple black and white colour scheme. |
| "green" | A classic green terminal. |
| "bright" | Black background, green and yellow scheme. |
| "tlj256" | Shades of black white and red - 256 colour terminals only. |

You can add your own theme to this list by defining a new entry in the `THEMES`

## 6.3.7 Custom widget colours

In some cases, a single palette for the entire Frame is not sufficient. If you need a more fine-grained approach to the colouring, you can customize the colour for any Widget by setting the `custom_colour` for that Widget. The only constraint on this property is that it must still be the value of one of the keys within the owning Frame's palette.

## 6.3.8 Changing colours inline

The previous options should be enough for most UIs. However, sometimes it is useful to be able to change the colour of some text inside the value for some widgets, e.g. to provide syntax highlighting in a *TextBox*. You can do this using a `Parser` object for those widgets that support it.

By passing in a parser that understands extra control codes or the need to highlight certain characters differently, you can control colours on a letter by letter basis. Out of the box, asciimatics provides 2 parsers, which can handle the ${c,a,b} format used by its Renderers, or the ANSI standard terminal escape codes (used by many Linux terminals). Simply use the relevant parser (`AsciimaticsParser` or `AnsiTerminalParser`) and pass in values containing the associated control codes to change colours where needed.

Check out the latest code in forms.py and top.py for examples of how this works.

## 6.4 Setting values

By this stage, you should have a basic User Interface up and running, but how do you set the values in each of the Widgets - e.g. to pre-populate known values in a form? There are 2 ways to handle this:

1. You can set the value directly on each *Widget* using the `value` property.
2. You can set the value for all Widgets in a *Frame* by setting at the `data` property. This is a simple key/value dictionary, using the *name* property for each *Widget* as the keys.

The latter is a preferred as a symmetrical solution is provided to access all the data for each Widget, thus giving you a simple way to read and then replay the data back into your Frame.

## 6.5 Getting values

Now that you have a *Frame* with some *Widgets* in it and the user is filling them in, how do you find out what they entered? There are 2 basic ways to do this:

1. You can query each Widget directly, using the *value* property. This returns the current value the user has entered at any time (even when the Frame is not active). Note that it may be *None* for those *Widgets* where there is no value - e.g. buttons.
2. You can query the *Frame'by looking at the 'data* property. This will return the value for every Widget in the former as a dictionary, using the Widget *name* properties for the keys. Note that *data* is just a cache, which only gets updated when you call `save()`, so you need to call this method to refresh the cache before accessing it.

For example:

```
# Form definition
layout = Layout([100])
frame.add_layout(layout)
layout.add_widget(Text("Name:", "name"))
layout.add_widget(Text("Address:", "address"))
layout.add_widget(TextBox(5, "Notes:", "notes", as_string=True))

# Sample frame.data after user has filled it in.
{
    "name": "Peter",
    "address": "Somewhere on earth",
    "notes": "Some multi-line\ntext from the user."
}
```

### 6.5.1 Validating text data

Free-form text input sometimes needs validating to make sure that the user has entered the right thing - e.g. a valid email address - in a form. Asciimatics makes this easy by adding the *validator* parameter to *Text* widgets.

This parameter takes either a regular expression string or a function (taking a single parameter of the current widget value). Asciimatics will use it to determine if the widget contains valid data. It uses this information in 2 places.

1. Whenever the *Frame* is redrawn, asciimatics will check the state and flag any invalid values using the *invalid* colour palette selection.
2. When your program calls `save()` specifying *validate=True*, asciimatics will check all fields and throw an `InvalidFields` exception if it finds any invalid data.

### 6.5.2 Input focus

As mentioned in the explanation of colour palettes, asciimatics has the concept of an input focus. This is the Widget that will take any input from the keyboard. Assuming you are using the default palette, the Widget with the input focus will be highlighted. You can move the focus using the cursor keys, tab/backtab or by using the mouse.

The exact way that the mouse affects the focus depends on a combination of the capabilities of your terminal/console and the settings of your Frame. At a minimum, clicking on the Widget will always work. If you specify *hover_focus=True* and your terminal supports reporting mouse move events, just hovering over the Widget with the mouse pointer will move the focus.

### 6.5.3 Modal Frames

When constructing a Frame, you can specify whether it is modal or not using the *is_modal* parameter. Modal Frames will not allow any input to filter through to other Effects in the Scene, so when one is on top of all other Effects, this means that only it will see the user input.

This is commonly used for, but not limited to, notifications to the user that must be acknowledged (as implemented by `PopUpDialog`).

### 6.5.4 Global key handling

In addition to mouse control to switch focus, you can also set up a global event handler to navigate your forms. This is useful for keyboard shortcuts - e.g. Ctrl+Q to quit your program.

To set up this handler, you need to pass it into your screen on the *play()* Method. For example

```python
# Event handler for global keys
def global_shortcuts(event):
    if isinstance(event, KeyboardEvent):
        c = event.key_code
        # Stop on ctrl+q or ctrl+x
        if c in (17, 24):
            raise StopApplication("User terminated app")

# Pass this to the screen...
screen.play(scenes, unhandled_input=global_shortcuts)
```

**Warning:** Note that the global handler is only called if the focus does not process the event. Some widgets - e.g. TextBox - take any printable text and so the only keys that always get to this handler are the control codes. Others will sometimes get here depending on the type of Widget in focus and whether the Frame is modal or not..

By default, the global handler will do nothing if you are playing any Scenes containing a Frame. Otherwise it contains the top-level logic for skipping to the next Scene (on space or enter), or exiting the program (on Q or X).

### 6.5.5 Dealing with Ctrl+C and Ctrl+Z

A lot of modern UIs want to be able to use Ctrl+C/Z to do something other than kill the application. The problem for Python is that this normally triggers a *KeyboardInterrupt* - which typically kills the application - or causes the operating system to suspend the process (on UNIX variants).

If you want to prevent this and use Ctrl+C/Z for another purpose, you can tell asciimatics to catch the low-level signals to prevent these interrupts from being generated (and so return the keypress to your application). This is done by specifying *catch_interrupt=True* when you create the *Screen* by calling `wrapper()`.

### 6.5.6 Dealing with Ctrl+S

Back in the days when terminals really were separate machines connected over wires to a computer, it was necessary to be able to signal that the terminal needed time to catch up. This was done using software flow control, using the Ctrl+S/Ctrl+Q control codes to tell the computer to stop/restart sending text.

These days, it's not really necessary, but is still a supported feature on most terminals. On some systems you can switch this off so you get access to Ctrl+S, but it is not possible on them all. See *Ctrl+S does not work* for details on how to fix this.

## 6.6 Flow of control

By this stage you should have a program with some Frames and can extract what your user has entered into any of them. But how do you know when to act and move between Frames? The answer is callbacks and exceptions.

### 6.6.1 Callbacks

A callback is just a function that you pass into another function to be called when the associated event occurs. In asciimatics, they can usually be identified by the fact that they start with *on* and correspond to a significant input action from the user, e.g. *on_click*.

When writing your application, you simply need to decide which events you want to use to trigger some processing and create appropriate callbacks. The most common pattern is to use a *Button* and define an *on_click* callback.

In addition, there are other events that can be triggered when widget values change. These can be used to provide dynamic effects like enabling/disabling Buttons based on the current value of another Widget.

### 6.6.2 Exceptions

**Asciimatics uses exceptions to tell the animation engine to move to a new Scene or stop the whole** process.
Other exceptions are not caught and so can still be used as normal. The details for the new exceptions are as follows:

1. `StopApplication` - This exception will stop the animation engine and return flow to the function that called into the Screen.
2. `NextScene` - This exception tells the animation engine to move to a new Scene. The precise Scene is determined by the name passed into the exception. If none is specified, the engine will simply roundi robin to the next available Scene.

Note that the above logic requires each Scene to be given a unique name on construction. For example:

```
# Given this scene list...
scenes = [
    Scene([ListView(screen, contacts)], -1, name="Main"),
    Scene([ContactView(screen, contacts)], -1, name="Edit Contact")
]
screen.play(scenes)
```

(continues on next page)

(continued from previous page)

```
# You can use this code to move back to the first scene at any time...
raise NextScene("Main")
```

## 6.7 Data handling

By this stage you should have everything you need for a fully functional UI. However, it may not be quite clear how to pass data around all your component parts because asciimatics doesn't provide any classes to do it for you. Why? Because we don't want to tie you down to a specific implementation. You should be able to pick your own!

Look back at the earlier explanation of model/view design. The model can be any class you like! All you need to do is:

1. Define a model class to store any state and provide suitable APIs to access it as needed from your UI (a.k.a. views).
2. Define your own views (based on an `Effect` or `Frame`) to define your UI and store a reference to the model (typically as a parameter on construction).
3. Use that saved reference to the model to handle updates as needed inside your view's callbacks or methods.

For a concrete example of how to do this check out the contact list sample and look at how it defines and uses the `ContactModel`. Alternatively, the quick_model sample shows how the same forms would work witha simple list of dictionaries instead.

## 6.8 Dynamic scenes

That done, there are just a few more final touches to consider. These all touch on dynamically changing or reconstructing your Scene.

At a high level, you need to decide what you want to achieve. The basic options are as follows.

1. If you just want to have some extra Frames on the same Screen - e.g. pop-up windows - that's fine. Just use the existing classes (see below)!
2. If you want to be able to draw other content outside of your existing Frame(s), you probably want to use other Effects.
3. If you want to be able to add something inside your Frame(s), you almost certainly want to create a custom Widget for that new content.

The rest of this section goes through those options (and a couple more related changes) in a little more detail.

### 6.8.1 Adding other effects

Since Frames are just another Effect, they can be combined with any other Effect in a Scene. For example, this will put a simple input form over the top of the animated Julia set Effect:

```
scenes = []
effects = [
    Julia(screen),
    InputFormFrame(screen)
]
scenes.append(Scene(effects, -1))
screen.play(scenes)
```

The ordering is important. The effects at the bottom of the list are at the top of the screen Z order and so will be displayed in preference to those lower in the Z order (i.e. those earlier in the list).

The most likely reason you will want to use this is to use the *Background* Effect to set a background colour for the whole screen behind your Frames. See the forms.py demo for an example of this use case.

### 6.8.2 Pop-up dialogs

Along a similar line, you can also add a *PopUpDialog* to your Scenes at any time. These consist of a single text message and a set of buttons that you can define when creating the dialog.

Owing to restrictions on how objects need to be rebuilt when the screen is resized, these should be limited to simple are confirmation or error cases - e.g. “Are you sure you want to quit?” For more details on the restrictions, see the section on restoring state.

### 6.8.3 Pop-up menus

You can also add a *PopupMenu* to your Scenes in the same way. These allow you to create a simple temporary list of options from which the user has to select just one entry (by clicking on it or moving the focus and pressing Enter) or dismiss the whole list (by pressing Escape or clicking outside of the menu).

Owing to their temporary nature, they are not maintained over screen resizing.

### 6.8.4 Screen resizing

If you follow the standard application mainline logic as found in all the sample code, your application will want to resize all your Effects and Widgets whenever the user resizes the terminal. To do this you need to get a new Screen then rebuild a new set of objects to use that Screen.

Sound like a bit of a drag, huh? This is why it is recommended that you separate your presentation from the rest of your application logic. If you do it right you will find that it actually just means you go through exactly the same initialization path as you did before to create your Scenes in the first place. There are a couple of gotchas, though.

First, you need to make sure that asciimatics will exit and recreate a new Screen when the terminal is resized. You do that with this boilerplate code that is in most of the samples.

```python
def main(screen, scene):
    # Define your Scenes here
    scenes = ...

    # Run your program
    screen.play(scenes, stop_on_resize=True, start_scene=scene)

last_scene = None
while True:
    try:
        Screen.wrapper(main, arguments=[last_scene])
        sys.exit(0)
    except ResizeScreenError as e:
        last_scene = e.scene
```

**This will allow you to decide how all your UI should look whenever the screen is resized and will** restart at the
Scene that was playing at the time of the resizing.

### 6.8.5 Restoring state

Recreating your view is only half the story. Now you need to ensure that you have restored any state inside your application - e.g. any dynamic effects are added back in, your new Scene has the same internal state as the old, etc. Asciimatics provides a standard interface (the *clone* method) to help you out here.

When the running *Scene* is resized (and passed back into the Screen as the start scene), the new *Scene* will run through all the *Effects* in the old copy looking for any with a *clone* method. If it finds one, it will call it with 2 parameters: the new *Screen* and the new *Scene* to own the cloned *Effect*. This allows you to take full control of how the new *Effect* is recreated. Asciimatics uses this interface in 2 ways by default:

1. To ensure that any `data` is restored in the new *Scene*.
2. To duplicate any dynamically added `PopUpDialog` objects in the new *Scene*.

You could override this processing to handle your own custom cloning logic. The formal definition of the API is defined as follows.

```
def clone(self, screen, scene):
    """
    Create a clone of this Effect into a new Screen.

    :param screen: The new Screen object to clone into.
    :param scene: The new Scene object to clone into.
    """
```

### 6.8.6 Reducing CPU usage

It is the nature of text UIs that they don't need to refresh anywhere near as often as a full-blown animated Scene. Asciimatics therefore optimizes the refresh rate when only Frames are being displayed on the Screen.

However, there are some widgets that can reduce the need for animation even further by not requesting animation updates (e.g. for a blinking cursor). If this is an issue for your application, you can specify `reduce_cpu=True` when constructing your Frames. See contact_list.py for an example of this.

## 6.9 Custom widgets

To develop your own widget, you need to define a new class that inherits from `Widget`. You then have to implement the following functions.

1. `reset()` - This is where you should reset any state for your widget. It gets called whenever the owning Frame is initialised, which can be when it is first displayed, when the user moves to a new Scene or when the screen is resized.
2. `update()` - This is where you should put the logic to draw your widget. It gets called every time asciimatics needs to redraw the screen (and so should always draw the entire widget).
3. `process_event()` - This is where you should put your code to handle mouse and keyboard events.
4. `value` - This must return the current value for the widget.
5. `required_height()` - This returns the minimum required height for your widget. It is used by the owning Layout to determine the size and location of your widget.

With these all defined, you should now be able to add your new custom widget to a Layout like any of the standard ones delivered in this package.

# CHAPTER 7

# Troubleshooting

## 7.1 Installation issues

### 7.1.1 Android

To run on Android, you need access to a CLI environment. I've found that https://termux.com does the trick, but you need to install some extra packages before you can install asciimatics.

After installing termux, start up the app and run the following commands:

```
apt update
apt-get install clang python-dev libjpeg-turbo-dev
LDFLAGS=-L/system/lib pip install Pillow
pip install asciimatics
```

### 7.1.2 Linux

Although asciimatics is a pure python implementation, it depends on Pillow (a fork of the Python Imaging Library). This package depends on some native libraries that must be installed first. For details of what libararies you need, see the Pillow documentation.

For a list of possible solutions, see the answer on Stackoverflow. In short, either install the native libraries you need, or force an installation of an older version (2.9.0) of Pillow.

## 7.2 My application only runs on Windows

Given that your application runs on Windows, but not any curses-based solution (i.e. Mac or Linux), the fundamental logic in your code is right. It might be a bug in asciimatics, but it could also be a bug in your system installation.

Curses-based systems date back to an era when people connected to a computer via dumb terminals. Each terminal needed different control character sequences to tell it what to do. These differences are handled by curses, which picks the right definition based on your *TERM* environment variable. If you have the wrong terminal definition, you may find that curses believes some fundamental services are unavailable to your application. In particular, if you use `xterm-color`, you are using a definition of xterm that dates back to 1996 and will see errors like this, where the critical point is thet a curses function returned an unexpected error (the "ERR" result).

```
Traceback (most recent call last):
    File "demo.py", line 18, in <module>
       Screen.wrapper(demo)
    File "./lib/python3.6/site-packages/asciimatics/screen.py", line 1162, in wrapper
       unicode_aware=unicode_aware)
    File "./lib/python3.6/site-packages/asciimatics/screen.py", line 1131, in open
        unicode_aware=unicode_aware)
    File "./lib/python3.6/site-packages/asciimatics/screen.py", line 2001, in __init__
        curses.curs_set(0)
_curses.error: curs_set() returned ERR
```

The fix is to use a more modern terminal definition like `xterm` or `xterm-256color`.

## 7.3 256 colours not working

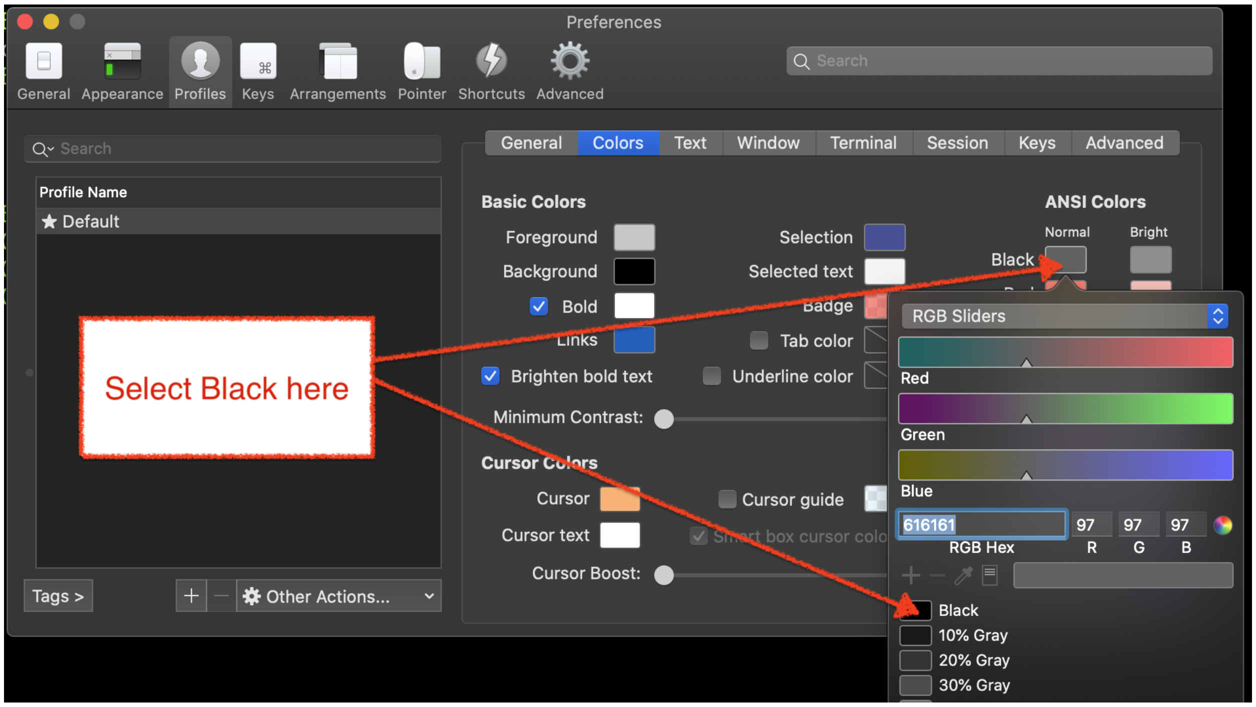
By default a lot of terminals will only support 8/16 colours. Windows users are limited to just these options for a native Windows console. However other systems can enable extra colours by picking a terminal that supports the extended colour palettes. For details of which terminals support additional colours and how to enable them see this Wikipedia article.

In most cases, simply selecting a terminal type of `xterm-256color` will usually do the trick these days.

## 7.4 My colours are wrong

When picking colours you may find that your selection doesn't have the desired effect. This is because terminals have a nasty habit of using different definitions of the standard colours.

Asciimatics relies on the ANSI colour set for its standard settings. As you can see here each terminal has its own interpretation of the exact colour. However, you can usually tweak that in your terminal settings. For example, iTerm on Mac uses a dark grey for black, which you can change as shown below.



## 7.5 The color theme resets when I resize the terminal

There was a bug where asciimatics would not maintain its own colour themes on resize. This has been fixed as of early 2020. You should just upgrade to the latest version to fix this.

However, there are also some other applications that change the terminal colour scheme on startup using the terminal's control sequences. These will not be invoked by asciimatics on a resize event. If you use such an application, you will need to invoke the control sequences yourself.

For example, to re-apply a pywal color theme:

```python
from pathlib import Path
from asciimatics.screen import ManagedScreen

with ManagedScreen() as screen:
    # do stuff
    if screen.has_resized():
        wal_sequences = Path.home() / ".cache" / "wal" / "sequences"
        try:
            with wal_sequences.open("rb") as fd:
                contents = fd.read()
                sys.stdout.buffer.write(contents)
        except Exception:
            pass
```

## 7.6 Mouse support not working

### 7.6.1 Curses systems

Mouse support isn't fully enabled by default on all terminal types. This will often require some extra extensions to be enabled as described here. In addition, if you want 256 colours, you will need to mix modes as described here.

Although it is possible to get Linux terminals to report all mouse movement, the reporting of mouse buttons along with movement appears to be highly erratic. The best reporting appears to be using the button event mode - i.e. mixing `xterm-1002` with `xterm-256color`.

### 7.6.2 Windows

Asciimatics will reprogram the Windows console to report mouse events on start-up. However, it is possible to change this while the application is running. In particular, if you switch on QuickEdit mode, Windows will stop reporting mouse events and process them itself. It is not possible to have both, so if you want to use the mouse in your app, please switch off QuickEdit mode.

## 7.7 Windows title does not change

Much like mouse support, the commands to set the window title is not supported on all terminal types. Windows should work without any changes. Other systems may need to use a similar method as above to mix modes to add status line support as described here.

## 7.8 Why can't I detect all key combinations?

Asciimatics is designed to run in terminals. These date back decades and so have many restrictions that were perfectly reasonable back then, but seem incomprehensible these days. The biggest one by a long way is that they can't detect exactly what the user types, even though computers have been able to detect exactly which key is being pressed (and when it is released) for a very long time.

The fundamental reason for this limitation is that there is no standard for the terminal emulators to tell the application more than a limited set of keys (typically lower and upper case characters, a random smattering of special keys and then some basic control/shift modifiers). While some terminals do try to do a little better, the coverage is patchy and inconsistent.

Until this improves the only way to get this sort of information is to run a system-wide keyboard hook, using something the the *keyboard* package on pypi. Unfortunately, this requires that you run as root and will not work on remote terminal sessions (e.g. over an SSH connection).

For more details, you can read https://invisible-island.net/ncurses/ncurses.faq.html#modified_keys or https://blog.robertelder.org/detect-keyup-event-linux-terminal/.

## 7.9 Ctrl+S does not work

In order to maintain legacy support for real terminal systems, most terminals/consoles still support software flow control using Ctrl+S/Ctrl+Q. You can switch this off on Linux by typing `stty -ixon` in your shell before you start asciimatics as explained here. Sadly, there is nothing that can be done on Windows to prevent this as it is built in to the operating system, so you will never be able to detect the Ctrl+S key. See here for details.

## 7.10 Backspace or delete are not working

Some users have reported this on curses systems. So far this has been tracked down to issues with the terminal configuration. For an in-depth explanation of the problem and several possible solutions see here and here.

This seems to be particularly problematic for Mac OS X users, where the default terminal app as shipped with the OS doesn't match the terminfo definitions. Genius! If you're on OSX, running the following inside your terminal *should* fix up the mismatch.

```
infocmp "$TERM" | sed -Ee 's/(kbs)=[^,]*/\1=\\177/' -e 's/(kdch1)=[^,]*/\1=\\E[3~/' >
↪"$TERM"
tic "$TERM"
rm -f "$TERM"
```

In an attempt to minimize the number of affected platforms, asciimatics v1.9.0 and later will also check the OS terminal definitions for ERASE and use that for backspace if it differs from the curses terminal definition.

## 7.11 There's a big delay when I press Escape

This is a well-known default operation for ncurses. As documented here you need to set the *ESCDELAY* environment variable before opening the Screen so that ncurses uses a shorter delay.

## 7.12 I can't run it inside PyCharm or other IDEs

Depending on which version you're using, you may see pywintypes.error 6 (ERROR_INVALID_HANDLE), or simply nothing (i.e. it looks like the program has hung). The reason for this is that the IDE Terminal/Console is not a true native terminal/console and so the native interfaces used by asciimatics will not work. There are 2 workarounds.

1. PyCharm, both Professional and Community editions, now offers an option to emulate console output directly in PyCharm. To enable this functionality, see *Run* | *Edit Configurations* | *Configuration* | *Execution* | *Emulate terminal in output console*.
2. Alternatively, you can just run asciimatics inside a real terminal or window - i.e. not inside PyCharm/the IDE. For example, you can force a real console from the Terminal window using *start cmd /c "python <your file name>"*.

## 7.13 It runs differently/does not work inside PyCharm

While PyCharm's terminal support is constantly improving, it is not perfect and so there are still cases where the imperfections cause glitches in asciimatics applications. This is most common on Windows (e.g. unexpected keys detected on resizing), but it is not just a Windows issue.

If you hit such glitches, the first thing to do is check whether you can also see them on a native terminal/console window of your operating system. If you can't reproduce on the native terminal/console, you need to raise the bug with JetBrains.

## 7.14 Unicode characters are not working

### 7.14.1 Curses systems

Most modern versions of Linux/OSX come with a good selection of glyphs supported as standard. The most likely issue is that you are not using a UTF-8 locale.

To set this up, follow the instructions here for OSX or here for Linux.

If that doesn't solve the problem and you are seeing unexpected lines in your block drawing characters, you are using a Terminal with extra spacing between your lines.

OSX allows you to edit the spacing as explained here, but Linux users will probably need to install a different terminal as explained here. I have found that *rxvt-unicode-256color* is most likely to work.

### 7.14.2 Windows

On Windows systems, there are a couple of potential issues. The first is that you might be using the wrong code page. Windows comes with many code pages. By default, asciimatics will only enable unicode features if you are using code page 65001 (the UTF-8 code page). You can fix this issue by running:

```
chcp 65001
```

If this does not solve the issue, the next possibility is that you may be using the Lucida Console or Consolas fonts. These do not have a full enough range of glyphs to support all the unicode output that asciimatics can generate.

To fix this issue, you need to download a font with a wider range of glyphs and then install them as the default for your command prompt. Details of how to do that are available here. I recommend that you use the DejaVu Mono font, which you can extract from the ZIP file from the download page - it is DejaVuSansMono.ttf in the TTF folder of the ZIP.

## 7.15 Redirecting STDIN

Generally speaking, it is not recommended that you try to do this as it will prevent asciimatics from being able to read the terminal input. However, if you must do this, this question on StackOverflow should give you some help on how to reconnect terminal input on curses based systems.

## 7.16 It's just not working at all

Some people have reported truly strange issues where things simply don't start up at all. Symptoms vary wildly from blank screens to other applications or tests running instead.

If you are hitting something like this, check that you haven't created a file called `test.py` in your project. This is because the `future` package, which asciimatics uses for compatibility with Python 2 and 3, imports the test package. If you happen to have a file called `test.py` in your project, this import could pick up your file instead of the built-in package.

Shout out to Andrew Penniman for spotting and solving this one!

## 7.17 It's too slow!

When people say this, they either mean that asciimatics is using too much CPU, or that it is unresponsive in some scenario. Either way, the solution is to reduce the work being done behind the scenes for your application. At a high-level you have 3 options.

1. Switch off any animations you don't need.
2. Move to a more responsive input loop.
3. Use a faster implementation of the underlying infrastructure.

Lets look at these options in more detail...

### 7.17.1 Switch off animations

The key to this optimization is to understand what you're drawing when. The biggest cost in the mainline loop is the actual re-drawing of all the content to the double-buffers, so asciimatics only does this when it knows something has, or may have, changed. You give hints to asciimatics as you construct your application - for example the rate at which a `Print` Effect needs to redraw, or whether you want to minimize CPU usage inside a `Frame`. It then uses these hints and the current application state to decide whether to draw a new frame into the double-buffer.

The first thing to look at is things that are not actually changing. For example if you use the `Print` Effect to display a static piece of text (like a `FigletText` renderer), the output never changes and so you only need to draw it once. in such cases, you should tell the Effect that it is pointless to refresh by specifying `speed=0` on construction.

Next you should consider removing unnecessary Effects. This is only really an option for TUI systems. Simply avoid adding other `Effects` into your `Scene` and keep it down the to the `Frame` for your user input.

Finally, consider switching off the cursor animation if you really need to minimize CPU usage. You can do this by setting `reduce_cpu=True` when constructing your `Frame`.

### 7.17.2 Input responsiveness

First things first, you should make sure that you're running at least version 1.11. Once you have that installed, you can use the `allow_int` option in *`play()`* to permit mouse and keyboard input to interrupt the normal frame refresh rate.

This should prevent users from seeing any delay in refreshes when they press a key. However there is a downside to this option - it will slightly mess up the timings for any animations, so it is only recommended to use it in TUI applications.

### 7.17.3 Use faster infrastructure

Asciimatics needs to do a lot of array manipulation in order to provide equivalent features to ncurses. In v1.11, I benchmarked various options and optimized the buffering to use the fastest version. If you haven't already moved to that version (or later), please do that now.

From here you have the usual options to speed up such calculations further.

1. Use `numpy` - which is a native C package to optimize array calculations
2. Use `pypy` - which is an optimized version of the Python language.

Right now, asciimatics doesn't support `numpy`, because I only got marginal gains when I made the prototype for 1.11. However, I got significant improvements from `pypy` and so I'd definitely recommend considering this option.

For example, running some samples for 20s on my test machine, I got the following results:

| julia.py | Average CPU |
|---|---|
| Python 2.7 (1.10) | 54.8% |
| Python 2.7 (1.11) | 47.8% |
| Pypy 6.0 | 20.0% |

| experimental.py | Average CPU |
|---|---|
| Python 2.7 (1.10) | 100.0% |
| Python 2.7 (1.11) | 71.4% |
| Pypy 6.0 | 34.3% |

Note that the v1.10 test for experimental.py was actually CPU-bound and so slow it was visibly juddering.

CHAPTER 8

# asciimatics

## 8.1 asciimatics package

### 8.1.1 Subpackages

#### asciimatics.renderers package

#### Submodules

#### asciimatics.renderers.base module

This module provides common code for all Renderers.

`asciimatics.renderers.base.`**`ATTRIBUTES = {'1': 1, '2': 2, '3': 3, '4': 4}`**

Attribute conversion table for the ${c,a} form of attributes for `paint`.

**`class`** `asciimatics.renderers.base.`**`DynamicRenderer`**(*height*, *width*, *clear=True*)

Bases: `asciimatics.renderers.base.Renderer`

A DynamicRenderer is a Renderer that creates each image as requested. It has a defined maximum size on construction.

**Parameters**

- **`height`** – The max height of the rendered image.
- **`width`** – The max width of the rendered image.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.base.**Renderer**

Bases: object

A Renderer is simply a class that will return one or more text renderings for display by an Effect.

In the simple case, this can be a single string that contains some unchanging content - e.g. a simple text message.

It can also represent a sequence of strings that can be played one after the other to make a simple animation sequence - e.g. a rotating globe.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.base.**StaticRenderer**(*images=None*, *animation=None*)

Bases: *asciimatics.renderers.base.Renderer*

A StaticRenderer is a Renderer that can create all possible images in advance. After construction the images will not change, but can by cycled for animation purposes.

This class will also convert text like ${c,a,b} into colour c, attribute a and background b for any subsequent text in the line, thus allowing multi-coloured text. The attribute and background are optional.

**Parameters**

- **images** – An optional set of ascii images to be rendered.
- **animation** – A function to pick the image (from images) to be rendered for any given frame.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.box module

This module implements an ASCII box renderer.

**class** `asciimatics.renderers.box.`**`Box`**(*width*, *height*, *uni=False*, *style=1*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

Renders a simple box using ASCII characters. This does not render in extended box drawing characters as that requires non-ASCII characters in Windows and direct access to curses in Linux.

**Parameters**

- **`width`** – width of box
- **`height`** – height of box
- **`uni`** – True to use UNICODE character set, defaults to False
- **`style`** – desired line style, based on line style definitions in *`constants`*: *ASCII_LINE*, *SINGLE_LINE*, *DOUBLE_LINE*. *uni* parameter takes precedence and the style will be ignored if *uni==False*

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.charts module

This module implements bar chart renderers.

**class** `asciimatics.renderers.charts.`**`BarChart`**(*height*, *width*, *functions*, *char='#'*, *colour=2*, *bg=0*, *gradient=None*, *scale=None*, *axes=2*, *intervals=None*, *labels=False*, *border=True*, *keys=None*, *gap=None*)

Bases: `asciimatics.renderers.charts._BarChartBase`

Renderer to create a horizontal bar chart using the specified functions as inputs for each entry. Can be used to chart distributions or for more graphical effect - e.g. to imitate a sound equalizer or a progress indicator.

**Parameters**

- **`height`** – The max height of the rendered image.
- **`width`** – The max width of the rendered image.
- **`functions`** – List of functions to chart.
- **`char`** – Character to use for the bar. Defaults to '#'
- **`colour`** – Colour(s) to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to green.

- **`bg`** – Background colour to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to black.
- **`gradient`** – Colour gradient to use for the bars. This is a list of tuple pairs specifying a threshold and a colour, or triplets to include a background colour too. Defaults to no gradients.
- **`scale`** – Maximum value for the bars. This is used to scale the function values to the maximum space available. Any value over this will be truncated when drawn. Defaults to the number of available characters in the chart.
- **`axes`** – Which axes to draw.
- **`intervals`** – Units for interval markers on the main axis. Defaults to none.
- **`labels`** – Whether to draw size indication labels on the x-axis.
- **`border`** – Whether to draw a border around the chart.
- **`keys`** – Optional keys to name each bar on the y-axis.
- **`gap`** – distance between bars. A value of None will auto-calculate (default).

If the scale parameter is not specified, the maximum length of the bar is based on the available space. A chart with no borders, no axes, no keys or labels will have a bar length based solely on the width of the graph.

- Borders use 4 characters height and 6 characters width
- Keys use the width of the widest key plus 1
- Labels use a height of 1
- An X_AXIS uses a height of 1
- A Y_AXIS uses a width of 1

**`axes_style`**

The current drawing style of the axes. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters
- *SINGLE_LINE* – UNICODE based single line

Note that your canvas must support UNICODE style characters to use them

**`border_style`**

The current drawing style of the border. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters
- *SINGLE_LINE* – UNICODE based single line
- *DOUBLE_LINE* – UNICODE based double line

Note that your canvas must support UNICODE style characters to use them

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.charts.`**`VBarChart`**(*height*, *width*, *functions*, *char='#'*, *colour=2*, *bg=0*, *gradient=None*, *scale=None*, *axes=1*, *intervals=None*, *labels=False*, *border=True*, *keys=None*, *gap=None*)

Bases: `asciimatics.renderers.charts._BarChartBase`

Renderer to create a vertical bar chart using the specified functions as inputs for each entry. Can be used to chart distributions or for more graphical effect - e.g. to imitate a sound equalizer or a progress indicator.

**Parameters**

- **height** – The max height of the rendered image.
- **width** – The max width of the rendered image.
- **functions** – List of functions to chart.
- **char** – Character to use for the bar. Defaults to '#'
- **colour** – Colour(s) to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to green.
- **bg** – Background colour to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to black.
- **gradient** – Colour gradient to use for the bars. This is a list of tuple pairs specifying a threshold and a colour, or triplets to include a background colour too. Defaults to no gradients.
- **scale** – Maximum value for the bars. This is used to scale the function values to the maximum space available. Any value over this will be truncated when drawn. Defaults to the number of available characters in the chart.
- **axes** – Which axes to draw.
- **intervals** – Units for interval markers on the main axis. Defaults to none.
- **labels** – Whether to draw size indication labels on the y-axis.
- **border** – Whether to draw a border around the chart.
- **keys** – Optional keys to name each bar on the x-axis.
- **gap** – distance between bars. A value of None will auto-calculate (default). Minimum value when auto-calculated is 1, for no gaps specify 0.

If the scale parameter is not specified, the maximum length of the bar is based on the available space. A chart with no borders, no axes, no keys or labels will have a bar height based solely on the width of the graph.

- Borders use 4 characters height and 6 characters width
- Keys use a height of 1
- Labels vertical bar chart use the width of the widest label plus 1 (label values depend on the scale of the chart)
- An X_AXIS uses a height of 1
- A Y_AXIS uses a width of 1

**axes_style**

The current drawing style of the axes. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters

- *SINGLE_LINE* – UNICODE based single line

Note that your canvas must support UNICODE style characters to use them

**`border_style`**
The current drawing style of the border. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters
- *SINGLE_LINE* – UNICODE based single line
- *DOUBLE_LINE* – UNICODE based double line

Note that your canvas must support UNICODE style characters to use them

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.figlettext module

This module implements Figlet text renderer.

**class** `asciimatics.renderers.figlettext.`**`FigletText`**(*text*, *font='standard'*, *width=200*)
Bases: `asciimatics.renderers.base.StaticRenderer`

This class renders the supplied text using the specified Figlet font. See http://www.figlet.org/ for details of available fonts.

**Parameters**

- **`text`** – The text string to convert with Figlet.
- **`font`** – The Figlet font to use (optional).
- **`width`** – The maximum width for this text in characters.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.fire module

This module implements a fire effect renderer.

**class** `asciimatics.renderers.fire.`**`Fire`**(*height*, *width*, *emitter*, *intensity*, *spot*, *colours*, *bg=False*)

Bases: *`asciimatics.renderers.base.DynamicRenderer`*

Renderer to create a fire effect based on a specified *emitter* that defines the heat source.

The implementation here uses the same techniques described in http://freespace.virgin.net/hugo.elias/models/m_fire.htm, although a slightly different implementation.

**Parameters**

- **`height`** – Height of the box to contain the flames.
- **`width`** – Width of the box to contain the flames.
- **`emitter`** – Heat source for the flames. Any non-whitespace character is treated as part of the heat source.
- **`intensity`** – The strength of the flames. The bigger the number, the hotter the fire. 0 <= intensity <= 1.0.
- **`spot`** – Heat of each spot source. Must be an integer > 0.
- **`colours`** – Number of colours the screen supports.
- **`bg`** – (Optional) Whether to render background colours only.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

`asciimatics.renderers.fire.`**`random`**() → x in the interval [0, 1).

### asciimatics.renderers.images module

This module implements renderers that produce content based on image files.

**class** `asciimatics.renderers.images.`**`ColourImageFile`**(*screen*, *filename*, *height=30*, *bg=0*, *fill_background=False*, *uni=False*, *dither=False*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

Renderer to convert an image file (as supported by the Python Imaging Library) into an block image of available colours.

**Warning:** This is only compatible with 256-colour terminals. Results in other terminals with reduced colour capabilities are severely restricted. Since Windows only has 8 base colours, it is recommended that you avoid this renderer on that platform.

**Parameters**

- **`screen`** – The screen to use when displaying the image.
- **`filename`** – The name of the file to render.
- **`height`** – The height of the text rendered image.
- **`bg`** – The default background colour for this image.
- **`fill_background`** – Whether to set background colours too.
- **`uni`** – Whether to use unicode box characters or not.
- **`dither`** – Whether to dither the rendered image or not.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.images.`**`ImageFile`**(*filename*, *height=30*, *colours=8*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

Renderer to convert an image file (as supported by the Python Imaging Library) into an ascii grey scale text image.

**Parameters**

- **`filename`** – The name of the file to render.
- **`height`** – The height of the text rendered image.
- **`colours`** – The number of colours the terminal supports.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.kaleidoscope module

This module implements a kaeldioscope effect renderer.

**class** `asciimatics.renderers.kaleidoscope.`**`Kaleidoscope`**(*height*, *width*, *cell*, *symmetry*)

Bases: *`asciimatics.renderers.base.DynamicRenderer`*

Renderer to create a 2-mirror kaleidoscope effect.

This is a chained renderer (i.e. it acts upon the output of another Renderer which is passed to it on construction). The other Renderer is used as the cell that is rotated over time to create the animation.

You can specify the desired rotational symmetry of the kaleidoscope (which determines the angle between the mirrors). If you chose values of less than 2, you are effectively removing one or both mirrors, thus either getting the original cell or a simple mirrored image of the cell.

Since this renderer rotates the background cell, it needs operate on square pixels, which means each character in the cell is drawn as 2 next to each other on the screen. In other words the cell needs to be half the width of the desired output (when measured in text characters).

**Parameters**

- **`height`** – Height of the box to contain the kaleidoscope.
- **`width`** – Width of the box to contain the kaleidoscope.
- **`cell`** – A Renderer to use as the backing cell for the kaleidoscope.
- **`symmetry`** – The desired rotational symmetry. Must be a non-negative integer.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.old module

### asciimatics.renderers.plasma module

This module implements a plasma effect renderer.

**class** `asciimatics.renderers.plasma.`**`Plasma`**(*height*, *width*, *colours*)

Bases: *`asciimatics.renderers.base.DynamicRenderer`*

Renderer to create a "plasma" effect using sinusoidal functions.

The implementation here uses the same techniques described in http://lodev.org/cgtutor/plasma.html

**Parameters**

- **`height`** – Height of the box to contain the plasma.
- **`width`** – Width of the box to contain the plasma.

- **colours** – Number of colours the screen supports.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

## asciimatics.renderers.players module

This module implements renderers that play content to the screen.

**class** asciimatics.renderers.players.**AbstractScreenPlayer**(*height*, *width*)

Bases: *asciimatics.renderers.base.DynamicRenderer*

Abstract renderer to play terminal text with support for ANSI control codes.

**Parameters**

- **height** – required height of the renderer.
- **width** – required width of the renderer.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.players.**AnsiArtPlayer**(*filename*, *height=25*, *width=80*, *encoding='cp437'*, *strip=False*, *rate=2*)

Bases: *asciimatics.renderers.players.AbstractScreenPlayer*

Renderer to play ANSI art text files.

In order to tidy up files, this must be used as a context manager (i.e. using *with*).

**Parameters**

- **filename** – the file containingi the ANSI art.
- **height** – required height of the renderer.
- **width** – required width of the renderer.
- **encoding** – text encoding ofnthe file.

- **strip** – whether to strip CRLF from the file content.
- **rate** – number of lines to render on each update.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.players.**AsciinemaPlayer**(*filename*, *height=None*, *width=None*, *max_delay=None*)

Bases: *asciimatics.renderers.players.AbstractScreenPlayer*

Renderer to play terminal recordings created by asciinema.

This only supports the version 2 file format. Use the max_delay setting to speed up human interactions (i.e. to reduce delays from typing).

In order to tidy up files, this must be used as a context manager (i.e. using *with*).

**Parameters**

- **filename** – the file containingi the ANSI art.
- **height** – required height of the renderer.
- **width** – required width of the renderer.
- **max_delay** – maximum time interval (in secs) to wait between frame updates.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.rainbow module

This module implements a rainbow effect renderer.

**class** asciimatics.renderers.rainbow.**Rainbow**(*screen*, *renderer*)

Bases: *asciimatics.renderers.base.StaticRenderer*

Chained renderer to add rainbow colours to output of another renderer. The embedded rendered must not use multi-colour mode (i.e. ${c,a} mark-ups) as these will be converted to explicit text by this renderer.

**Parameters**

- **screen** – The screen object for this renderer.
- **renderer** – The renderer to wrap.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

## asciimatics.renderers.rotatedduplicate module

This module implements a renderer that renders another renderer but rotated.

**class** asciimatics.renderers.rotatedduplicate.**RotatedDuplicate**(*width*, *height*, *renderer*)

Bases: *asciimatics.renderers.base.StaticRenderer*

Chained renderer to add a rotated version of the original renderer underneath and centre the whole thing within within the specified dimensions.

**Parameters**

- **width** – The maximum width of the rendered text.
- **height** – The maximum height of the rendered text.
- **renderer** – The renderer to wrap.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

## asciimatics.renderers.scales module

This module implements renderers that show measuring scales to the screen.

**class** asciimatics.renderers.scales.**Scale**(*width*)

Bases: *asciimatics.renderers.base.StaticRenderer*

This renders a linear scale, useful for debugging positions of your creations. Every 5 spaces gets a tick mark, every 10 a number.

**Parameters** **`width`** – The width of the scale

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.scales.`**`VScale`**(*height*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

This renders a vertical linear scale, useful for debugging positions of your creations. Writes lowest significant digit of a count running vertically.

**Parameters** **`width`** – The width of the scale

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.renderers.speechbubble module

This module implements a speech-bubble effect renderer.

**class** `asciimatics.renderers.speechbubble.`**`SpeechBubble`**(*text*, *tail=None*, *uni=False*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

Renders supplied text into a speech bubble.

**Parameters**

- **`text`** – The text to be put into a speech bubble.
- **`tail`** – Where to put the bubble callout tail, specifying "L" or "R" for left or right tails. Can be None for no tail.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

### Module contents

This module provides *Renderers* to create complex animation effects. For more details see http://asciimatics.readthedocs.io/en/latest/rendering.html

**class** asciimatics.renderers.**Renderer**

Bases: object

A Renderer is simply a class that will return one or more text renderings for display by an Effect.

In the simple case, this can be a single string that contains some unchanging content - e.g. a simple text message.

It can also represent a sequence of strings that can be played one after the other to make a simple animation sequence - e.g. a rotating globe.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**StaticRenderer**(*images=None*, *animation=None*)

Bases: *asciimatics.renderers.base.Renderer*

A StaticRenderer is a Renderer that can create all possible images in advance. After construction the images will not change, but can by cycled for animation purposes.

This class will also convert text like ${c,a,b} into colour c, attribute a and background b for any subsequent text in the line, thus allowing multi-coloured text. The attribute and background are optional.

**Parameters**

- **images** – An optional set of ascii images to be rendered.
- **animation** – A function to pick the image (from images) to be rendered for any given frame.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.`**`DynamicRenderer`**(*height*, *width*, *clear=True*)

Bases: *`asciimatics.renderers.base.Renderer`*

A DynamicRenderer is a Renderer that creates each image as requested. It has a defined maximum size on construction.

**Parameters**

- **`height`** – The max height of the rendered image.
- **`width`** – The max width of the rendered image.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.`**`Box`**(*width*, *height*, *uni=False*, *style=1*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

Renders a simple box using ASCII characters. This does not render in extended box drawing characters as that requires non-ASCII characters in Windows and direct access to curses in Linux.

**Parameters**

- **`width`** – width of box
- **`height`** – height of box
- **`uni`** – True to use UNICODE character set, defaults to False
- **`style`** – desired line style, based on line style definitions in *`constants`*: *ASCII_LINE*, *SINGLE_LINE*, *DOUBLE_LINE*. *uni* parameter takes precedence and the style will be ignored if *uni==False*

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.`**`BarChart`**(*height*, *width*, *functions*, *char='#'*, *colour=2*, *bg=0*, *gradient=None*, *scale=None*, *axes=2*, *intervals=None*, *labels=False*, *border=True*, *keys=None*, *gap=None*)

Bases: `asciimatics.renderers.charts._BarChartBase`

Renderer to create a horizontal bar chart using the specified functions as inputs for each entry. Can be used to chart distributions or for more graphical effect - e.g. to imitate a sound equalizer or a progress indicator.

**Parameters**

- **`height`** – The max height of the rendered image.
- **`width`** – The max width of the rendered image.
- **`functions`** – List of functions to chart.
- **`char`** – Character to use for the bar. Defaults to '#'
- **`colour`** – Colour(s) to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to green.
- **`bg`** – Background colour to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to black.
- **`gradient`** – Colour gradient to use for the bars. This is a list of tuple pairs specifying a threshold and a colour, or triplets to include a background colour too. Defaults to no gradients.
- **`scale`** – Maximum value for the bars. This is used to scale the function values to the maximum space available. Any value over this will be truncated when drawn. Defaults to the number of available characters in the chart.
- **`axes`** – Which axes to draw.
- **`intervals`** – Units for interval markers on the main axis. Defaults to none.
- **`labels`** – Whether to draw size indication labels on the x-axis.
- **`border`** – Whether to draw a border around the chart.
- **`keys`** – Optional keys to name each bar on the y-axis.
- **`gap`** – distance between bars. A value of None will auto-calculate (default).

If the scale parameter is not specified, the maximum length of the bar is based on the available space. A chart with no borders, no axes, no keys or labels will have a bar length based solely on the width of the graph.

- Borders use 4 characters height and 6 characters width
- Keys use the width of the widest key plus 1
- Labels use a height of 1
- An X_AXIS uses a height of 1
- A Y_AXIS uses a width of 1

**`axes_style`**

The current drawing style of the axes. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters
- *SINGLE_LINE* – UNICODE based single line

Note that your canvas must support UNICODE style characters to use them

**`border_style`**

The current drawing style of the border. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters
- *SINGLE_LINE* – UNICODE based single line

- *DOUBLE_LINE* – UNICODE based double line

Note that your canvas must support UNICODE style characters to use them

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**VBarChart**(*height*, *width*, *functions*, *char='#'*, *colour=2*, *bg=0*, *gradient=None*, *scale=None*, *axes=1*, *intervals=None*, *labels=False*, *border=True*, *keys=None*, *gap=None*)

Bases: `asciimatics.renderers.charts._BarChartBase`

Renderer to create a vertical bar chart using the specified functions as inputs for each entry. Can be used to chart distributions or for more graphical effect - e.g. to imitate a sound equalizer or a progress indicator.

**Parameters**

- **height** – The max height of the rendered image.
- **width** – The max width of the rendered image.
- **functions** – List of functions to chart.
- **char** – Character to use for the bar. Defaults to '#'
- **colour** – Colour(s) to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to green.
- **bg** – Background colour to use for the bars. This can be a single value or list of values (to cycle around for each bar). Defaults to black.
- **gradient** – Colour gradient to use for the bars. This is a list of tuple pairs specifying a threshold and a colour, or triplets to include a background colour too. Defaults to no gradients.
- **scale** – Maximum value for the bars. This is used to scale the function values to the maximum space available. Any value over this will be truncated when drawn. Defaults to the number of available characters in the chart.
- **axes** – Which axes to draw.
- **intervals** – Units for interval markers on the main axis. Defaults to none.
- **labels** – Whether to draw size indication labels on the y-axis.
- **border** – Whether to draw a border around the chart.
- **keys** – Optional keys to name each bar on the x-axis.
- **gap** – distance between bars. A value of None will auto-calculate (default). Minimum value when auto-calculated is 1, for no gaps specify 0.

If the scale parameter is not specified, the maximum length of the bar is based on the available space. A chart with no borders, no axes, no keys or labels will have a bar height based solely on the width of the graph.

- Borders use 4 characters height and 6 characters width
- Keys use a height of 1
- Labels vertical bar chart use the width of the widest label plus 1 (label values depend on the scale of the chart)
- An X_AXIS uses a height of 1
- A Y_AXIS uses a width of 1

**`axes_style`**
The current drawing style of the axes. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters
- *SINGLE_LINE* – UNICODE based single line

Note that your canvas must support UNICODE style characters to use them

**`border_style`**
The current drawing style of the border. Possible values are defined in `constants`:

- *ASCII_LINE* – ASCII safe characters
- *SINGLE_LINE* – UNICODE based single line
- *DOUBLE_LINE* – UNICODE based double line

Note that your canvas must support UNICODE style characters to use them

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.`**`FigletText`**(*text*, *font='standard'*, *width=200*)
Bases: `asciimatics.renderers.base.StaticRenderer`

This class renders the supplied text using the specified Figlet font. See http://www.figlet.org/ for details of available fonts.

**Parameters**

- **`text`** – The text string to convert with Figlet.
- **`font`** – The Figlet font to use (optional).
- **`width`** – The maximum width for this text in characters.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**Fire**(*height*, *width*, *emitter*, *intensity*, *spot*, *colours*, *bg=False*)

Bases: *asciimatics.renderers.base.DynamicRenderer*

Renderer to create a fire effect based on a specified *emitter* that defines the heat source.

The implementation here uses the same techniques described in http://freespace.virgin.net/hugo.elias/models/m_fire.htm, although a slightly different implementation.

**Parameters**

- **height** – Height of the box to contain the flames.
- **width** – Width of the box to contain the flames.
- **emitter** – Heat source for the flames. Any non-whitespace character is treated as part of the heat source.
- **intensity** – The strength of the flames. The bigger the number, the hotter the fire. 0 <= intensity <= 1.0.
- **spot** – Heat of each spot source. Must be an integer > 0.
- **colours** – Number of colours the screen supports.
- **bg** – (Optional) Whether to render background colours only.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**ImageFile**(*filename*, *height=30*, *colours=8*)

Bases: *asciimatics.renderers.base.StaticRenderer*

Renderer to convert an image file (as supported by the Python Imaging Library) into an ascii grey scale text image.

**Parameters**

- **filename** – The name of the file to render.
- **height** – The height of the text rendered image.
- **colours** – The number of colours the terminal supports.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**ColourImageFile**(*screen*, *filename*, *height=30*, *bg=0*, *fill_background=False*, *uni=False*, *dither=False*)

Bases: *asciimatics.renderers.base.StaticRenderer*

Renderer to convert an image file (as supported by the Python Imaging Library) into an block image of available colours.

**Warning:** This is only compatible with 256-colour terminals. Results in other terminals with reduced colour capabilities are severely restricted. Since Windows only has 8 base colours, it is recommended that you avoid this renderer on that platform.

**Parameters**

- **screen** – The screen to use when displaying the image.
- **filename** – The name of the file to render.
- **height** – The height of the text rendered image.
- **bg** – The default background colour for this image.
- **fill_background** – Whether to set background colours too.
- **uni** – Whether to use unicode box characters or not.
- **dither** – Whether to dither the rendered image or not.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**AbstractScreenPlayer**(*height*, *width*)

Bases: *asciimatics.renderers.base.DynamicRenderer*

Abstract renderer to play terminal text with support for ANSI control codes.

**Parameters**

- **height** – required height of the renderer.
- **width** – required width of the renderer.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**AnsiArtPlayer**(*filename*, *height=25*, *width=80*, *encoding='cp437'*, *strip=False*, *rate=2*)

Bases: *asciimatics.renderers.players.AbstractScreenPlayer*

Renderer to play ANSI art text files.

In order to tidy up files, this must be used as a context manager (i.e. using *with*).

**Parameters**

- **filename** – the file containingi the ANSI art.
- **height** – required height of the renderer.
- **width** – required width of the renderer.
- **encoding** – text encoding ofnthe file.
- **strip** – whether to strip CRLF from the file content.
- **rate** – number of lines to render on each update.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**AsciinemaPlayer**(*filename*, *height=None*, *width=None*, *max_delay=None*)

Bases: *asciimatics.renderers.players.AbstractScreenPlayer*

Renderer to play terminal recordings created by asciinema.

This only supports the version 2 file format. Use the max_delay setting to speed up human interactions (i.e. to reduce delays from typing).

In order to tidy up files, this must be used as a context manager (i.e. using *with*).

**Parameters**

- **filename** – the file containingi the ANSI art.
- **height** – required height of the renderer.
- **width** – required width of the renderer.
- **max_delay** – maximum time interval (in secs) to wait between frame updates.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**Kaleidoscope**(*height*, *width*, *cell*, *symmetry*)

Bases: *asciimatics.renderers.base.DynamicRenderer*

Renderer to create a 2-mirror kaleidoscope effect.

This is a chained renderer (i.e. it acts upon the output of another Renderer which is passed to it on construction). The other Renderer is used as the cell that is rotated over time to create the animation.

You can specify the desired rotational symmetry of the kaleidoscope (which determines the angle between the mirrors). If you chose values of less than 2, you are effectively removing one or both mirrors, thus either getting the original cell or a simple mirrored image of the cell.

Since this renderer rotates the background cell, it needs operate on square pixels, which means each character in the cell is drawn as 2 next to each other on the screen. In other words the cell needs to be half the width of the desired output (when measured in text characters).

**Parameters**

- **height** – Height of the box to contain the kaleidoscope.
- **width** – Width of the box to contain the kaleidoscope.
- **cell** – A Renderer to use as the backing cell for the kaleidoscope.
- **symmetry** – The desired rotational symmetry. Must be a non-negative integer.

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**Plasma**(*height*, *width*, *colours*)

Bases: *asciimatics.renderers.base.DynamicRenderer*

Renderer to create a "plasma" effect using sinusoidal functions.

The implementation here uses the same techniques described in http://lodev.org/cgtutor/plasma.html

**Parameters**

- **height** – Height of the box to contain the plasma.

- **`width`** – Width of the box to contain the plasma.
- **`colours`** – Number of colours the screen supports.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.`**`Rainbow`**(*screen*, *renderer*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

Chained renderer to add rainbow colours to output of another renderer. The embedded rendered must not use multi-colour mode (i.e. ${c,a} mark-ups) as these will be converted to explicit text by this renderer.

**Parameters**

- **`screen`** – The screen object for this renderer.
- **`renderer`** – The renderer to wrap.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

**class** `asciimatics.renderers.`**`RotatedDuplicate`**(*width*, *height*, *renderer*)

Bases: *`asciimatics.renderers.base.StaticRenderer`*

Chained renderer to add a rotated version of the original renderer underneath and centre the whole thing within within the specified dimensions.

**Parameters**

- **`width`** – The maximum width of the rendered text.
- **`height`** – The maximum height of the rendered text.
- **`renderer`** – The renderer to wrap.

**`images`**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**Scale**(*width*)

Bases: *asciimatics.renderers.base.StaticRenderer*

This renders a linear scale, useful for debugging positions of your creations. Every 5 spaces gets a tick mark, every 10 a number.

**Parameters** **width** – The width of the scale

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**VScale**(*height*)

Bases: *asciimatics.renderers.base.StaticRenderer*

This renders a vertical linear scale, useful for debugging positions of your creations. Writes lowest significant digit of a count running vertically.

**Parameters** **width** – The width of the scale

**images**

**Returns** An iterator of all the images in the Renderer.

**max_height**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**max_width**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**rendered_text**

**Returns** The next image and colour map in the sequence as a tuple.

**class** asciimatics.renderers.**SpeechBubble**(*text*, *tail=None*, *uni=False*)

Bases: *asciimatics.renderers.base.StaticRenderer*

Renders supplied text into a speech bubble.

**Parameters**

- **text** – The text to be put into a speech bubble.
- **tail** – Where to put the bubble callout tail, specifying "L" or "R" for left or right tails. Can be None for no tail.

**images**

**Returns** An iterator of all the images in the Renderer.

**`max_height`**

**Returns** The max height of the rendered text (across all images if an animated renderer).

**`max_width`**

**Returns** The max width of the rendered text (across all images if an animated renderer).

**`rendered_text`**

**Returns** The next image and colour map in the sequence as a tuple.

### asciimatics.widgets package

### Submodules

### asciimatics.widgets.baselistbox module

This is the baseclass for list box types

### asciimatics.widgets.button module

This module defines a button widget

**class** `asciimatics.widgets.button.`**`Button`**(*text*, *on_click*, *label=None*, *add_box=True*, *name=None*, ***kwargs*)

Bases: `asciimatics.widgets.widget.Widget`

A Button widget to be displayed in a Frame.

It is typically used to represent a desired action for te user to invoke (e.g. a submit button on a form).

**Parameters**

- **`text`** – The text for the button.
- **`on_click`** – The function to invoke when the button is clicked.
- **`label`** – An optional label for the widget.
- **`add_box`** – Whether to wrap the text with chevrons.
- **`name`** – The name of this widget.

Also see the common keyword arguments in `Widget`.

**`blur`**()

Call this to take the input focus from this Widget.

**`custom_colour`**

A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**`disabled`**

Whether this widget is disabled or not.

**`focus`**()

Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)

Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**text**

The current text for this Button.

**update**(*frame_no*)

The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**

The current value for this Button.

**width**

The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.checkbox module

This module defines a checkbox widget

**class** asciimatics.widgets.checkbox.**CheckBox**(*text*, *label=None*, *name=None*, *on_change=None*, ***kwargs*)

Bases: `asciimatics.widgets.widget.Widget`

A CheckBox widget is used to ask for Boolean (i.e. yes/no) input.

It consists of an optional label (typically used for the first in a group of CheckBoxes), the box and a field name.

**Parameters**

- **text** – The text to explain this specific field to the user.
- **label** – An optional label for the widget.
- **name** – The internal name for the widget.
- **on_change** – Optional function to call when text changes.

Also see the common keyword arguments in `Widget`.

**blur**()

Call this to take the input focus from this Widget.

**custom_colour**

A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**`disabled`**
Whether this widget is disabled or not.

**`focus`**()
Call this to give this Widget the input focus.

**`frame`**
The Frame that contains this Widget.

**`frame_update_count`**
The number of frames before this Widget should be updated.

**`get_location`**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**`is_mouse_over`**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **`event`** – The MouseEvent to check.
- **`include_label`** – Include space reserved for the label when checking.
- **`width_modifier`** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**`is_tab_stop`**
Whether this widget is a valid tab stop for keyboard navigation.

**`is_valid`**
Whether this widget has passed its data validation or not.

**`is_visible`**
Whether this widget is visible on the Canvas or not.

**`label`**
The label for this widget. Can be *None*.

**`name`**
The name for this widget (for reference in the persistent data). Can be *None*.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_frame`**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **`frame`** – The owning Frame.

**`required_height`**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **`offset`** – The allowed width for any labels.
- **`width`** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update**(*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**
The current value for this Checkbox.

**width**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.datepicker module

This module defines a datepicker widget

**class** asciimatics.widgets.datepicker.**DatePicker**(*label=None*, *name=None*, *year_range=None*, *on_change=None*, ***kwargs*)
Bases: `asciimatics.widgets.widget.Widget`

A DatePicker widget allows you to pick a date from a compact, temporary, pop-up Frame.

**Parameters**

- **label** – An optional label for the widget.
- **name** – The name for the widget.
- **on_change** – Optional function to call when the selected time changes.

Also see the common keyword arguments in `Widget`.

**blur**()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**focus**()
Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)

Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update**(*frame_no*)

The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**

The current selected date.

**width**

The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

## asciimatics.widgets.divider module

This module defines a divider between widgets

**class** asciimatics.widgets.divider.**Divider**(*draw_line=True*, *height=1*, *line_char=None*)

Bases: `asciimatics.widgets.widget.Widget`

A divider to break up a group of widgets.

**Parameters**

- **draw_line** – Whether to draw a line in the centre of the gap.
- **height** – The required vertical gap.
- **line_char** – Optional character to use for drawing the line.

**blur**()

Call this to take the input focus from this Widget.

**custom_colour**

A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**

Whether this widget is disabled or not.

**focus**()

Call this to give this Widget the input focus.

**frame**

The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update** (*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**
The current value for this Divider.

**width**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

## asciimatics.widgets.dropdownlist module

This module defines a dropdown list widget

**class** asciimatics.widgets.dropdownlist.**DropdownList** (*options*, *label=None*, *name=None*, *on_change=None*, *fit=None*, ***kwargs*)

Bases: *asciimatics.widgets.widget.Widget*

This widget allows you to pick an item from a temporary pop-up list.

**Parameters**

- **options** – The options for each row in the widget.
- **label** – An optional label for the widget.
- **name** – The name for the widget.
- **on_change** – Optional function to call when the selected time changes.
- **fit** – Shrink width of dropdown to fit the width of options. Default False.

The *options* are a list of tuples, where the first value is the string to be displayed to the user and the second is an interval value to identify the entry to the program. For example:

options=[(“First option”, 1), (“Second option”, 2)]

Also see the common keyword arguments in *Widget*.

**blur** ()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**fit**
Whether to shrink to largest element width or not.

**`focus`**()
Call this to give this Widget the input focus.

**`frame`**
The Frame that contains this Widget.

**`frame_update_count`**
The number of frames before this Widget should be updated.

**`get_location`**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**`is_mouse_over`**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **`event`** – The MouseEvent to check.
- **`include_label`** – Include space reserved for the label when checking.
- **`width_modifier`** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**`is_tab_stop`**
Whether this widget is a valid tab stop for keyboard navigation.

**`is_valid`**
Whether this widget has passed its data validation or not.

**`is_visible`**
Whether this widget is visible on the Canvas or not.

**`label`**
The label for this widget. Can be *None*.

**`name`**
The name for this widget (for reference in the persistent data). Can be *None*.

**`options`**
The set of allowed options for the drop-down list.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_frame`**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **`frame`** – The owning Frame.

**`required_height`**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **`offset`** – The allowed width for any labels.
- **`width`** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the *Layout* class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update**(*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**
The current value for this DropdownList.

**width**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.filebrowser module

This module defines a file browser selection

**class** asciimatics.widgets.filebrowser.**FileBrowser**(*height*, *root*, *name=None*, *on_select=None*, *on_change=None*, *file_filter=None*)
Bases: *asciimatics.widgets.multicolumnlistbox.MultiColumnListBox*

A FileBrowser is a widget for finding a file on the local disk.

**Parameters**

- **height** – The desired height for this widget.
- **root** – The starting root directory to display in the widget.
- **name** – The name of this widget.
- **on_select** – Optional function that gets called when user selects a file (by pressing enter or double-clicking).
- **on_change** – Optional function that gets called on any movement of the selection.
- **file_filter** – Optional RegEx string that can be passed in to filter the files to be displayed.

Most people will want to use a filter to finx files with a particular extension. In this case, you must use a regex that matches to the end of the line - e.g. use ".*.txt$" to find files ending with ".txt". This ensures that you don't accidentally pick up files containing the filter.

**blur**()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**focus**()
Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**options**
The list of options available for user selection

This is a list of tuples ([<col 1 string>, . . . , <col n string>], <internal value>).

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame** (*frame*)

Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height** (*offset*, *width*)

Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset** ()

The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout** (*x*, *y*, *offset*, *w*, *h*)

Set the size and position of the Widget.

This should not be called directly. It is used by the *Layout* class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**start_line**

The line that will be drawn at the top of the visible section of this list.

**update** (*frame_no*)

The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**

The current value for this list box.

**width**

The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.frame module

This module defines a class to display widgets

**class** asciimatics.widgets.frame.**Frame** (*screen*, *height*, *width*, *data=None*, *on_load=None*, *has_border=True*, *hover_focus=False*, *name=None*, *title=None*, *x=None*, *y=None*, *has_shadow=False*, *reduce_cpu=False*, *is_modal=False*, *can_scroll=True*)

Bases: *asciimatics.effects.Effect*

A Frame is a special Effect for controlling and displaying Widgets.

It is similar to a window as used in native GUI applications. Widgets are text UI elements that can be used to create an interactive application within your Frame.

**Parameters**

- **`screen`** – The Screen that owns this Frame.
- **`width`** – The desired width of the Frame.
- **`height`** – The desired height of the Frame.
- **`data`** – optional data dict to initialize any widgets in the frame.
- **`on_load`** – optional function to call whenever the Frame reloads.
- **`has_border`** – Whether the frame has a border box. Defaults to True.
- **`hover_focus`** – Whether hovering a mouse over a widget (i.e. mouse move events) should change the input focus. Defaults to false.
- **`name`** – Optional name to identify this Frame. This is used to reset data as needed from on old copy after the screen resizes.
- **`title`** – Optional title to display if has_border is True.
- **`x`** – Optional x position for the top left corner of the Frame.
- **`y`** – Optional y position for the top left corner of the Frame.
- **`has_shadow`** – Optional flag to indicate if this Frame should have a shadow when drawn.
- **`reduce_cpu`** – Whether to minimize CPU usage (for use on low spec systems).
- **`is_modal`** – Whether this Frame is "modal" - i.e. will stop all other Effects from receiving input events.
- **`can_scroll`** – Whether a scrollbar should be available on the border, or not.

**`add_effect`**(*effect*)
Add an Effect to the Frame.

**Parameters** **`effect`** – The Effect to be added.

**`add_layout`**(*layout*)
Add a Layout to the Frame.

**Parameters** **`layout`** – The Layout to be added.

**`border_box`**
Instance of `BoxTool` that specifies the characters used to draw the border to this frame. You can change the border character style by modifying the `style` property on this object. Allowed styles are defined in `constants`.

**`canvas`**
The Canvas that backs this Frame.

**`clone`**(*_*, *scene*)
Create a clone of this Frame into a new Screen.

**Parameters**

- **`_`** – ignored.
- **`scene`** – The new Scene object to clone into.

**`data`**
Data dictionary containing values from the contained widgets.

**`delete_count`**
The number of frames before this Effect should be deleted.

**find_widget**(*name*)
Look for a widget with a specified name.

**Parameters** **name** – The name to search for.

**Returns** The widget that matches or None if one couldn't be found.

**fix**()
Fix the layouts and calculate the locations of all the widgets.

This function should be called once all Layouts have been added to the Frame and all widgets added to the Layouts.

**focussed_widget**
The widget that currently has the focus within this Frame.

**frame_update_count**
The number of frames before this Effect should be updated.

**get_scroll_pos**()
Get current position for scroll bar.

**move_to**(*x*, *y*, *h*)
Make the specified location visible. This is typically used by a widget to scroll the canvas such that it is visible.

**Parameters**

- **x** – The x location to make visible.
- **y** – The y location to make visible.
- **h** – The height of the location to make visible.

**palette = {}**
Colour palette for the widgets within the Frame. Each entry should be a 3-tuple of (foreground colour, attribute, background colour).

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**rebase_event**(*event*)
Rebase the coordinates of the passed event to frame-relative coordinates.

**Parameters** **event** – The event to be rebased.

**Returns** A new event object appropriately re-based.

**reduce_cpu**
Whether this Frame should try to optimize refreshes to reduce CPU.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**save** (*validate=False*)
Save the current values in all the widgets back to the persistent data storage.

**Parameters** **validate** – Whether to validate the data before saving.

Calling this while setting the *data* field (e.g. in a widget callback) will have no effect.

When validating data, it can throw an Exception for any

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**set_scroll_pos** (*pos*)
Set current position for scroll bar.

**set_theme** (*theme*)
Pick a palette from the list of supported THEMES.

**Parameters** **theme** – The name of the theme to set.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**switch_focus** (*layout*, *column*, *widget*)
Switch focus to the specified widget.

**Parameters**

- **layout** – The layout that owns the widget.
- **column** – The column the widget is in.
- **widget** – The index of the widget to take the focus.

**title**
Title for this Frame.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

## asciimatics.widgets.label module

This mdoule implements a widget to give a text label

**class** asciimatics.widgets.label.**Label** (*label*, *height=1*, *align='<'*, *name=None*)
Bases: *asciimatics.widgets.widget.Widget*

A text label.

**Parameters**

- **label** – The text to be displayed for the Label.
- **height** – Optional height for the label. Defaults to 1 line.
- **align** – Optional alignment for the Label. Defaults to left aligned. Options are "<" = left, ">" = right and "^" = centre

- **name** – The name of this widget.

**blur**()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**focus**()
Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height** (*offset*, *width*)

Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset** ()

The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout** (*x*, *y*, *offset*, *w*, *h*)

Set the size and position of the Widget.

This should not be called directly. It is used by the *Layout* class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**text**

The current text for this Label.

**update** (*frame_no*)

The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**

The current value for this Label.

**width**

The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.layout module

This module implements the displaying of widgets appropriately

**class** asciimatics.widgets.layout.**Layout** (*columns*, *fill_frame=False*)

Bases: object

Widget layout handler.

All Widgets must be contained within a Layout within a Frame.The Layout class is responsible for deciding the exact size and location of the widgets. The logic uses similar ideas as used in modern web frameworks and is as follows.

1. The Frame owns one or more Layouts. The Layouts stack one above each other when displayed - i.e. the first Layout in the Frame is above the second, etc.
2. Each Layout defines the horizontal constraints by defining columns as a percentage of the full canvas width.

3. The Widgets are assigned a column within the Layout that owns them.
4. The Layout then decides the exact size and location to make the Widget best fit the canvas as constrained by the above.

   **Parameters**

   - **`columns`** – A list of numbers specifying the width of each column in this layout.
   - **`fill_frame`** – Whether this Layout should attempt to fill the rest of the Frame. Defaults to False.

The Layout will automatically normalize the units used for the columns, e.g. converting [2, 6, 2] to [20%, 60%, 20%] of the available canvas.

**`add_widget`**(*widget*, *column=0*)

Add a widget to this Layout.

If you are adding this Widget to the Layout dynamically after starting to play the Scene, don't forget to ensure that the value is explicitly set before the next update.

**Parameters**

- **`widget`** – The widget to be added.
- **`column`** – The column within the widget for this widget. Defaults to zero.

**Returns** The passed in widget (so you can store a reference if needed).

**`blur`**()

Call this to take the input focus from this Layout.

**`clear_widgets`**()

Clear all widgets from this Layout.

This method allows users of the Layout to dynamically recreate a new Layout. After calling this method, you can add new widgetsback into the Layout and then need to call *fix* to force the Frame to recalculate the resulting new overall layout.

**`disable`**(*columns=None*)

Disable all widgets in the specified columns of this Layout.

**Parameters** **`columns`** – The list of columns to disable. Defaults to all columns.

**`enable`**(*columns=None*)

Enable all widgets in the specified columns of this Layout.

**Parameters** **`columns`** – The list of columns to enable. Defaults to all columns.

**`fill_frame`**

Whether this Layout is variable height or not.

**`find_widget`**(*name*)

Look for a widget with a specified name.

**Parameters** **`name`** – The name to search for.

**Returns** The widget that matches or None if one couldn't be found.

**`fix`**(*start_x*, *start_y*, *max_width*, *max_height*)

Fix the location and size of all the Widgets in this Layout.

**Parameters**

- **`start_x`** – The start column for the Layout.

- **start_y** – The start line for the Layout.
- **max_width** – Max width to allow this layout.
- **max_height** – Max height to allow this layout.

**Returns** The next line to be used for any further Layouts.

**focus** (*force_first=False*, *force_last=False*, *force_column=None*, *force_widget=None*)
Call this to give this Layout the input focus.

**Parameters**

- **force_first** – Optional parameter to force focus to first widget.
- **force_last** – Optional parameter to force focus to last widget.
- **force_column** – Optional parameter to mandate the new column index.
- **force_widget** – Optional parameter to mandate the new widget index.

The force_column and force_widget parameters must both be set together or they will otherwise be ignored.

**Raises IndexError** – if a force option specifies a bad column or widget, or if the whole Layout is readonly.

**frame_update_count**
The number of frames before this Layout should be updated.

**get_current_widget** ()
Return the current widget with the focus, or None if there isn't one.

**get_nearest_widget** (*target_widget*, *direction*)
Find the nearest enabled widget to the specified target widget, bearing in mind direction of travel.

Direction of travel is defined to be the movement from current Layout to next. This is important for the case where we wrap back to the beginning or end of the Layouts - and so should still only look for the widgets nearest the top/bottom (depending on direction of travel).

This function may return None if there is no match (e.g. all widgets are disabled).

**Parameters**

- **target_widget** – the target widget to match.
- **direction** – The direction of travel across Layouts.

**process_event** (*event*, *hover_focus*)
Process any input event.

**Parameters**

- **event** – The event that was triggered.
- **hover_focus** – Whether to trigger focus change on mouse moves.

**Returns** None if the Effect processed the event, else the original event.

**register_frame** (*frame*)
Register the Frame that owns this Widget.

**Parameters frame** – The owning Frame.

**reset** ()
Reset this Layout and the Widgets it contains.

**save**(*validate*)
Save the current values in all the widgets back to the persistent data storage.

**Parameters** **validate** – whether to validate the saved data or not.

**Raises** InvalidFields if any invalid data is found.

**update**(*frame_no*)
Redraw the widgets inside this Layout.

**Parameters** **frame_no** – The current frame to be drawn.

**update_widgets**(*new_frame=None*)
Reset the values for any Widgets in this Layout based on the current Frame data store.

**Parameters** **new_frame** – optional old Frame - used when cloning scenes.

### asciimatics.widgets.listbox module

This module implements the listbox widget

**class** `asciimatics.widgets.listbox.`**ListBox**(*height*, *options*, *centre=False*, *label=None*, *name=None*, *add_scroll_bar=False*, *parser=None*, *on_change=None*, *on_select=None*, *validator=None*)
Bases: `asciimatics.widgets.baselistbox._BaseListBox`

A ListBox is a widget that displays a list from which the user can select one option.

**Parameters**

- **height** – The required number of input lines for this ListBox.
- **options** – The options for each row in the widget.
- **centre** – Whether to centre the selected line in the list.
- **label** – An optional label for the widget.
- **name** – The name for the ListBox.
- **parser** – Optional parser to colour text.
- **on_change** – Optional function to call when selection changes.
- **on_select** – Optional function to call when the user actually selects an entry from
- **validator** – Optional function to validate selection for this widget.

The *options* are a list of tuples, where the first value is the string to be displayed to the user and the second is an interval value to identify the entry to the program. For example:

options=[(“First option”, 1), (“Second option”, 2)]

**blur**()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**focus**()
Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**options**
The list of options available for user selection

This is a list of tuples (<human readable string>, <internal value>).

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.

- **width** – The total width of the widget, including labels.

**reset**()

The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)

Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**start_line**

The line that will be drawn at the top of the visible section of this list.

**update**(*frame_no*)

The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**

The current value for this list box.

**width**

The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

## asciimatics.widgets.multicolumnlistbox module

This module implements the widget for a multiple column list box

**class** asciimatics.widgets.multicolumnlistbox.**MultiColumnListBox**(*height*, *columns*, *options*, *titles=None*, *label=None*, *name=None*, *add_scroll_bar=False*, *parser=None*, *on_change=None*, *on_select=None*, *space_delimiter=' '*)

Bases: `asciimatics.widgets.baselistbox._BaseListBox`

A MultiColumnListBox is a widget for displaying tabular data.

It displays a list of related data in columns, from which the user can select a line.

**Parameters**

- **height** – The required number of input lines for this ListBox.

- **columns** – A list of widths and alignments for each column.
- **options** – The options for each row in the widget.
- **titles** – Optional list of titles for each column. Must match the length of *columns*.
- **label** – An optional label for the widget.
- **name** – The name for the ListBox.
- **add_scroll_bar** – Whether to add optional scrollbar for large lists.
- **parser** – Optional parser to colour text.
- **on_change** – Optional function to call when selection changes.
- **on_select** – Optional function to call when the user actually selects an entry from
- **space_delimiter** – Optional parameter to define the delimiter between columns. The default value is blank space.

The *columns* parameter is a list of integers or strings. If it is an integer, this is the absolute width of the column in characters. If it is a string, it must be of the format "[<align>]<width>[%]" where:

- <align> is the alignment string ("<" = left, ">" = right, "^" = centre)
- <width> is the width in characters
- % is an optional qualifier that says the number is a percentage of the width of the widget.

Column widths need to encompass any space required between columns, so for example, if your column is 5 characters, allow 6 for an extra space at the end. It is not possible to do this when you have a right-justified column next to a left-justified column, so this widget will automatically space them for you.

An integer value of 0 is interpreted to be use whatever space is left available after the rest of the columns have been calculated. There must be only one of these columns.

The number of columns is for this widget is determined from the number of entries in the *columns* parameter. The *options* list is then a list of tuples of the form ([val1, val2, ... , valn], index). For example, this data provides 2 rows for a 3 column widget:

> options=[(["One", "row", "here"], 1), (["Second", "row", "here"], 2)]

The options list may be None and then can be set later using the *options* property on this widget.

**blur**()
: Call this to take the input focus from this Widget.

**custom_colour**
: A custom colour to use instead of the normal calculated one when drawing this widget.

  This must be a key name from the palette dictionary.

**disabled**
: Whether this widget is disabled or not.

**focus**()
: Call this to give this Widget the input focus.

**frame**
: The Frame that contains this Widget.

**frame_update_count**
: The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**options**
The list of options available for user selection

This is a list of tuples ([<col 1 string>, . . . , <col n string>], <internal value>).

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **`x`** – The x position of the widget.
- **`y`** – The y position of the widget.
- **`offset`** – The allowed label size for the widget.
- **`w`** – The width of the widget.
- **`h`** – The height of the widget.

**`start_line`**
The line that will be drawn at the top of the visible section of this list.

**`update`** (*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **`frame_no`** – The frame number for this screen update.

**`value`**
The current value for this list box.

**`width`**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

## asciimatics.widgets.popupdialog module

This module implements a Pop up dialog message box

**class** `asciimatics.widgets.popupdialog.`**`PopUpDialog`** (*screen*, *text*, *buttons*, *on_close=None*, *has_shadow=False*, *theme='warning'*)

Bases: *`asciimatics.widgets.frame.Frame`*

A fixed implementation Frame that provides a standard message box dialog.

**Parameters**

- **`screen`** – The Screen that owns this dialog.
- **`text`** – The message text to display.
- **`buttons`** – A list of button names to display. This may be an empty list.
- **`on_close`** – Optional function to invoke on exit.
- **`has_shadow`** – optional flag to specify if dialog should have a shadow when drawn.
- **`theme`** – optional colour theme for this pop-up. Defaults to the warning colours.

The *on_close* method (if specified) will be called with one integer parameter that corresponds to the index of the button passed in the array of available *buttons*.

Note that *on_close* must be a static method to work across screen resizing. Either it is static (and so the dialog will be cloned) or it is not (and the dialog will disappear when the screen is resized).

**`add_effect`** (*effect*)
Add an Effect to the Frame.

**Parameters** **`effect`** – The Effect to be added.

**add_layout**(*layout*)
Add a Layout to the Frame.

**Parameters** **layout** – The Layout to be added.

**border_box**
Instance of *BoxTool* that specifies the characters used to draw the border to this frame. You can change the border character style by modifying the `style` property on this object. Allowed styles are defined in *constants*.

**canvas**
The Canvas that backs this Frame.

**clone**(*screen*, *scene*)
Create a clone of this Dialog into a new Screen.

**Parameters**

- **screen** – The new Screen object to clone into.
- **scene** – The new Scene object to clone into.

**data**
Data dictionary containing values from the contained widgets.

**delete_count**
The number of frames before this Effect should be deleted.

**find_widget**(*name*)
Look for a widget with a specified name.

**Parameters** **name** – The name to search for.

**Returns** The widget that matches or None if one couldn't be found.

**fix**()
Fix the layouts and calculate the locations of all the widgets.

This function should be called once all Layouts have been added to the Frame and all widgets added to the Layouts.

**focussed_widget**
The widget that currently has the focus within this Frame.

**frame_update_count**
The number of frames before this Effect should be updated.

**get_scroll_pos**()
Get current position for scroll bar.

**move_to**(*x*, *y*, *h*)
Make the specified location visible. This is typically used by a widget to scroll the canvas such that it is visible.

**Parameters**

- **x** – The x location to make visible.
- **y** – The y location to make visible.
- **h** – The height of the location to make visible.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**rebase_event**(*event*)
Rebase the coordinates of the passed event to frame-relative coordinates.

**Parameters** **event** – The event to be rebased.

**Returns** A new event object appropriately re-based.

**reduce_cpu**
Whether this Frame should try to optimize refreshes to reduce CPU.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**save**(*validate=False*)
Save the current values in all the widgets back to the persistent data storage.

**Parameters** **validate** – Whether to validate the data before saving.

Calling this while setting the *data* field (e.g. in a widget callback) will have no effect.

When validating data, it can throw an Exception for any

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**set_scroll_pos**(*pos*)
Set current position for scroll bar.

**set_theme**(*theme*)
Pick a palette from the list of supported THEMES.

**Parameters** **theme** – The name of the theme to set.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**switch_focus**(*layout*, *column*, *widget*)
Switch focus to the specified widget.

**Parameters**

- **layout** – The layout that owns the widget.
- **column** – The column the widget is in.
- **widget** – The index of the widget to take the focus.

**title**
Title for this Frame.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

## asciimatics.widgets.popupmenu module

This module implements a pop up menu widget

**class** asciimatics.widgets.popupmenu.**PopupMenu** (*screen*, *menu_items*, *x*, *y*)
Bases: *asciimatics.widgets.frame.Frame*

A widget for displaying a menu.

**Parameters**

- **screen** – The Screen being used for this pop-up.
- **menu_items** – a list of items to be displayed in the menu.
- **x** – The X coordinate for the desired pop-up.
- **y** – The Y coordinate for the desired pop-up.

The menu_items parameter is a list of 2-tuples, which define the text to be displayed in the menu and the function to call when that menu item is clicked. For example:

menu_items = [("Open", file_open), ("Save", file_save), ("Close", file_close)]

**add_effect** (*effect*)
Add an Effect to the Frame.

**Parameters** **effect** – The Effect to be added.

**add_layout** (*layout*)
Add a Layout to the Frame.

**Parameters** **layout** – The Layout to be added.

**border_box**
Instance of *BoxTool* that specifies the characters used to draw the border to this frame. You can change the border character style by modifying the `style` property on this object. Allowed styles are defined in *constants*.

**canvas**
The Canvas that backs this Frame.

**clone** (*_*, *scene*)
Create a clone of this Frame into a new Screen.

**Parameters**

- **_** – ignored.
- **scene** – The new Scene object to clone into.

**data**
Data dictionary containing values from the contained widgets.

**delete_count**
The number of frames before this Effect should be deleted.

**find_widget** (*name*)
Look for a widget with a specified name.

**Parameters** `name` – The name to search for.

**Returns** The widget that matches or None if one couldn't be found.

**`fix`**()
Fix the layouts and calculate the locations of all the widgets.

This function should be called once all Layouts have been added to the Frame and all widgets added to the Layouts.

**`focussed_widget`**
The widget that currently has the focus within this Frame.

**`frame_update_count`**
The number of frames before this Effect should be updated.

**`get_scroll_pos`**()
Get current position for scroll bar.

**`move_to`**(*x*, *y*, *h*)
Make the specified location visible. This is typically used by a widget to scroll the canvas such that it is visible.

**Parameters**

- `x` – The x location to make visible.
- `y` – The y location to make visible.
- `h` – The height of the location to make visible.

**`process_event`**(*event*)
Process any input event.

**Parameters** `event` – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`rebase_event`**(*event*)
Rebase the coordinates of the passed event to frame-relative coordinates.

**Parameters** `event` – The event to be rebased.

**Returns** A new event object appropriately re-based.

**`reduce_cpu`**
Whether this Frame should try to optimize refreshes to reduce CPU.

**`register_scene`**(*scene*)
Register the Scene that owns this Effect.

**Parameters** `scene` – The Scene to be registered

**`reset`**()
Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`save`**(*validate=False*)
Save the current values in all the widgets back to the persistent data storage.

**Parameters** `validate` – Whether to validate the data before saving.

Calling this while setting the *data* field (e.g. in a widget callback) will have no effect.

When validating data, it can throw an Exception for any

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**set_scroll_pos** (*pos*)
Set current position for scroll bar.

**set_theme** (*theme*)
Pick a palette from the list of supported THEMES.

**Parameters** **theme** – The name of the theme to set.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**switch_focus** (*layout*, *column*, *widget*)
Switch focus to the specified widget.

**Parameters**

- **layout** – The layout that owns the widget.
- **column** – The column the widget is in.
- **widget** – The index of the widget to take the focus.

**title**
Title for this Frame.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

### asciimatics.widgets.radiobuttons module

This module implements the widget for radio buttons

**class** asciimatics.widgets.radiobuttons.**RadioButtons** (*options*, *label=None*, *name=None*, *on_change=None*, ***kwargs*)

Bases: `asciimatics.widgets.widget.Widget`

A RadioButtons widget is used to ask for one of a list of values to be selected by the user.

It consists of an optional label and then a list of selection bullets with field names.

**Parameters**

- **options** – A list of (text, value) tuples for each radio button.
- **label** – An optional label for the widget.
- **name** – The internal name for the widget.
- **on_change** – Optional function to call when text changes.

Also see the common keyword arguments in `Widget`.

**`blur`**()
Call this to take the input focus from this Widget.

**`custom_colour`**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**`disabled`**
Whether this widget is disabled or not.

**`focus`**()
Call this to give this Widget the input focus.

**`frame`**
The Frame that contains this Widget.

**`frame_update_count`**
The number of frames before this Widget should be updated.

**`get_location`**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**`is_mouse_over`**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **`event`** – The MouseEvent to check.
- **`include_label`** – Include space reserved for the label when checking.
- **`width_modifier`** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**`is_tab_stop`**
Whether this widget is a valid tab stop for keyboard navigation.

**`is_valid`**
Whether this widget has passed its data validation or not.

**`is_visible`**
Whether this widget is visible on the Canvas or not.

**`label`**
The label for this widget. Can be *None*.

**`name`**
The name for this widget (for reference in the persistent data). Can be *None*.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_frame`**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **`frame`** – The owning Frame.

**required_height** (*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset** ()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout** (*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the *Layout* class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update** (*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**
The current value for these RadioButtons.

**width**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.scrollbar module

This module implements a scroll bar capability for widgets

### asciimatics.widgets.temppopup module

This module implements a base class for popups

### asciimatics.widgets.text module

This widget implements a text based input field

**class** asciimatics.widgets.text.**Text** (*label=None*, *name=None*, *on_change=None*, *validator=None*, *hide_char=None*, *max_length=None*, *readonly=False*, ***kwargs*)
Bases: *asciimatics.widgets.widget.Widget*

A Text widget is a single line input field.

It consists of an optional label and an entry box.

**Parameters**

- **label** – An optional label for the widget.
- **name** – The name for the widget.
- **on_change** – Optional function to call when text changes.
- **validator** – Optional definition of valid data for this widget. This can be a function (which takes the current value and returns True for valid content) or a regex string (which must match the entire allowed value).
- **hide_char** – Character to use instead of what the user types - e.g. to hide passwords.
- **max_length** – Optional maximum length of the field. If set, the widget will limit data entry to this length.
- **readonly** – Whether the widget prevents user input to change values. Default is False.

Also see the common keyword arguments in *Widget*.

**blur**()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**focus**()
Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event** (*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**readonly**
Whether this widget is readonly or not.

**register_frame** (*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height** (*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset** ()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout** (*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update** (*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**
The current value for this Text.

**width**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.textbox module

This module implements a multi line editing text box

**class** `asciimatics.widgets.textbox.`**`TextBox`**(*height*, *label=None*, *name=None*, *as_string=False*, *line_wrap=False*, *parser=None*, *on_change=None*, *readonly=False*, ***kwargs*)

Bases: `asciimatics.widgets.widget.Widget`

A TextBox is a widget for multi-line text editing.

It consists of a framed box with option label.

**Parameters**

- **`height`** – The required number of input lines for this TextBox.
- **`label`** – An optional label for the widget.
- **`name`** – The name for the TextBox.
- **`as_string`** – Use string with newline separator instead of a list for the value of this widget.
- **`line_wrap`** – Whether to wrap at the end of the line.
- **`parser`** – Optional parser to colour text.
- **`on_change`** – Optional function to call when text changes.
- **`readonly`** – Whether the widget prevents user input to change values. Default is False.

Also see the common keyword arguments in `Widget`.

**`auto_scroll`**

When set to True the TextBox will scroll to the bottom when created or next text is added. When set to False, the current scroll position will remain even if the contents are changed.

Defaults to True.

**`blur`**()

Call this to take the input focus from this Widget.

**`custom_colour`**

A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**`disabled`**

Whether this widget is disabled or not.

**`focus`**()

Call this to give this Widget the input focus.

**`frame`**

The Frame that contains this Widget.

**`frame_update_count`**

The number of frames before this Widget should be updated.

**`get_location`**()

Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**hide_cursor**
Set to True to stop the cursor from showing. Defaults to False.

**is_mouse_over** (*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event** (*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**readonly**
Whether this widget is readonly or not.

**register_frame** (*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height** (*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset** ()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout** (*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.

- **`y`** – The y position of the widget.
- **`offset`** – The allowed label size for the widget.
- **`w`** – The width of the widget.
- **`h`** – The height of the widget.

**`update`** (*frame_no*)

The update method is called whenever this widget needs to redraw itself.

**Parameters** **`frame_no`** – The frame number for this screen update.

**`value`**

The current value for this TextBox.

**`width`**

The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

## asciimatics.widgets.timepicker module

This module implements a time picker widget

**class** `asciimatics.widgets.timepicker.`**`TimePicker`**(*label=None*, *name=None*, *seconds=False*, *on_change=None*, ***kwargs*)

Bases: `asciimatics.widgets.widget.Widget`

A TimePicker widget allows you to pick a time from a compact, temporary, pop-up Frame.

**Parameters**

- **`label`** – An optional label for the widget.
- **`name`** – The name for the widget.
- **`seconds`** – Whether to include selection of seconds or not.
- **`on_change`** – Optional function to call when the selected time changes.

Also see the common keyword arguments in `Widget`.

**`blur`**()

Call this to take the input focus from this Widget.

**`custom_colour`**

A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**`disabled`**

Whether this widget is disabled or not.

**`focus`**()

Call this to give this Widget the input focus.

**`frame`**

The Frame that contains this Widget.

**`frame_update_count`**

The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.

- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update** (*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**
The current selected time.

**width**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

## asciimatics.widgets.utilities module

This module defines commonly used pieces for widgets

asciimatics.widgets.utilities.**THEMES = {'bright': {'control': (3, 1, 0), 'disabled': (0,**
Standard palettes for use with `set_theme()`. Each entry in THEMES contains a colour palette for use by the widgets within a Frame. Each colour palette is a dictionary mapping a colour key to a 3-tuple of (foreground colour, attribute, background colour). The "default" theme defines all the required keys for a palette.

## asciimatics.widgets.verticaldivider module

This module implements a vertical division between widgets

**class** asciimatics.widgets.verticaldivider.**VerticalDivider** (*height=-135792467*)
Bases: *asciimatics.widgets.widget.Widget*

A vertical divider for separating columns.

This widget should be put into a column of its own in the Layout.

**Parameters** **height** – The required height for this divider.

**blur** ()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**focus** ()
Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout**(*x*, *y*, *offset*, *w*, *h*)
Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.

- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update** (*frame_no*)
The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**
The current value for this VerticalDivider.

**width**
The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

### asciimatics.widgets.widget module

This module allows you to create interactive text user interfaces. For more details see http://asciimatics.readthedocs.io/en/latest/widgets.html

**class** asciimatics.widgets.widget.**Widget** (*name*, *tab_stop=True*, *disabled=False*, *on_focus=None*, *on_blur=None*)
Bases: object

A Widget is a re-usable component that can be used to create a simple GUI.

**Parameters**

- **name** – The name of this Widget.
- **tab_stop** – Whether this widget should take focus or not when tabbing around the Frame.
- **disabled** – Whether this Widget should be disabled or not.
- **on_focus** – Optional callback whenever this widget gets the focus.
- **on_blur** – Optional callback whenever this widget loses the focus.

**FILL_COLUMN = -135792467**
Widgets with this constant for the required height will be re-sized to fit the maximum space used by any other column in the Layout.

**FILL_FRAME = -135792468**
Widgets with this constant for the required height will be re-sized to fit the available vertical space in the Layout.

**blur** ()
Call this to take the input focus from this Widget.

**custom_colour**
A custom colour to use instead of the normal calculated one when drawing this widget.

This must be a key name from the palette dictionary.

**disabled**
Whether this widget is disabled or not.

**focus** ()
Call this to give this Widget the input focus.

**frame**
The Frame that contains this Widget.

**frame_update_count**
The number of frames before this Widget should be updated.

**get_location**()
Return the absolute location of this widget on the Screen, taking into account the current state of the Frame that is displaying it and any label offsets of the Widget.

**Returns** A tuple of the form (<X coordinate>, <Y coordinate>).

**is_mouse_over**(*event*, *include_label=True*, *width_modifier=0*)
Check if the specified mouse event is over this widget.

**Parameters**

- **event** – The MouseEvent to check.
- **include_label** – Include space reserved for the label when checking.
- **width_modifier** – Adjustment to width (e.g. for scroll bars).

**Returns** True if the mouse is over the active parts of the widget.

**is_tab_stop**
Whether this widget is a valid tab stop for keyboard navigation.

**is_valid**
Whether this widget has passed its data validation or not.

**is_visible**
Whether this widget is visible on the Canvas or not.

**label**
The label for this widget. Can be *None*.

**name**
The name for this widget (for reference in the persistent data). Can be *None*.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_frame**(*frame*)
Register the Frame that owns this Widget.

**Parameters** **frame** – The owning Frame.

**required_height**(*offset*, *width*)
Calculate the minimum required height for this widget.

**Parameters**

- **offset** – The allowed width for any labels.
- **width** – The total width of the widget, including labels.

**reset**()
The reset method is called whenever the widget needs to go back to its default (initially created) state.

**set_layout** (*x*, *y*, *offset*, *w*, *h*)

Set the size and position of the Widget.

This should not be called directly. It is used by the `Layout` class to arrange all widgets within the Frame.

**Parameters**

- **x** – The x position of the widget.
- **y** – The y position of the widget.
- **offset** – The allowed label size for the widget.
- **w** – The width of the widget.
- **h** – The height of the widget.

**update** (*frame_no*)

The update method is called whenever this widget needs to redraw itself.

**Parameters** **frame_no** – The frame number for this screen update.

**value**

The value to return for this widget based on the user's input.

**width**

The width of this Widget (excluding any labels).

Only valid after the Frame has been fixed in place.

#### Module contents

This is the module initialization for widgets

### 8.1.2 Submodules

### 8.1.3 asciimatics.constants module

This module is just a collection of simple helper functions.

asciimatics.constants.**COLOUR_REGEX = '^\\$\\{((\\d+),(\\d+),(\\d+)|(\\d+),(\\d+)|(\\d+))\**

Regex for asciimatics ${c,a,b} embedded colour attributes.

asciimatics.constants.**MAPPING_ATTRIBUTES = {'1': 1, '2': 2, '3': 3, '4': 4}**

Attribute conversion table for the ${c,a} form of attributes for `paint`.

### 8.1.4 asciimatics.effects module

This module defines *Effects* which can be used for animations. For more details see http://asciimatics.readthedocs.io/en/latest/animation.html

**class** asciimatics.effects.**Background** (*screen*, *bg=0*, ***kwargs*)

Bases: `asciimatics.effects.Effect`

Effect to be used as a Desktop background. This sets the background to the specified colour.

**Parameters**

- **screen** – The Screen being used for the Scene.

- **bg** – Optional colour for the background.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**BannerText**(*screen*, *renderer*, *y*, *colour*, *bg=0*, ***kwargs*)
Bases: *asciimatics.effects.Effect*

Special effect to scroll some text (from a Renderer) horizontally like a banner.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **renderer** – The renderer to be scrolled
- **y** – The line (y coordinate) for the start of the text.
- **colour** – The default foreground colour to use for the text.
- **bg** – The default background colour to use for the text.

Also see the common keyword arguments in `Effect`.

**`delete_count`**
The number of frames before this Effect should be deleted.

**`frame_update_count`**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_scene`**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **`scene`** – The Scene to be registered

**`reset`**()
Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`scene`**
The Scene that owns this Effect.

**`screen`**
The Screen that will render this Effect.

**`stop_frame`**
Last frame for this effect. A value of zero means no specific end.

**`update`**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **`frame_no`** – The index of the frame being generated.

**class** `asciimatics.effects.`**`Clock`**(*screen*, *x*, *y*, *r*, *bg=0*, ***kwargs*)
Bases: `asciimatics.effects.Effect`

An ASCII ticking clock (telling the correct local time).

**Parameters**

- **`screen`** – The Screen being used for the Scene.
- **`x`** – X coordinate for the centre of the clock.
- **`y`** – Y coordinate for the centre of the clock.
- **`r`** – Radius of the clock.
- **`bg`** – Background colour for the clock.

Also see the common keyword arguments in `Effect`.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**Cog**(*screen*, *x*, *y*, *radius*, *direction=1*, *colour=7*, ***kwargs*)
Bases: *asciimatics.effects.Effect*

A rotating cog.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **x** – X coordinate of the centre of the cog.
- **y** – Y coordinate of the centre of the cog.
- **radius** – The radius of the cog.
- **direction** – The direction of rotation. Positive numbers are anti-clockwise, negative numbers clockwise.
- **colour** – The colour of the cog.

Also see the common keyword arguments in `Effect`.

**`delete_count`**
The number of frames before this Effect should be deleted.

**`frame_update_count`**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_scene`**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **`scene`** – The Scene to be registered

**`reset`**()
Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`scene`**
The Scene that owns this Effect.

**`screen`**
The Screen that will render this Effect.

**`stop_frame`**
Last frame for this effect. A value of zero means no specific end.

**`update`**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **`frame_no`** – The index of the frame being generated.

**class** `asciimatics.effects.`**`Cycle`**(*screen*, *renderer*, *y*, ***kwargs*)
Bases: `asciimatics.effects.Effect`

Special effect to cycle the colours on some specified text from a Renderer. The text is automatically centred to the width of the Screen. This effect is not compatible with multi-colour rendered text.

**Parameters**

- **`screen`** – The Screen being used for the Scene.
- **`renderer`** – The Renderer which is to be cycled.
- **`y`** – The line (y coordinate) for the start of the text.

Also see the common keyword arguments in `Effect`.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event** (*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene** (*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset** ()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**Effect** (*screen*, *start_frame=0*, *stop_frame=0*, *delete_count=None*)
Bases: `object`

Abstract class to handle a special effect on the screen. An Effect can cover anything from a static image at the start of the Scene through to dynamic animations that need to be redrawn for every frame.

The basic interaction with a `Scene` is as follows:

1. The Scene will register with the Effect when it as added using `register_scene()`.
2. The Scene will call `Effect.reset()` for all Effects when it starts.
3. The Scene will determine the number of frames required (either through explicit configuration or querying `stop_frame` for every Effect).
4. It will then run the scene, calling `Effect.update()` for each effect that is in the scene. The base Effect will then call the abstract method _update() if the effect should be visible.

5. If any keys are pressed or the mouse moved/clicked, the scene will call `Effect.process_event()` for each event, allowing the effect to act on it if needed.

New Effects, therefore need to implement the abstract methods on this class to satisfy the contract with Scene. Since most effects don't require user interaction, the default process_event() implementation will ignore the event (and so effects don't need to implement this method unless needed).

**Parameters**

- **`screen`** – The Screen that will render this Effect.
- **`start_frame`** – Start index for the effect.
- **`stop_frame`** – Stop index for the effect.
- **`delete_count`** – Number of frames before this effect is deleted.

**`delete_count`**
The number of frames before this Effect should be deleted.

**`frame_update_count`**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_scene`**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **`scene`** – The Scene to be registered

**`reset`**()
Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`scene`**
The Scene that owns this Effect.

**`screen`**
The Screen that will render this Effect.

**`stop_frame`**
Last frame for this effect. A value of zero means no specific end.

**`update`**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **`frame_no`** – The index of the frame being generated.

**class** asciimatics.effects.**Julia**(*screen*, *c=None*, ***kwargs*)
Bases: *asciimatics.effects.Effect*

Julia Set generator. See http://en.wikipedia.org/wiki/Julia_set for more information on this fractal.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **c** – The starting value of 'c' for the Julia Set.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**Matrix**(*screen*, ***kwargs*)
Bases: *asciimatics.effects.Effect*

Matrix-like falling green letters.

**Parameters** **screen** – The Screen being used for the Scene.

Also see the common keyword arguments in `Effect`.

**`delete_count`**
The number of frames before this Effect should be deleted.

**`frame_update_count`**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_scene`**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **`scene`** – The Scene to be registered

**`reset`**()
Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`scene`**
The Scene that owns this Effect.

**`screen`**
The Screen that will render this Effect.

**`stop_frame`**
Last frame for this effect. A value of zero means no specific end.

**`update`**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **`frame_no`** – The index of the frame being generated.

**class** `asciimatics.effects.`**`Mirage`**(*screen*, *renderer*, *y*, *colour*, ***kwargs*)
Bases: `asciimatics.effects.Effect`

Special effect to make bits of the specified text appear over time. This text is automatically centred on the screen.

**Parameters**

- **`screen`** – The Screen being used for the Scene.
- **`renderer`** – The renderer to be displayed.
- **`y`** – The line (y coordinate) for the start of the text.
- **`colour`** – The colour attribute to use for the text.

Also see the common keyword arguments in `Effect`.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**Print**(*screen*, *renderer*, *y*, *x=None*, *colour=7*, *attr=0*, *bg=0*, *clear=False*, *transparent=True*, *speed=4*, ***kwargs*)
Bases: *asciimatics.effects.Effect*

Special effect that simply prints the specified text (from a Renderer) at the required location.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **renderer** – The renderer to be printed.
- **x** – The column (x coordinate) for the start of the text. If not specified, defaults to centring the text on screen.
- **y** – The line (y coordinate) for the start of the text.
- **colour** – The foreground colour to use for the text.
- **attr** – The colour attribute to use for the text.

- **bg** – The background colour to use for the text.
- **clear** – Whether to clear the text before stopping.
- **transparent** – Whether to print spaces (and so be able to overlay other Effects). If False, this will redraw all characters and so replace any Effect underneath it.
- **speed** – The refresh rate in frames between refreshes.

Note that a speed of 1 will force the Screen to redraw the Effect every frame update, while a value of 0 will redraw on demand - i.e. will redraw every time that an update is required by another Effect.

Also see the common keyword arguments in `Effect`.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event** (*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene** (*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset** ()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**RandomNoise** (*screen*, *signal=None*, *jitter=6*, ***kwargs*)
Bases: `asciimatics.effects.Effect`

White noise effect - like an old analogue TV set that isn't quite tuned right. If desired, a signal image (from a renderer) can be specified that will appear from the noise.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **signal** – The renderer to use as the ‘signal’ in the white noise.
- **jitter** – The amount that the signal will jump when there is noise.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**Scroll**(*screen*, *rate*, ***kwargs*)
Bases: *asciimatics.effects.Effect*

Special effect to scroll the screen up at a required rate. Since the Screen has a limited size and will not wrap, ensure that it is large enough to Scroll for the desired time.

**Parameters**

- **screen** – The Screen being used for the Scene.

- **rate** – How many frames to wait between scrolling the screen.

Also see the common keyword arguments in `Effect`.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event** (*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene** (*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset** ()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**Snow** (*screen*, ***kwargs*)
Bases: `asciimatics.effects.Effect`

Settling snow effect.

**Parameters** **screen** – The Screen being used for the Scene.

Also see the common keyword arguments in `Effect`.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.effects.**Sprite**(*screen*, *renderer_dict*, *path*, *colour=7*, *clear=True*, ***kwargs*)

Bases: *asciimatics.effects.Effect*

An animated character capable of following a path around the screen.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **renderer_dict** – A dictionary of Renderers to use for displaying the Sprite.
- **path** – The Path for the Sprite to follow.
- **colour** – The colour to use to render the Sprite.
- **clear** – Whether to clear out old images or leave a trail.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**`last_position`**()

Returns the last position of this Sprite as a tuple (x, y, width, height).

**`overlaps`**(*other*, *use_new_pos=False*)

Check whether this Sprite overlaps another.

**Parameters**

- **`other`** – The other Sprite to check for an overlap.
- **`use_new_pos`** – Whether to use latest position (due to recent update). Defaults to False.

**Returns** True if the two Sprites overlap.

**`process_event`**(*event*)

Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_scene`**(*scene*)

Register the Scene that owns this Effect.

**Parameters** **`scene`** – The Scene to be registered

**`reset`**()

Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**

Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`scene`**

The Scene that owns this Effect.

**`screen`**

The Screen that will render this Effect.

**`stop_frame`**

Last frame for this effect. A value of zero means no specific end.

**`update`**(*frame_no*)

Process the animation effect for the specified frame number.

**Parameters** **`frame_no`** – The index of the frame being generated.

**class** `asciimatics.effects.`**`Stars`**(*screen*, *count*, *pattern='..+.. ...x... ...*... '*, ***kwargs*)

Bases: `asciimatics.effects.Effect`

Add random stars to the screen and make them twinkle.

**Parameters**

- **`screen`** – The Screen being used for the Scene.
- **`count`** – The number of starts to create.
- **`pattern`** – The string pattern for the stars to loop through

Also see the common keyword arguments in `Effect`.

**`delete_count`**
The number of frames before this Effect should be deleted.

**`frame_update_count`**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_scene`**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **`scene`** – The Scene to be registered

**`reset`**()
Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`scene`**
The Scene that owns this Effect.

**`screen`**
The Screen that will render this Effect.

**`stop_frame`**
Last frame for this effect. A value of zero means no specific end.

**`update`**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **`frame_no`** – The index of the frame being generated.

**class** `asciimatics.effects.`**`Wipe`**(*screen*, *bg=0*, ***kwargs*)
Bases: `asciimatics.effects.Effect`

Wipe the screen down from top to bottom.

**Parameters**

- **`screen`** – The Screen being used for the Scene.
- **`bg`** – Optional background colour to use for the wipe.

Also see the common keyword arguments in `Effect`.

**`delete_count`**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

asciimatics.effects.**random**() → x in the interval [0, 1).

## 8.1.5 asciimatics.event module

This module defines basic input events. For more details, see http://asciimatics.readthedocs.io/en/latest/.html

**class** asciimatics.event.**Event**
Bases: object

A class to hold information about an input event.

The exact contents varies from event to event. See specific classes for more information.

**class** asciimatics.event.**KeyboardEvent**(*key_code*)
Bases: *asciimatics.event.Event*

An event that represents a key press.

Its key field is the *key_code*. This is the ordinal representation of the key (taking into account keyboard state - e.g. caps lock) if possible, or an extended key code (the *KEY_xxx* constants in the *Screen* class) where not.

**Parameters** **`key_code`** – the ordinal value of the key that was pressed.

**class** `asciimatics.event.`**`MouseEvent`**(*x*, *y*, *buttons*)

Bases: *`asciimatics.event.Event`*

An event that represents a mouse move or click.

Allowed values for the buttons are any bitwise combination of *LEFT_CLICK*, *RIGHT_CLICK* and *DOUBLE_CLICK*.

**Parameters**

- **`x`** – The X coordinate of the mouse event.
- **`y`** – The Y coordinate of the mouse event.
- **`buttons`** – A bitwise flag for any mouse buttons that were pressed (if any).

### 8.1.6 asciimatics.exceptions module

This module defines the exceptions used by asciimatics.

**exception** `asciimatics.exceptions.`**`Highlander`**

Bases: `Exception`

There can be only one Layout or Widget with certain options set (designed to fill the rest of the screen). If you hit this exception you have a bug in your application.

If you don't get the name, take a look at this link.

**`with_traceback`**()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

**exception** `asciimatics.exceptions.`**`InvalidFields`**(*fields*)

Bases: `Exception`

When saving data from a Frame, you can ask the Frame to validate the data before saving. This is the exception that gets thrwn if any invalid datd is found.

**Parameters** **`fields`** – The list of the fields that are invalid.

**`fields`**

The list of fields that are invalid.

**`with_traceback`**()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

**exception** `asciimatics.exceptions.`**`NextScene`**(*name=None*)

Bases: `Exception`

Any component can raise this exception to tell Asciimatics to move to the next Scene being played. Only effective inside *Screen.play()*.

**Parameters** **`name`** – Next Scene to invoke. Defaults to next in the list.

**`name`**

The name of the next Scene to invoke.

**`with_traceback`**()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

**exception** `asciimatics.exceptions.`**`ResizeScreenError`**(*message*, *scene=None*)

Bases: `Exception`

Asciimatics raises this Exception if the terminal is resized while playing a Scene (and the Screen has been told not to ignore a resizing event).

**Parameters**

- **message** – Error message for this exception.
- **scene** – Scene that was active at time of resize.

**scene**
The Scene that was running when the Screen resized.

**with_traceback**()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

**exception** asciimatics.exceptions.**StopApplication**(*message*)
Bases: Exception

Any component can raise this exception to tell Asciimatics to stop running. If playing a Scene (i.e. inside *Screen.play()*) the Screen will return to the calling function. When used at any other time, the exception will need to be caught by the application using Asciimatics.

**Parameters** **message** – Error message for this exception.

**with_traceback**()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

### 8.1.7 asciimatics.parsers module

This module provides parsers to create ColouredText objects from embedded control strings.

**class** asciimatics.parsers.**AnsiTerminalParser**
Bases: *asciimatics.parsers.Parser*

Parser to handle ANSI terminal escape codes.

Initialize the parser.

**append**(*text*)
Append more text to the current text being processed.

**Parameters** **text** – raw text to process.

**parse**()
Generator to return coloured text from raw text.

Generally returns a stream of text/color tuple/offset tuples. If there is a colour update with no visible text, the first element of the tuple may be None.

**Returns** a 3-tuple of (start offset in raw text, command to execute, parameters)

**reset**(*text*, *colours*)
Reset the parser to analyze the supplied raw text.

**Parameters**

- **text** – raw text to process.
- **colours** – colour tuple to initialise the colour map.

**class** asciimatics.parsers.**AsciimaticsParser**
Bases: *asciimatics.parsers.Parser*

Parser to handle Asciimatics rendering escape strings.

Initialize the parser.

**append** (*text*)

Append more text to the current text being processed.

**Parameters** **text** – raw text to process.

**parse** ()

Generator to return coloured text from raw text.

**Returns** a 3-tuple of (start offset in raw text, command to execute, parameters)

**reset** (*text*, *colours*)

Reset the parser to analyze the supplied raw text.

**Parameters**

- **text** – raw text to process.
- **colours** – colour tuple to initialise the colour map.

**class** asciimatics.parsers.**ControlCodeParser**

Bases: *asciimatics.parsers.Parser*

Parser to replace all control codes with a readable version - e.g. "^M" for carriage return.

Initialize the parser.

**append** (*text*)

Append more text to the current text being processed.

**Parameters** **text** – raw text to process.

**parse** ()

Generator to return coloured text from raw text.

**Returns** a 3-tuple of (start offset in raw text, command to execute, parameters)

**reset** (*text*, *colours*)

Reset the parser to analyze the supplied raw text.

**Parameters**

- **text** – raw text to process.
- **colours** – colour tuple to initialise the colour map.

**class** asciimatics.parsers.**Parser**

Bases: object

Abstract class to represent text parsers that extract colour control codes from raw text and convert them to displayable text and associated colour maps.

Initialize the parser.

**CHANGE_COLOURS = 1**

Command to change active colour tuple. Parameters are the 3-tuple of (fg, attr, bg)

**CLEAR_SCREEN = 8**

Clear the screen. No parameters.

**DELETE_CHARS = 5**

Command to delete next N characters from this line.

**DELETE_LINE = 4**

Command to delete part of the current line. Params are 0, 1 and 2 for end, start, all.

**DISPLAY_TEXT = 0**
Command to display some text. Parameter is the text to display

**MOVE_ABSOLUTE = 2**
Command to move cursor to abs position. Parameters are (x, y) where each are absolute positions.

**MOVE_RELATIVE = 3**
Command to move cursor to relative position. Parameters are (x, y) where each are relative positions.

**NEXT_TAB = 6**
Next tab stop

**RESTORE_CURSOR = 10**
Restore the cursor position. No parameters.

**SAVE_CURSOR = 9**
Save the cursor position. No parameters.

**SHOW_CURSOR = 7**
Set cursor visibility. Param is boolean setting True=visible

**append**(*text*)
Append more text to the current text being processed.

**Parameters** **text** – raw text to process.

**parse**()
Generator to return coloured text from raw text.

Generally returns a stream of text/color tuple/offset tuples. If there is a colour update with no visible text, the first element of the tuple may be None.

**Returns** a 3-tuple of (start offset in raw text, command to execute, parameters)

**reset**(*text*, *colours*)
Reset the parser to analyze the supplied raw text.

**Parameters**

- **text** – raw text to process.
- **colours** – colour tuple to initialise the colour map.

## 8.1.8 asciimatics.particles module

This module implements a particle system for complex animcation effects. For more details, see http://asciimatics.readthedocs.io/en/latest/animation.html

**class** asciimatics.particles.**DropEmitter**(*screen*, *life_time*)
Bases: *asciimatics.particles.ParticleEmitter*

Replicate the whole screen with Particles and then drop them a cell at a time.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **life_time** – The life time of this particle system.

**update**()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**DropScreen**(*screen*, *life_time*, ***kwargs*)

Bases: *asciimatics.particles.ParticleEffect*

Drop all the text on the screen as if it was subject to gravity.

See *ParticleEffect* for details of the parameters.

**delete_count**

The number of frames before this Effect should be deleted.

**frame_update_count**

The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)

Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)

Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()

Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**

Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**

The Scene that owns this Effect.

**screen**

The Screen that will render this Effect.

**stop_frame**

Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)

Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**Explosion**(*screen*, *x*, *y*, *life_time*, ***kwargs*)

Bases: *asciimatics.particles.ParticleEffect*

An explosion effect.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **x** – The column (x coordinate) for the origin of the effect.
- **y** – The line (y coordinate) for the origin of the effect.

- **life_time** – The life time of the effect.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event** (*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene** (*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset** ()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**ExplosionFlames** (*screen*, *x*, *y*, *life_time*)
Bases: *asciimatics.particles.ParticleEmitter*

An explosion of flame and smoke.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **x** – The column (x coordinate) for the origin of this explosion.
- **y** – The line (y coordinate) for the origin of this explosion.
- **life_time** – The life time of this explosion.

**update**()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**PalmExplosion**(*screen*, *x*, *y*, *life_time*, *on_each=None*)
Bases: *asciimatics.particles.ParticleEmitter*

A classic firework explosion into a palm shape.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **x** – The column (x coordinate) for the origin of this explosion.
- **y** – The line (y coordinate) for the origin of this explosion.
- **life_time** – The life time of this explosion.
- **on_each** – The function to call to spawn a trail.

**update**()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**PalmFirework**(*screen*, *x*, *y*, *life_time*, ***kwargs*)
Bases: *asciimatics.particles.ParticleEffect*

Classic palm shaped firework.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **x** – The column (x coordinate) for the origin of the effect.
- **y** – The line (y coordinate) for the origin of the effect.
- **life_time** – The life time of the effect.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**Particle**(*chars*, *x*, *y*, *dx*, *dy*, *colours*, *life_time*, *move*, *next_colour=None*, *next_char=None*, *parm=None*, *on_create=None*, *on_each=None*, *on_destroy=None*)
Bases: `object`

A single particle in a Particle Effect.

**Parameters**

- **chars** – String of characters to use for the particle.
- **x** – The initial horizontal position of the particle.
- **y** – The initial vertical position of the particle.
- **dx** – The initial horizontal velocity of the particle.
- **dy** – The initial vertical velocity of the particle.
- **colours** – A list of colour tuples to use for the particle.
- **life_time** – The life time of the particle.
- **move** – A function which returns the next location of the particle.
- **next_colour** – An optional function to return the next colour for the particle. Defaults to a linear progression of *chars*.
- **next_char** – An optional function to return the next character for the particle. Defaults to a linear progression of *colours*.
- **parm** – An optional parameter for use within any of the
- **on_create** – An optional function to spawn new particles when this particle first is created.
- **on_each** – An optional function to spawn new particles for every frame of this particle (other than creation/destruction).
- **on_destroy** – An optional function to spawn new particles when this particle is destroyed.

**last**()
The last attributes returned for this particle - typically used for clearing out the particle on the next frame. See *next()* for details of the returned results.

**next**()
The set of attributes for this particle for the next frame to be rendered.

**Returns** A tuple of (character, x, y, fg, attribute, bg)

**class** asciimatics.particles.**ParticleEffect**(*screen*, *x*, *y*, *life_time*, ***kwargs*)
Bases: *asciimatics.effects.Effect*

An Effect that uses a *ParticleEmitter* to create the animation.

To define a new ParticleEffect, you must implement the reset() method to construct a chain of ParticleEmitter objects and append them to the internal _active_systems list.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **x** – The column (x coordinate) for the origin of the effect.
- **y** – The line (y coordinate) for the origin of the effect.
- **life_time** – The life time of the effect.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update** (*frame_no*)

Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**ParticleEmitter** (*screen*, *x*, *y*, *count*, *new_particle*, *spawn*, *life_time*, *blend=False*)

Bases: `object`

An emitter for a particle system to create a set of `_Particle` objects for a *ParticleEffect*. After initialization, the emitter will be called once per frame to be displayed on the Screen.

**Parameters**

- **screen** – The screen to which the particle system will be rendered.
- **x** – The x location of origin of the particle system.
- **y** – The y location of origin of the particle system.
- **count** – The count of new particles to spawn on each frame.
- **new_particle** – The function to call to spawn a new particle.
- **spawn** – The number of frames for which to spawn particles.
- **life_time** – The life time of the whole particle system.
- **blend** – Whether to blend particles or not. A blended system picks the colour based on the number of overlapping particles, while an unblended one picks the colour based on a the state of Each Particle individually as they are drawn. Defaults to False.

**update** ()

The function to draw a new frame for the particle system.

**class** asciimatics.particles.**Rain** (*screen*, *life_time*, ***kwargs*)

Bases: *asciimatics.particles.ParticleEffect*

Rain storm effect.

See *ParticleEffect* for details of the parameters.

**delete_count**

The number of frames before this Effect should be deleted.

**frame_update_count**

The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event** (*event*)

Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene** (*scene*)

Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**RainSource**(*screen*, *life_time*, *on_each*)
Bases: *asciimatics.particles.ParticleEmitter*

Source of the raindrops for a rain storm effect. This emits rain drops from a single line at the top of the screen (starting sufficiently off- screen to ensure that it can cover all the screen due to horizontal motion).

**Parameters**

- **screen** – The Screen being used for this particle system.
- **life_time** – The life time of this particle system.
- **on_each** – Function to call on each iteration of the particle.

**update**()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**RingExplosion**(*screen*, *x*, *y*, *life_time*)
Bases: *asciimatics.particles.ParticleEmitter*

A classic firework explosion in a simple ring.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **x** – The column (x coordinate) for the origin of this explosion.
- **y** – The line (y coordinate) for the origin of this explosion.
- **life_time** – The life time of this explosion.

**update**()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**RingFirework**(*screen*, *x*, *y*, *life_time*, ***kwargs*)
Bases: *asciimatics.particles.ParticleEffect*

Classic rocket with ring explosion.

**Parameters**

- **screen** – The Screen being used for the Scene.

- **x** – The column (x coordinate) for the origin of the effect.
- **y** – The line (y coordinate) for the origin of the effect.
- **life_time** – The life time of the effect.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**Rocket**(*screen*, *x*, *y*, *life_time*, *on_destroy=None*)
Bases: *asciimatics.particles.ParticleEmitter*

A rocket being launched from the ground.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **x** – The column (x coordinate) for the origin of the rocket.
- **y** – The line (y coordinate) for the origin of the rocket.

- **life_time** – The life time of the rocket.
- **on_destroy** – The function to call when the rocket explodes.

**update**()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**SerpentExplosion**(*screen*, *x*, *y*, *life_time*)
Bases: *asciimatics.particles.ParticleEmitter*

A firework explosion where each trail changes direction.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **x** – The column (x coordinate) for the origin of this explosion.
- **y** – The line (y coordinate) for the origin of this explosion.
- **life_time** – The life time of this explosion.

**update**()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**SerpentFirework**(*screen*, *x*, *y*, *life_time*, ***kwargs*)
Bases: *asciimatics.particles.ParticleEffect*

A firework where each trail changes direction.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **x** – The column (x coordinate) for the origin of the effect.
- **y** – The line (y coordinate) for the origin of the effect.
- **life_time** – The life time of the effect.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**ShootScreen**(*screen*, *x*, *y*, *life_time*, *diameter=None*, ***kwargs*)
Bases: *asciimatics.particles.ParticleEffect*

Shoot the screen out like a massive gunshot.

See *ParticleEffect* for details of the parameters.

In addition, it is possible to set the diameter of this effect using the extra keyword parameter.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**ShotEmitter** (*screen*, *x*, *y*, *diameter*, *life_time*)
Bases: *asciimatics.particles.ParticleEmitter*

Replicate the whole screen with Particles and then explode the screen from a given location.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **x** – The x position of the origin of the explosion.
- **y** – The y position of the origin of the explosion.
- **diameter** – The diameter of the explosion.
- **life_time** – The life time of this particle system.

**update** ()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**Splash** (*screen*, *x*, *y*)
Bases: *asciimatics.particles.ParticleEmitter*

Splash effect for falling rain.

**Parameters** **screen** – The Screen being used for this particle system.

**update** ()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**StarExplosion** (*screen*, *x*, *y*, *life_time*, *points*, *on_each*)
Bases: *asciimatics.particles.ParticleEmitter*

A classic firework explosion to a Peony shape with trails.

**Parameters**

- **screen** – The Screen being used for this particle system.
- **x** – The column (x coordinate) for the origin of this explosion.
- **y** – The line (y coordinate) for the origin of this explosion.
- **life_time** – The life time of this explosion.
- **points** – Number of points the explosion should have.
- **on_each** – The function to call to spawn a trail.

**update** ()
The function to draw a new frame for the particle system.

**class** asciimatics.particles.**StarFirework** (*screen*, *x*, *y*, *life_time*, ***kwargs*)
Bases: *asciimatics.particles.ParticleEffect*

Classic rocket with star explosion.

**Parameters**

- **screen** – The Screen being used for the Scene.
- **x** – The column (x coordinate) for the origin of the effect.
- **y** – The line (y coordinate) for the origin of the effect.
- **life_time** – The life time of the effect.

Also see the common keyword arguments in *Effect*.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**process_event** (*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene** (*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset** ()
Reset the particle effect back to its initial state. This must be implemented by the child classes.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update** (*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.particles.**StarTrail** (*screen*, *x*, *y*, *life_time*, *colour*)
Bases: *asciimatics.particles.ParticleEmitter*

A trail for a *StarExplosion*.

**Parameters**

- **screen** – The Screen being used for this particle system.

- **x** – The column (x coordinate) for the origin of this trail.
- **y** – The line (y coordinate) for the origin of this trail.
- **life_time** – The life time of this trail.
- **colour** – The colour of this trail.

**update**()
The function to draw a new frame for the particle system.

## 8.1.9 asciimatics.paths module

This module provides *Paths* to create animation effects with Sprites. For more details see http://asciimatics.readthedocs.io/en/latest/animation.html

**class** asciimatics.paths.**DynamicPath**(*screen*, *x*, *y*)
Bases: `asciimatics.paths._AbstractPath`

Class to create a dynamic path that reacts to events

The Screen will reset() the Path before iterating through each position using next_pos() and checking whether it has reached the end using is_finished().

To implement a DynamicPath, override the `process_event()` method to react to any user input.

**is_finished**()

**Returns** Whether this path has got to the end.

**next_pos**()

**Returns** The next position tuple (x, y) for the Sprite on this path.

**process_event**(*event*)
Process any mouse event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**reset**()
Reset the Path for use next time.

**class** asciimatics.paths.**Path**
Bases: `asciimatics.paths._AbstractPath`

Class to record and play back the motion of a Sprite.

The Screen will reset() the Path before iterating through each position using next_pos() and checking whether it has reached the end using is_finished().

To define a Path, use the methods to jump to a location, wait or move between points.

**is_finished**()

**Returns** Whether this path has got to the end.

**jump_to**(*x*, *y*)
Jump straight to the newly specified location - i.e. teleport there and don't create a path to get there.

**Parameters**

- **x** – X coord for the end position.
- **y** – Y coord for the end position.

**move_round_to** (*points*, *steps*)
Follow a path pre-defined by a set of at least 4 points. This Path will interpolate the points into a curve and follow that curve.

**Parameters**

- **points** – The list of points that defines the path.
- **steps** – The number of steps to take to follow the path.

**move_straight_to** (*x*, *y*, *steps*)
Move straight to the newly specified location - i.e. create a straight line Path from the current location to the specified point.

**Parameters**

- **x** – X coord for the end position.
- **y** – Y coord for the end position.
- **steps** – How many steps to take for the move.

**next_pos** ()

**Returns** The next position tuple (x, y) for the Sprite on this path.

**reset** ()
Reset the Path for use next time.

**wait** (*delay*)
Wait at the current location for the specified number of iterations.

**Parameters** **delay** – The time to wait (in animation frames).

### 8.1.10 asciimatics.scene module

This module defines Scene objects for animation purposes. For more details, see http://asciimatics.readthedocs.io/en/latest/animation.html

**class** asciimatics.scene.**Scene** (*effects*, *duration=0*, *clear=True*, *name=None*)
Bases: object

Class to store the details of a single scene to be displayed. This is made up of a set of *Effect* objects. See the documentation for Effect to understand the interaction between the two classes and http://asciimatics.readthedocs.io/en/latest/animation.html for how to use them together.

**Parameters**

- **effects** – The list of effects to apply to this scene.
- **duration** – The number of frames in this Scene. A value of 0 means that the Scene should query the Effects to find the duration. A value of -1 means don't stop.
- **clear** – Whether to clear the Screen at the start of the Scene.
- **name** – Optional name to identify the scene.

**add_effect** (*effect*, *reset=True*)
Add an effect to the Scene.

This method can be called at any time - even when playing the Scene. The default logic assumes that the Effect needs to be reset before being displayed. This can be overridden using the *reset* parameter.

**Parameters**

- **effect** – The Effect to be added.
- **reset** – Whether to reset the Effect that has just been added.

**clear**

**Returns** Whether the Scene should clear at the start.

**duration**

**Returns** The length of the scene in frames.

**effects**

**Returns** The list of Effects in this Scene.

**exit**()

Handle any tidy up required on the exit of the Scene.

**name**

**Returns** The name of this Scene. May be None.

**process_event**(*event*)

Process a new input event.

This method will pass the event on to any Effects in reverse Z order so that the top-most Effect has priority.

**Parameters** **event** – The Event that has been triggered.

**Returns** None if the Scene processed the event, else the original event.

**remove_effect**(*effect*)

Remove an effect from the scene.

**Parameters** **effect** – The effect to remove.

**reset**(*old_scene=None*, *screen=None*)

Reset the scene ready for playing.

**Parameters**

- **old_scene** – The previous version of this Scene that was running before the application reset - e.g. due to a screen resize.
- **screen** – New screen to use if old_scene is not None.

## 8.1.11 asciimatics.screen module

This module defines common screen output function. For more details, see http://asciimatics.readthedocs.io/en/latest/io.html

**class** asciimatics.screen.**Canvas**(*screen*, *height*, *width*, *x=None*, *y=None*)

Bases: asciimatics.screen._AbstractCanvas

A Canvas is an object that can be used to draw to the screen. It maintains its own buffer that will be flushed to the screen when *refresh()* is called.

**Parameters**

- **screen** – The underlying Screen that will be drawn to on refresh.
- **height** – The height of the screen buffer to be used.
- **width** – The width of the screen buffer to be used.
- **x** – The x position for the top left corner of the Canvas.

- **y** – The y position for the top left corner of the Canvas.

If either of the x or y positions is not set, the Canvas will default to centring within the current Screen for that location.

`block_transfer`(*buffer*, *x*, *y*)
Copy a buffer to the screen double buffer at a specified location.

**Parameters**

- **buffer** – The double buffer to copy
- **x** – The X origin for where to place it in the Screen
- **y** – The Y origin for where to place it in the Screen

`centre`(*text*, *y*, *colour=7*, *attr=0*, *colour_map=None*)
Centre the text on the specified line (y) using the optional colour and attributes.

**Parameters**

- **text** – The (single line) text to be printed.
- **y** – The line (y coord) for the start of the text.
- **colour** – The colour of the text to be displayed.
- **attr** – The cell attribute of the text to be displayed.
- **colour_map** – Colour/attribute list for multi-colour text.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class.

`clear_buffer`(*fg*, *attr*, *bg*, *x=0*, *y=0*, *w=None*, *h=None*)
Clear a box in the current double-buffer used by this object.

This is the recommended way to clear parts, or all, ofthe Screen without causing flicker as it will only become visible at the next refresh. Defaults to the whole buffer if no box is specified.

**Parameters**

- **fg** – The foreground colour to use for the new buffer.
- **attr** – The attribute value to use for the new buffer.
- **bg** – The background colour to use for the new buffer.
- **x** – Optional X coordinate for top left of box.
- **y** – Optional Y coordinate for top left of box.
- **w** – Optional width of the box.
- **h** – Optional height of the box.

`dimensions`

**Returns** The full dimensions of the canvas as a (height, width) tuple.

`draw`(*x*, *y*, *char=None*, *colour=7*, *bg=0*, *thin=False*)
Draw a line from drawing cursor to the specified position.

This uses a modified Bressenham algorithm, interpolating twice as many points to render down to anti-aliased characters when no character is specified, or uses standard algorithm plotting with the specified character.

**Parameters**

- **x** – The column (x coord) for the location to check.

- **y** – The line (y coord) for the location to check.
- **char** – Optional character to use to draw the line.
- **colour** – Optional colour for plotting the line.
- **bg** – Optional background colour for plotting the line.
- **thin** – Optional width of anti-aliased line.

**fill_polygon** (*polygons*, *colour=7*, *bg=0*)
Draw a filled polygon.

This function uses the scan line algorithm to create the polygon. See https://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/PolygonFilling.html for details.

**Parameters**

- **polygons** – A list of polygons (which are each a list of (x,y) coordinates for the points of the polygon) - i.e. nested list of 2-tuples.
- **colour** – The foreground colour to use for the polygon
- **bg** – The background colour to use for the polygon

**get_from** (*x*, *y*)
Get the character at the specified location.

**Parameters**

- **x** – The column (x coord) of the character.
- **y** – The row (y coord) of the character.

**Returns** A 4-tuple of (ascii code, foreground, attributes, background) for the character at the location.

**highlight** (*x*, *y*, *w*, *h*, *fg=None*, *bg=None*, *blend=100*)
Highlight a specified section of the screen.

**Parameters**

- **x** – The column (x coord) for the start of the highlight.
- **y** – The line (y coord) for the start of the highlight.
- **w** – The width of the highlight (in characters).
- **h** – The height of the highlight (in characters).
- **fg** – The foreground colour of the highlight.
- **bg** – The background colour of the highlight.
- **blend** – How much (as a percentage) to take of the new colour when blending.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class. If fg or bg are None that means don't change the foreground/background as appropriate.

**is_visible** (*x*, *y*)
Return whether the specified location is on the visible screen.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.

**move** (*x*, *y*)
Move the drawing cursor to the specified position.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.

**origin**
The location of top left corner of the canvas on the Screen.

**Returns** A tuple (x, y) of the location

**paint** (*text*, *x*, *y*, *colour=7*, *attr=0*, *bg=0*, *transparent=False*, *colour_map=None*)
Paint multi-colour text at the defined location.

**Parameters**

- **text** – The (single line) text to be printed.
- **x** – The column (x coord) for the start of the text.
- **y** – The line (y coord) for the start of the text.
- **colour** – The default colour of the text to be displayed.
- **attr** – The default cell attribute of the text to be displayed.
- **bg** – The default background colour of the text to be displayed.
- **transparent** – Whether to print spaces or not, thus giving a transparent effect.
- **colour_map** – Colour/attribute list for multi-colour text.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class. colour_map is a list of tuples (foreground, attribute, background) that must be the same length as the passed in text (or None if no mapping is required).

**palette**

**Returns** A palette compatible with the PIL.

**print_at** (*text*, *x*, *y*, *colour=7*, *attr=0*, *bg=0*, *transparent=False*)
Print the text at the specified location using the specified colour and attributes.

**Parameters**

- **text** – The (single line) text to be printed.
- **x** – The column (x coord) for the start of the text.
- **y** – The line (y coord) for the start of the text.
- **colour** – The colour of the text to be displayed.
- **attr** – The cell attribute of the text to be displayed.
- **bg** – The background colour of the text to be displayed.
- **transparent** – Whether to print spaces or not, thus giving a transparent effect.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class.

**refresh** ()
Flush the canvas content to the underlying screen.

**reset** ()
Reset the internal buffers for the abstract canvas.

**scroll**(*lines=1*)
Scroll the abstract canvas up one line.

**Parameters** **lines** – The number of lines to scroll. Defaults to down by one.

**scroll_to**(*line*)
Scroll the abstract canvas to make a specific line.

**Parameters** **line** – The line to scroll to.

**start_line**

**Returns** The start line of the top of the canvas.

**unicode_aware**

**Returns** Whether unicode input/output is supported or not.

**class** asciimatics.screen.**ManagedScreen**(*func=<function ManagedScreen.<lambda>>*)
Bases: `object`

Decorator and class to create a managed Screen. It can be used in two ways. If used as a method decorator it will create and open a new Screen, pass the screen to the method as a keyword argument, and close the screen when the method has completed. If used with the with statement the class will create and open a new Screen, return the screen for using in the block, and close the screen when the statement ends. Note that any arguments are in this class so that you can use it as a decorator or using the with statment. No arguments are required to use.

**Parameters** **func** – The function to call once the Screen has been created.

**class** asciimatics.screen.**Screen**(*height, width, buffer_height, unicode_aware*)
Bases: `asciimatics.screen._AbstractCanvas`

Class to track basic state of the screen. This constructs the necessary resources to allow us to do the ASCII animations.

This is an abstract class that will build the correct concrete class for you when you call `wrapper()`. If needed, you can use the `open()` and `close()` methods for finer grained control of the construction and tidy up.

Note that you need to define the required height for your screen buffer. This is important if you plan on using any Effects that will scroll the screen vertically (e.g. Scroll). It must be big enough to handle the full scrolling of your selected Effect.

Don't call this constructor directly.

**block_transfer**(*buffer, x, y*)
Copy a buffer to the screen double buffer at a specified location.

**Parameters**

- **buffer** – The double buffer to copy
- **x** – The X origin for where to place it in the Screen
- **y** – The Y origin for where to place it in the Screen

**centre**(*text, y, colour=7, attr=0, colour_map=None*)
Centre the text on the specified line (y) using the optional colour and attributes.

**Parameters**

- **text** – The (single line) text to be printed.
- **y** – The line (y coord) for the start of the text.
- **colour** – The colour of the text to be displayed.

- **attr** – The cell attribute of the text to be displayed.
- **colour_map** – Colour/attribute list for multi-colour text.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class.

**clear**()

Clear the Screen of all content.

Note that this will instantly clear the Screen and reset all buffers to the default state, without waiting for you to call `refresh()`. It is designed for use once at the start of your application to reset all buffers and the screen to a known state.

If you want to clear parts, or all, of the Screen inside your application without any flicker, use `clear_buffer()` instead.

**clear_buffer**(*fg*, *attr*, *bg*, *x=0*, *y=0*, *w=None*, *h=None*)

Clear a box in the current double-buffer used by this object.

This is the recommended way to clear parts, or all, ofthe Screen without causing flicker as it will only become visible at the next refresh. Defaults to the whole buffer if no box is specified.

**Parameters**

- **fg** – The foreground colour to use for the new buffer.
- **attr** – The attribute value to use for the new buffer.
- **bg** – The background colour to use for the new buffer.
- **x** – Optional X coordinate for top left of box.
- **y** – Optional Y coordinate for top left of box.
- **w** – Optional width of the box.
- **h** – Optional height of the box.

**close**(*restore=True*)

Close down this Screen and tidy up the environment as required.

**Parameters** **restore** – whether to restore the environment or not.

**static ctrl**(*char*)

Calculate the control code for a given key. For example, this converts "a" to 1 (which is the code for ctrl-a).

**Parameters** **char** – The key to convert to a control code.

**Returns** The control code as an integer or None if unknown.

**current_scene**

**Returns** The scene currently being rendered. To be used in conjunction with `draw_next_frame()`.

**dimensions**

**Returns** The full dimensions of the canvas as a (height, width) tuple.

**draw**(*x*, *y*, *char=None*, *colour=7*, *bg=0*, *thin=False*)

Draw a line from drawing cursor to the specified position.

This uses a modified Bressenham algorithm, interpolating twice as many points to render down to anti-aliased characters when no character is specified, or uses standard algorithm plotting with the specified character.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.
- **char** – Optional character to use to draw the line.
- **colour** – Optional colour for plotting the line.
- **bg** – Optional background colour for plotting the line.
- **thin** – Optional width of anti-aliased line.

**draw_next_frame** (*repeat=True*)
Draw the next frame in the currently configured Scenes. You must call `set_scenes()` before using this for the first time.

**Parameters** **repeat** – Whether to repeat the Scenes once it has reached the end. Defaults to True.

**Raises** ***StopApplication*** – if the application should be terminated.

**fill_polygon** (*polygons*, *colour=7*, *bg=0*)
Draw a filled polygon.

This function uses the scan line algorithm to create the polygon. See https://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/PolygonFilling.html for details.

**Parameters**

- **polygons** – A list of polygons (which are each a list of (x,y) coordinates for the points of the polygon) - i.e. nested list of 2-tuples.
- **colour** – The foreground colour to use for the polygon
- **bg** – The background colour to use for the polygon

**force_update** (*full_refresh=False*)
Force the Screen to redraw the current Scene on the next call to draw_next_frame, overriding the frame_update_count value for all the Effects.

**Parameters** **full_refresh** – if True force the whole screen to redraw.

**get_event** ()
Check for any events (e.g. key-press or mouse movement) without waiting.

**Returns** A `Event` object if anything was detected, otherwise it returns None.

**get_from** (*x*, *y*)
Get the character at the specified location.

**Parameters**

- **x** – The column (x coord) of the character.
- **y** – The row (y coord) of the character.

**Returns** A 4-tuple of (ascii code, foreground, attributes, background) for the character at the location.

**get_key** ()
Check for a key without waiting. This method is deprecated. Use `get_event()` instead.

**getch** (*x*, *y*)
Get the character at a specified location. This method is deprecated. Use `get_from()` instead.

**Parameters**

- **x** – The x coordinate.
- **y** – The y coordinate.

**has_resized**()
Check whether the screen has been re-sized.

**Returns** True when the screen has been re-sized since the last check.

**highlight**(*x*, *y*, *w*, *h*, *fg=None*, *bg=None*, *blend=100*)
Highlight a specified section of the screen.

**Parameters**

- **x** – The column (x coord) for the start of the highlight.
- **y** – The line (y coord) for the start of the highlight.
- **w** – The width of the highlight (in characters).
- **h** – The height of the highlight (in characters).
- **fg** – The foreground colour of the highlight.
- **bg** – The background colour of the highlight.
- **blend** – How much (as a percentage) to take of the new colour when blending.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class. If fg or bg are None that means don't change the foreground/background as appropriate.

**is_visible**(*x*, *y*)
Return whether the specified location is on the visible screen.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.

**move**(*x*, *y*)
Move the drawing cursor to the specified position.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.

**classmethod open**(*height=None*, *catch_interrupt=False*, *unicode_aware=None*)
Construct a new Screen for any platform. This will just create the correct Screen object for your environment. See *wrapper()* for a function to create and tidy up once you've finished with the Screen.

**Parameters**

- **height** – The buffer height for this window (for testing only).
- **catch_interrupt** – Whether to catch and prevent keyboard interrupts. Defaults to False to maintain backwards compatibility.
- **unicode_aware** – Whether the application can use unicode or not. If None, try to detect from the environment if UTF-8 is enabled.

**paint**(*text*, *x*, *y*, *colour=7*, *attr=0*, *bg=0*, *transparent=False*, *colour_map=None*)
Paint multi-colour text at the defined location.

**Parameters**

- **`text`** – The (single line) text to be printed.
- **`x`** – The column (x coord) for the start of the text.
- **`y`** – The line (y coord) for the start of the text.
- **`colour`** – The default colour of the text to be displayed.
- **`attr`** – The default cell attribute of the text to be displayed.
- **`bg`** – The default background colour of the text to be displayed.
- **`transparent`** – Whether to print spaces or not, thus giving a transparent effect.
- **`colour_map`** – Colour/attribute list for multi-colour text.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class. colour_map is a list of tuples (foreground, attribute, background) that must be the same length as the passed in text (or None if no mapping is required).

**`palette`**

**Returns** A palette compatible with the PIL.

**`play`** (*scenes*, *stop_on_resize=False*, *unhandled_input=None*, *start_scene=None*, *repeat=True*, *allow_int=False*)

Play a set of scenes.

This is effectively a helper function to wrap `set_scenes()` and `draw_next_frame()` to simplify animation for most applications.

**Parameters**

- **`scenes`** – a list of `Scene` objects to play.
- **`stop_on_resize`** – Whether to stop when the screen is resized. Default is to carry on regardless - which will typically result in an error. This is largely done for back-compatibility.
- **`unhandled_input`** – Function to call for any input not handled by the Scenes/Effects being played. Defaults to a function that closes the application on “Q” or “X” being pressed.
- **`start_scene`** – The old Scene to start from. This must have name that matches the name of one of the Scenes passed in.
- **`repeat`** – Whether to repeat the Scenes once it has reached the end. Defaults to True.
- **`allow_int`** – Allow input to interrupt frame rate delay.

**Raises** **`ResizeScreenError`** – if the screen is resized (and allowed by stop_on_resize).

The unhandled input function just takes one parameter - the input event that was not handled.

**`print_at`** (*text*, *x*, *y*, *colour=7*, *attr=0*, *bg=0*, *transparent=False*)

Print the text at the specified location using the specified colour and attributes.

**Parameters**

- **`text`** – The (single line) text to be printed.
- **`x`** – The column (x coord) for the start of the text.
- **`y`** – The line (y coord) for the start of the text.
- **`colour`** – The colour of the text to be displayed.
- **`attr`** – The cell attribute of the text to be displayed.

- **bg** – The background colour of the text to be displayed.
- **transparent** – Whether to print spaces or not, thus giving a transparent effect.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class.

**putch**(*text*, *x*, *y*, *colour=7*, *attr=0*, *bg=0*, *transparent=False*)
Print text at the specified location. This method is deprecated. Use *print_at()* instead.

**Parameters**

- **text** – The (single line) text to be printed.
- **x** – The column (x coord) for the start of the text.
- **y** – The line (y coord) for the start of the text.
- **colour** – The colour of the text to be displayed.
- **attr** – The cell attribute of the text to be displayed.
- **bg** – The background colour of the text to be displayed.
- **transparent** – Whether to print spaces or not, thus giving a transparent effect.

**refresh**()
Refresh the screen.

**reset**()
Reset the internal buffers for the abstract canvas.

**scroll**(*lines=1*)
Scroll the abstract canvas up one line.

**Parameters** **lines** – The number of lines to scroll. Defaults to down by one.

**scroll_to**(*line*)
Scroll the abstract canvas to make a specific line.

**Parameters** **line** – The line to scroll to.

**set_scenes**(*scenes*, *unhandled_input=None*, *start_scene=None*)
Remember a set of scenes to be played. This must be called before using *draw_next_frame()*.

**Parameters**

- **scenes** – a list of *Scene* objects to play.
- **unhandled_input** – Function to call for any input not handled by the Scenes/Effects being played. Defaults to a function that closes the application on "Q" or "X" being pressed.
- **start_scene** – The old Scene to start from. This must have name that matches the name of one of the Scenes passed in.

**Raises** ***ResizeScreenError*** – if the screen is resized (and allowed by stop_on_resize).

The unhandled input function just takes one parameter - the input event that was not handled.

**set_title**(*title*)
Set the title for this terminal/console session. This will typically change the text displayed in the window title bar.

**Parameters** **title** – The title to be set.

**start_line**

**Returns** The start line of the top of the canvas.

**`unicode_aware`**

**Returns** Whether unicode input/output is supported or not.

**`wait_for_input`** (*timeout*)
Wait until there is some input or the timeout is hit.

**Parameters** **`timeout`** – Time to wait for input in seconds (floating point).

**`classmethod wrapper`** (*func*, *height=None*, *catch_interrupt=False*, *arguments=None*, *unicode_aware=None*)
Construct a new Screen for any platform. This will initialize the Screen, call the specified function and then tidy up the system as required when the function exits.

**Parameters**

- **`func`** – The function to call once the Screen has been created.
- **`height`** – The buffer height for this Screen (only for test purposes).
- **`catch_interrupt`** – Whether to catch and prevent keyboard interrupts. Defaults to False to maintain backwards compatibility.
- **`arguments`** – Optional arguments list to pass to func (after the Screen object).
- **`unicode_aware`** – Whether the application can use unicode or not. If None, try to detect from the environment if UTF-8 is enabled.

**`class`** `asciimatics.screen.`**`TemporaryCanvas`** (*height*, *width*)
Bases: `asciimatics.screen._AbstractCanvas`

A TemporaryCanvas is an object that can only be used to draw to a buffer.

This class is desigend purely for use by dynamic renderers and so ignores some features of a full Canvas - most notably the screen related fhnction (e.g. the screen buffer and related properties).

**Parameters**

- **`height`** – The height of the screen buffer to be used.
- **`width`** – The width of the screen buffer to be used.

**`block_transfer`** (*buffer*, *x*, *y*)
Copy a buffer to the screen double buffer at a specified location.

**Parameters**

- **`buffer`** – The double buffer to copy
- **`x`** – The X origin for where to place it in the Screen
- **`y`** – The Y origin for where to place it in the Screen

**`centre`** (*text*, *y*, *colour=7*, *attr=0*, *colour_map=None*)
Centre the text on the specified line (y) using the optional colour and attributes.

**Parameters**

- **`text`** – The (single line) text to be printed.
- **`y`** – The line (y coord) for the start of the text.
- **`colour`** – The colour of the text to be displayed.
- **`attr`** – The cell attribute of the text to be displayed.
- **`colour_map`** – Colour/attribute list for multi-colour text.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class.

**clear_buffer** (*fg*, *attr*, *bg*, *x=0*, *y=0*, *w=None*, *h=None*)

Clear a box in the current double-buffer used by this object.

This is the recommended way to clear parts, or all, ofthe Screen without causing flicker as it will only become visible at the next refresh. Defaults to the whole buffer if no box is specified.

**Parameters**

- **fg** – The foreground colour to use for the new buffer.
- **attr** – The attribute value to use for the new buffer.
- **bg** – The background colour to use for the new buffer.
- **x** – Optional X coordinate for top left of box.
- **y** – Optional Y coordinate for top left of box.
- **w** – Optional width of the box.
- **h** – Optional height of the box.

**dimensions**

**Returns** The full dimensions of the canvas as a (height, width) tuple.

**draw** (*x*, *y*, *char=None*, *colour=7*, *bg=0*, *thin=False*)

Draw a line from drawing cursor to the specified position.

This uses a modified Bressenham algorithm, interpolating twice as many points to render down to anti-aliased characters when no character is specified, or uses standard algorithm plotting with the specified character.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.
- **char** – Optional character to use to draw the line.
- **colour** – Optional colour for plotting the line.
- **bg** – Optional background colour for plotting the line.
- **thin** – Optional width of anti-aliased line.

**fill_polygon** (*polygons*, *colour=7*, *bg=0*)

Draw a filled polygon.

This function uses the scan line algorithm to create the polygon. See https://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/PolygonFilling.html for details.

**Parameters**

- **polygons** – A list of polygons (which are each a list of (x,y) coordinates for the points of the polygon) - i.e. nested list of 2-tuples.
- **colour** – The foreground colour to use for the polygon
- **bg** – The background colour to use for the polygon

**get_from** (*x*, *y*)

Get the character at the specified location.

**Parameters**

- **x** – The column (x coord) of the character.

- **y** – The row (y coord) of the character.

**Returns** A 4-tuple of (ascii code, foreground, attributes, background) for the character at the location.

**highlight** (*x*, *y*, *w*, *h*, *fg=None*, *bg=None*, *blend=100*)
Highlight a specified section of the screen.

**Parameters**

- **x** – The column (x coord) for the start of the highlight.
- **y** – The line (y coord) for the start of the highlight.
- **w** – The width of the highlight (in characters).
- **h** – The height of the highlight (in characters).
- **fg** – The foreground colour of the highlight.
- **bg** – The background colour of the highlight.
- **blend** – How much (as a percentage) to take of the new colour when blending.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class. If fg or bg are None that means don't change the foreground/background as appropriate.

**is_visible** (*x*, *y*)
Return whether the specified location is on the visible screen.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.

**move** (*x*, *y*)
Move the drawing cursor to the specified position.

**Parameters**

- **x** – The column (x coord) for the location to check.
- **y** – The line (y coord) for the location to check.

**paint** (*text*, *x*, *y*, *colour=7*, *attr=0*, *bg=0*, *transparent=False*, *colour_map=None*)
Paint multi-colour text at the defined location.

**Parameters**

- **text** – The (single line) text to be printed.
- **x** – The column (x coord) for the start of the text.
- **y** – The line (y coord) for the start of the text.
- **colour** – The default colour of the text to be displayed.
- **attr** – The default cell attribute of the text to be displayed.
- **bg** – The default background colour of the text to be displayed.
- **transparent** – Whether to print spaces or not, thus giving a transparent effect.
- **colour_map** – Colour/attribute list for multi-colour text.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class. colour_map is a list of tuples (foreground, attribute, background) that must be the same length as the passed in text (or None if no mapping is required).

**palette**

**Returns** A palette compatible with the PIL.

**print_at** (*text*, *x*, *y*, *colour=7*, *attr=0*, *bg=0*, *transparent=False*)

Print the text at the specified location using the specified colour and attributes.

**Parameters**

- **text** – The (single line) text to be printed.
- **x** – The column (x coord) for the start of the text.
- **y** – The line (y coord) for the start of the text.
- **colour** – The colour of the text to be displayed.
- **attr** – The cell attribute of the text to be displayed.
- **bg** – The background colour of the text to be displayed.
- **transparent** – Whether to print spaces or not, thus giving a transparent effect.

The colours and attributes are the COLOUR_xxx and A_yyy constants defined in the Screen class.

**refresh** ()

Refresh this object - this will draw to the underlying display interface.

**reset** ()

Reset the internal buffers for the abstract canvas.

**scroll** (*lines=1*)

Scroll the abstract canvas up one line.

**Parameters** **lines** – The number of lines to scroll. Defaults to down by one.

**scroll_to** (*line*)

Scroll the abstract canvas to make a specific line.

**Parameters** **line** – The line to scroll to.

**start_line**

**Returns** The start line of the top of the canvas.

**unicode_aware**

**Returns** Whether unicode input/output is supported or not.

### 8.1.12 asciimatics.sprites module

This module provides *Sprites* to create animation effects with Paths. For more details see http://asciimatics.readthedocs.io/en/latest/animation.html

**class** asciimatics.sprites.**Arrow** (*screen*, *path*, *colour=7*, *start_frame=0*, *stop_frame=0*)

Bases: *asciimatics.effects.Sprite*

Sample arrow sprite - points where it is going.

See *Sprite* for details.

**delete_count**

The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**last_position**()
Returns the last position of this Sprite as a tuple (x, y, width, height).

**overlaps**(*other*, *use_new_pos=False*)
Check whether this Sprite overlaps another.

**Parameters**

- **other** – The other Sprite to check for an overlap.
- **use_new_pos** – Whether to use latest position (due to recent update). Defaults to False.

**Returns** True if the two Sprites overlap.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.sprites.**Plot**(*screen*, *path*, *colour=7*, *start_frame=0*, *stop_frame=0*)
Bases: *asciimatics.effects.Sprite*

Sample Sprite that simply plots an "X" for each step in the path. Useful for plotting a path to the screen.

See *Sprite* for details.

**delete_count**
The number of frames before this Effect should be deleted.

**frame_update_count**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**last_position**()
Returns the last position of this Sprite as a tuple (x, y, width, height).

**overlaps**(*other*, *use_new_pos=False*)
Check whether this Sprite overlaps another.

**Parameters**

- **other** – The other Sprite to check for an overlap.
- **use_new_pos** – Whether to use latest position (due to recent update). Defaults to False.

**Returns** True if the two Sprites overlap.

**process_event**(*event*)
Process any input event.

**Parameters** **event** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**register_scene**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **scene** – The Scene to be registered

**reset**()
Function to reset the effect when replaying the scene.

**safe_to_default_unhandled_input**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**scene**
The Scene that owns this Effect.

**screen**
The Screen that will render this Effect.

**stop_frame**
Last frame for this effect. A value of zero means no specific end.

**update**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **frame_no** – The index of the frame being generated.

**class** asciimatics.sprites.**Sam**(*screen*, *path*, *start_frame=0*, *stop_frame=0*)
Bases: *asciimatics.effects.Sprite*

Sam Paul sprite - an simple sample animated character.

See `Sprite` for details.

**`delete_count`**
The number of frames before this Effect should be deleted.

**`frame_update_count`**
The number of frames before this Effect should be updated.

Increasing this number potentially reduces the CPU load of a Scene (if no other Effect needs to be scheduled sooner), but can affect perceived responsiveness of the Scene if it is too long. Handle with care!

A value of 0 means refreshes are not required beyond a response to an input event. It defaults to 1 for all Effects.

**`last_position`**()
Returns the last position of this Sprite as a tuple (x, y, width, height).

**`overlaps`**(*other*, *use_new_pos=False*)
Check whether this Sprite overlaps another.

**Parameters**

- **`other`** – The other Sprite to check for an overlap.
- **`use_new_pos`** – Whether to use latest position (due to recent update). Defaults to False.

**Returns** True if the two Sprites overlap.

**`process_event`**(*event*)
Process any input event.

**Parameters** **`event`** – The event that was triggered.

**Returns** None if the Effect processed the event, else the original event.

**`register_scene`**(*scene*)
Register the Scene that owns this Effect.

**Parameters** **`scene`** – The Scene to be registered

**`reset`**()
Function to reset the effect when replaying the scene.

**`safe_to_default_unhandled_input`**
Whether it is safe to use the default handler for any unhandled input from this Effect.

A value of False means that asciimatics should not use the default handler. This is typically the case for Frames.

**`scene`**
The Scene that owns this Effect.

**`screen`**
The Screen that will render this Effect.

**`stop_frame`**
Last frame for this effect. A value of zero means no specific end.

**`update`**(*frame_no*)
Process the animation effect for the specified frame number.

**Parameters** **`frame_no`** – The index of the frame being generated.

### 8.1.13 asciimatics.strings module

This module provides classes to handle embedded control strings for widgets.

**class** asciimatics.strings.**ColouredText**(*raw_text*, *parser*, *colour=None*, *colour_map=None*, *offsets=None*, *text=None*)

Bases: `object`

Unicode string-like object to store text and colour maps, using a parser to convert the raw text passed in into visible text and an associated colour map. This only handles simple colour change commands and will ignore more complex commands).

**Parameters**

- **raw_text** – The raw unicode string to be processed
- **parser** – The parser to process the text
- **colour** – Optional starting colour tuple to use for this text.
- **colour_map** – Optional ready parsed colour map for this text.
- **offsets** – Optional ready parsed offsets for this text.
- **text** – Optional ready parsed text for this text.

The colour_map, offsets and text options are to optimize creation of substrings from an existing ColouredText object and should not be used in general.

**colour_map**

Colour map for the processed text (for use with *paint* method).

**first_colour**

First colour triplet used for this text.

**join**(*others*)

Join the list of ColouredObjects using this ColouredObject.

**Parameters** **others** – the list of other objects to join.

**last_colour**

Last colour triplet used for this text.

**raw_text**

Raw (unprocessed) text for this object.

**startswith**(*text*)

Check whether parsed (i.e. displayed) text starts woth specified string.

### 8.1.14 asciimatics.utilities module

This module is just a collection of simple helper functions.

**class** asciimatics.utilities.**BoxTool**(*unicode_aware*, *style=1*)

Bases: `object`

Tool for building boxes out of characters.

Supports a variety of line styles from *asciimatics.constants*:

- `ASCII_LINE` – ASCII safe characters (0)
- `SINGLE_LINE` – Unicode based single lined box (1)
- `DOUBLE_LINE` – Unicode based double lined box (2)

Individual characters of a box can be accessed directly through attributes. Most attribute names are based on the Extended-ASCII characters used for the UNICODE version of the shape. The names describe the directions the piece point in (not the part of the box). For example, "up_left" is a corner piece that has a part that points up and a part that points left – the character usually used for the bottom right hand corner of a box.

Attribute values are:

- `up_left` – corner piece facing up and left
- `up_right` – corner piece facing up and right
- `down_left` – corner piece facing down and left
- `down_right` – corner piece facing down and right
- `h` – horizontal line
- `v` – vertical line
- `v_inside` – vertical line used inside the grid
- `v_left` – vertical line with mid joiner facing to the left
- `v_right` – vertical line with mid joiner facing to the right
- `h_up` – horizontal line with a mid joiner facing up
- `h_down` – horizontal line with a mid joiner facing down
- `cross` – intersection between vertical and horizontal

**Parameters**

- **`unicode_aware`** – boolean indicating if the terminal is Unicode aware. If False, will force the use of the ASCII style
- **`style`** – line style specifier. Supports `ASCII_LINE`, `SINGLE_LINE`, and `DOUBLE_LINE`. Defaults to `SINGLE_LINE`.

**`box`**(*width*, *height*)
Returns a string containing a box with the given width and height.

**`box_bottom`**(*width*)
Returns a string containing the bottom border of a box

**Parameters** **`width`** – width of box, including corners

**`box_line`**(*width*)
Returns a string with a vertical bar on each end, padded with spaces in between for the given width.

**Parameters** **`width`** – width of box including sides

**`box_top`**(*width*)
Returns a string containing the top border of a box

**Parameters** **`width`** – width of box, including corners

**`style`**
The line drawing style used to draw boxes. Possible styles are set in *`asciimatics.constants`*.

**Parameters** **`style`** – One of `ASCII_LINE`, `SINGLE_LINE`, or `DOUBLE_LINE`

`asciimatics.utilities.`**`readable_mem`**(*mem*)

**Parameters** **`mem`** – An integer number of bytes to convert to human-readable form.

**Returns** A human-readable string representation of the number.

asciimatics.utilities.**readable_timestamp**(*stamp*)

**Parameters** **stamp** – A floating point number representing the POSIX file timestamp.

**Returns** A short human-readable string representation of the timestamp.

### 8.1.15 asciimatics.version module

### 8.1.16 Module contents

Asciimatics is a package to help people create full-screen text UIs (from interactive forms to ASCII animations) on any platform. It is licensed under the Apache Software Foundation License 2.0.

CHAPTER 9

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index