

# Slideshow: Figure and Presentation Tools

Version 6.9

Matthew Flatt  
and Robert Bruce Findler

May 1, 2017

Slideshow is a library for creating presentation slides. Unlike Powerpoint, Slideshow provides no WYSIWYG interface for constructing slides. Instead, like SliTeX, a presentation is generated by a program.

To get started, run the `slideshow` executable, and click the Run Tutorial link.

To learn more about Slideshow, see also “Slideshow: Functional Presentations” [Findler06].

```
#lang slideshow      package: slideshow-lib
```

Most of the bindings defined in the manual are provided by the `slideshow` language, which also re-exports all of `racket` except for `printable<%/>` (due to backward-compatibility issues) and all of `pict`.

# Contents

<b>1</b>	<b>Creating Slide Presentations</b>	<b>3</b>
1.1	Slide Basics . . . . .	3
1.2	Staging Slides . . . . .	4
1.3	Display Size and Fonts . . . . .	6
1.4	Command-line Options . . . . .	6
1.5	Printing . . . . .	7
<b>2</b>	<b>Making Slides</b>	<b>8</b>
2.1	Primary Slide Functions . . . . .	8
2.2	Slide Registration . . . . .	13
2.3	Viewer Control . . . . .	13
2.4	Constants and Layout Variables . . . . .	15
2.5	Configuration . . . . .	17
2.6	Pict-Staging Helper . . . . .	19
2.7	Text Formatting Helpers . . . . .	20
2.8	Slides to Picts . . . . .	21
<b>3</b>	<b>Typesetting Racket Code in Slideshow</b>	<b>23</b>
<b>4</b>	<b>Animations</b>	<b>24</b>
4.1	Generating Animated Slides . . . . .	24
<b>5</b>	<b>Legacy Libraries</b>	<b>27</b>
	<b>Bibliography</b>	<b>28</b>
	<b>Index</b>	<b>29</b>
	<b>Index</b>	<b>29</b>

# 1 Creating Slide Presentations

The `slideshow` module acts as a language that includes:

- all of `racket` except for `printable<%>`;
- pict-creating functions from `pict`; and
- slide-composing functions from `slideshow/base`.

The `slideshow` and `slideshow/base` module initialization also check the `current-command-line-arguments` parameter to configure the slide mode (e.g., printing).

The rest of this section repeats information that is presented by the tutorial slideshow, which can be viewed by running the `slideshow` executable and clicking the Run Tutorial link.

## 1.1 Slide Basics

The main Slideshow function is `slide`, which adds a slide to the presentation with a given content. For example, the “Hello World” presentation can be defined by the following module:

```
#lang slideshow

(slide
 #:title "How to Say Hello"
 (t "Hello World!"))
```

The `t` function in this example creates a pict containing the given text using the default font and style.

Executing the above module pops up a slide-presentation window. Type Alt-q (or Meta-q) to end the slides. Here are more controls:

Alt-q, Meta-q, or Cmd-q	: end slide show
Esc	: if confirmed, end show
Right arrow, Space, f, n, or click	: next slide
Left arrow, Backspace, Delete, or b	: previous slide
g	: last slide
l	: first slide
s	: next slide with a different title/name
a	: previous slide starting different title/name
Alt-g, Cmd-g, or Meta-g	: select a slide
Alt-p, Cmd-p, or Meta-p	: show/hide slide number

Alt-c, Cmd-c, or Meta-c	: show/hide commentary
Alt-d, Cmd-d, or Meta-d	: show/hide preview
Alt-m, Cmd-m, or Meta-m	: show/hide mouse cursor
Alt-l, Cmd-l, or Meta-l	: show/hide “spotlight”
Shift with arrow	: move window 1 pixel
Alt, Meta, or Cmd with arrow	: move window 10 pixels

The `slide` function accepts any number of arguments. Each argument is a pict to be centered on the slide. The picts are stacked vertically with `(current-gap-size)` separation between each pict, and the total result is centered (as long as there’s a gap of at least `(* 2 (current-gap-size))` between the title and content).

```
#lang slideshow

(slide
 #:title "How to Say Hello"
 (t "Hello World!")
 (t "Goodbye Drow!"))
```

Various functions format paragraphs and generate bulleted items for lists. For example, `item` creates a bulleted paragraph that spans (by default) the middle 2/3 of the slide:

```
#lang slideshow

(slide
 #:title "How to Say Hello"
 (item "If you want to create an example, you"
       "can always do something with" (bt "Hello World!"))
 (item "It's a bit silly, but a follow-up example"
       "could be" (bt "Goodbye Drow!")))
```

As the example illustrates, the `item` function accepts a mixture of strings and pict, and it formats them as a paragraph.

## 1.2 Staging Slides

The `slide` function creates a slide as a side effect. It can be put inside a function to abstract over a slide:

```
#lang slideshow

(define (slide-n n)
  (slide
   #:title "How to Generalize Slides"
```

```

      (item "This is slide number" (number->string n)))

(slide-n 1)
(slide-n 2)
(slide-n 3)

```

The `slide` function also has built-in support for some common multi-slide patterns. Each element argument to `slide` is usually a pict, but there are a few other possibilities:

- If an element is `'next`, then a slide is generated containing only the preceding elements, and then the elements are re-processed without the `'next`. Multiple `'next` elements generate multiple slides.
- If an element is `'alts`, then the next element must be a list of element lists. Each list up to the last one is appended to the elements before `'alts` and the resulting list of elements is processed. The last lists is appended to the preceding elements along with the remaining elements (after the list of lists) and the result is re-processed.
- A `'nothing` element is ignored (useful as a result of a branching expression).
- A `'next!` element is like `'next`, except that it is preserved when condensing (via the `--condense` flag).
- A `'alts~` element is like `'alts`, except that it is *not* preserved when condensing.
- A comment produced by `comment` is ignored, except when commentary is displayed.

Here's an example to illustrate how `'next` and `'alts` work:

```

#lang slideshow

(slide
 #:title "Example"
 (item "First step")
 'next
 (item "Second step")
 'next
 'alts
 (list (list (item "Tentative third step")
             'next
             (item "This isn't working... back up")))
       (list (item "Third step that works")))
 'next
 (item "Fourth step"))

```

### 1.3 Display Size and Fonts

Slideshow is configured for generating slides in 1024 by 768 pixel format. When the current display has a different size as Slideshow is started, the Slideshow display still occupies the entire screen, and pictures are scaled just before they are displayed. Thus, one picture unit reliably corresponds to a “pixel” that occupies 1/1024 by 1/768 of the screen.

The `text` form for generating text pictures takes into account any expected scaling for the display when measuring text. (All Slideshow text functions, such as `t` and `item` are built on `text`.) In particular, scaling the picture causes a different font size to be used for drawing the slide—rather than bitmap-scaling the original font—and changing the font size by a factor of  $k$  does not necessarily scale all text dimensions equally by a factor of  $k$ —because, for most devices, each character must have integer dimensions. Nevertheless, especially if you use the `current-expected-text-scale` parameter, Slideshow is usually able to produce good results when the slide is scaled.

More generally, different font sets on different platforms can change the way a slide is rendered. For example, the `tt` font on one platform might be slightly wider than on another, causing different line breaks, and so on. Beware.

Beware also of using bitmaps in slides when the presentation screen is not 1024 by 768. In that case, consider using `size-in-pixels` (with the caveat that the resulting picture will take up different amounts of the slide on different displays).

### 1.4 Command-line Options

```
(require slideshow/start)      package: slideshow-exe
```

The `slideshow` executable instantiates the `slideshow/start` module, which inspects the command line as reported by `current-command-line-arguments` to get another module to require for the slide content. The `slideshow/start` module also initializes variables like `printing?` and `condense?` based on flags supplied on the command line.

Thus, if the above example is in "multi-step.rkt", then the command

```
slideshow multi-step.rkt
```

runs the slides.

If the module given to `slideshow` has a `slideshow` submodule, then `slideshow/start` requires the `slideshow` submodule after requiring the module. If the module has no `slideshow` but has a `main` submodule, then the `main` submodule is required.

The `slideshow` executable accepts a number of command-line flags. Use the `--help` flag to obtain a list of other flags.

## 1.5 Printing

The `-p` or `--print` command-line flag causes Slideshow to print slides instead of showing them on the screen using the current platform's printing system. The `-P` or `--ps` generates PostScript directly, while `-D` or `--pdf` generates PDF directly.

PS-to-PDF converters vary on how well they handle landscape mode. Here's a Ghostscript command that converts slides reliably (when you replace `"src.ps"` and `"dest.pdf"` with your file names):

```
gs -q -dAutoRotatePages=/None -dSAFER -dNOPAUSE -dBATCH -
sOutputFile=dest.pdf -sDEVICE=pdfwrite -c .setpdfwrite -c "<</Ori-
entation 3>> setpagedevice" -f src.ps
```

## 2 Making Slides

```
(require slideshow/base)    package: slideshow-lib
```

The `slideshow/base` module, which is re-provided by `slideshow`, provides the functions for creating slides.

### 2.1 Primary Slide Functions

```
(slide [#:title title
       #:name name
       #:layout layout
       #:gap-size sep-gap-size
       #:inset inset
       #:timeout secs
       #:condense? condense?]
       element ...) → void?
title : (or/c #f string? pict?) = #f
name  : (or/c #f string?) = title
layout : (or/c 'auto 'center 'top 'tall) = 'auto
sep-gap-size : real? = (current-gap-size)
inset : slide-inset? = (make-slide-inset 0 0 0 0)
secs  : (or/c #f real?) = #f
condense? : any/c = (and timeout #t)
element : (flat-rec-contract elem/c
           (or/c pict?
                'next 'next! 'alts 'alts~ 'nothing
                comment?
                (listof (listof elem/c))))
```

Creates and registers a slide. See §1.2 “Staging Slides” for information about *elements*. Multiple *element* picts are separated by *sep-gap-size* vertical space.

When this function is first called in non-printing mode, then the viewer window is opened. Furthermore, each call to the function *yields*, so that the viewer window can be refreshed, and so the user can step through slides. If the user closes the slide window, then *slide* triggers an error unless *set-allow-new-slides-after-close!* was called with a true value before the window was closed.

If *title* is not *#f*, then a title is shown for the slide. The *name* is used in the slide-navigation dialog, and it defaults to *title*.

If *layout* is *'top*, then the content is top-aligned, with *(\* 2 sep-gap-size)* space between the title and the content. The *'tall* layout is similar, but with only *sep-gap-size*



space. The `'center` mode centers the content (ignoring space consumed by the title). The `'auto` mode is like `'center`, except when `title` is non-`#f` and when the space between the title and content would be less than `(* 2 sep-gap-size)`, in which case it behaves like `'top`.

The `inset` argument supplies an inset that makes the slide-viewing window smaller when showing the slide. See `make-slide-inset` for more information.

If `secs` argument for `#:timeout` is not `#f`, then the viewer automatically advances from this slide to the next after `secs` seconds, and manual advancing skips this slide.

If `condense?` is true, then in condense mode (as specified by the `-c` command-line flag), the slide is not created and registered.

```
(t str) → pict?  
  str : string?
```

The normal way to make plain text. Returns `(text str (current-main-font) (current-font-size))`.

```
(it str) → pict?  
  str : string?
```

The normal way to make italic text. Returns `(text str (cons 'italic (current-main-font)) (current-font-size))`.

```
(bt str) → pict?  
  str : string?
```

The normal way to make bold text. Returns `(text str (cons 'bold (current-main-font)) (current-font-size))`.

```
(bit str) → pict?  
  str : string?
```

Bold-italic text. Returns `(text str (list* 'bold 'italic (current-main-font)) (current-font-size))`.

```
(tt str) → pict?  
  str : string?
```

The normal way to make monospaced text. Returns `(text str `(bold . modern) (current-font-size))`.

```
(rt str) → pict?  
  str : string?
```

The normal way to make serif text. Returns `(text str 'roman (current-font-size))`.

```
(title? str) → pict?
  str : string?
```

Creates title text. Returns `((current-title?) str)`.

```
(para [:#width width
      #:align align
      #:fill? fill?
      #:decode? decode?]
      element ...) → pict?
width : real? = (current-para-width)
align : (or/c 'left 'center 'right) = 'left
fill? : any/c = #t
decode? : any/c = #t
element : (flat-rec-contract elem/c
          (or/c string? pict? (listof elem/c)))
```

Generates a paragraph pict that is no wider than *width* units, and that is exactly *width* units if *fill?* is true. If *fill?* is #f, then the result pict is as wide as the widest line.

Each list within *elements* is spliced into the sequence of string and pict elements. If *decode?* is true, then strings among the *elements* are decoded by performing the following substitutions: `---` → `--`, `--` → `=`, `~` → `'`, `'` → `'`, `'` → `'`. In addition, to better work with `at-exp` notation, if an *element* is `"\n"`, then it is dropped along with any spaces at the start of the next element.

Strings are split at spaces for word-wrapping to fit the page, and a space is added between elements. If a string element starts with one of the following punctuation marks (after decoding), however, no space is added before the string:

```
" = ' , . : ; ? ! ) ' ' ' "
```

The *align* argument specifies how to align lines within the paragraph.

See the spacing between lines is determined by the `current-line-sep` parameter.

```
(item [:#width width
      #:gap-size sep-gap-size
      #:bullet blt
      #:align align
      #:fill? fill?
      #:decode? decode?]
      element ...) → pict?
width : real? = (current-para-width)
```

```

sep-gap-size : real? = (current-gap-size)
blt : pict? = (scale bullet (/ sep-gap-size gap-size))
align : (or/c 'left 'center 'right) = 'left
fill? : any/c = #t
decode? : any/c = #t
element : (flat-rec-contract elem/c
           (or/c string? pict? (listof elem/c)))

```

Like `para`, but with `blt` followed by `(/ sep-gap-size 2)` space appended horizontally to the resulting paragraph, aligned with the top line. The paragraph width of `blt` plus `(/ sep-gap-size 2)` is subtracted from the maximum width of the paragraph.

```

(subitem [#:width width
         #:gap-size sep-gap-size
         #:bullet blt
         #:align align
         #:fill? fill?
         #:decode? decode?]
 element ...) → pict?
width : real? = (current-para-width)
sep-gap-size : real? = (current-gap-size)
blt : pict? = (scale o-bullet (/ sep-gap-size gap-size))
align : (or/c 'left 'center 'right) = 'left
fill? : any/c = #t
decode? : any/c = #t
element : (flat-rec-contract elem/c
           (or/c string? pict? (listof elem/c)))

```

Like `item`, but an additional `(* 2 sep-gap-size)` is subtracted from the paragraph width and added as space to the left of the pict. Also, `o-bullet` is the default bullet, instead of `bullet`.

```

(clickback pict thunk) → pict?
pict : pict?
thunk : (-> any)

```

Creates a pict that embeds the given one, and is the same size as the given pict, but that when clicked during a presentation calls `thunk`.

```

(interactive pict proc) → pict?
pict : pict?
proc : (frame% . -> . (-> any))

```

Creates a pict that embeds the given one, but that creates a floating frame at the pict's location on the screen during a presentation. After the floating frame is created (and before it is

shown), *proc* is applied to the frame. The result from *proc* must be a procedure that is called when the window is removed (because the slide changes, for example).

```
(size-in-pixels pict) → pict?  
pict : pict?
```

Scales *pict* so that it is displayed on the screen as (*pict-width pict*) pixels wide and (*pict-height pict*) pixels tall. The result is *pict* when using a 1024 by 768 display.

```
(pict->pre-render-pict pict) → pict?  
pict : pict?
```

Produces a pict that is like *pict*, but optimized for drawing on some platforms (currently Mac OS). This function may be useful to reduce drawing times for large bitmaps or complex drawings.

Added in version 1.1 of package *slideshow-lib*.

```
(make-outline name title subitems ...) → (symbol? . -> . void?)  
name : (or/c symbol? (listof symbol?))  
title : (or/c string? pict?)  
subitems : (or/c #f null?  
            (symbol? . -> . pict?))
```

Returns a function that takes a symbol and generates an outline slide.

The ... above applies to all three arguments together. Each trio of arguments defines a section for the outline:

- The section *name* is either a symbol or a list of symbols. When the outline function is called later to make an outline, the given symbol is compared to the section's symbol(s), and the section is marked as current if the symbol matches.
- The *title* is used as the displayed name of the section.
- The *subitems* are displayed when the section is active. It can be *#f* or *null* (for historical reasons) if no subitems are to be displayed. Otherwise, it should be a function that takes a symbol (the same one passed to the outline maker) and produces a pict.

```
(comment text ...) → comment?  
text : (or/c string? pict?)
```

Combines strings and pict to be used as a slide element for (usually hidden) commentary. Use the result as an argument to *slide*.

```
(comment? v) → boolean?  
v : any/c
```

Returns *#t* if *v* is a comment produced by *comment*.

## 2.2 Slide Registration

```
(slide? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a slide produced by `most-recent-slide` or `retract-most-recent-slide`.

```
(most-recent-slide) → slide?
```

Returns a slide structure that be supplied `re-slide` to make a copy of the slide or `slide->pict` to re-extract the entire slide as a pict.

```
(retract-most-recent-slide) → slide?
```

Cancels the most recently created slide, and also returns a slide structure that be supplied to `re-slide` to restore the slide (usually in a later position).

```
(re-slide slide [pict]) → void?  
slide : slide?  
pict : pict? = (blank)
```

Re-inserts a slide, `lt-superimpose`ing the given additional `pict`.

```
(slide->pict slide) → pict?  
slide : slide?
```

Converts a complete slide to a `pict`. The bounding box of the result corresponds to the slide within its margins.

## 2.3 Viewer Control

```
(start-at-recent-slide) → void?
```

Sets the starting slide for the talk to the most recently created slide. If this function is used multiple times, the last use overrides the earlier uses.

```
(enable-click-advance! on?) → void?  
on? : any/c
```

Enables or disables slide advance as a result of a mouse click.

```
(set-use-background-frame! on?) → void?  
on? : any/c
```

Enables or disables the creation of a background frame, which is typically useful only when `make-slide-inset` is used are active. The last enable/disable before the first slide registration takes effect once and for all.

```
(set-page-numbers-visible! on?) → void?  
on? : any/c
```

Determines whether slide numbers are initially visible in the viewer.

```
(current-page-number-font) → (is-a?/c font%)  
(current-page-number-font font) → void?  
font : (is-a?/c font%)
```

Parameter that determines the font used to draw the page number (if visible).

```
(current-page-number-color) → (or/c string? (is-a?/c color%))  
(current-page-number-color color) → void?  
color : (or/c string? (is-a?/c color%))
```

Parameter that determines the color used to draw the page number (if visible).

```
(current-page-number-adjust) → (-> number? string? string?)  
(current-page-number-adjust proc) → void?  
proc : (-> number? string? string?)
```

Parameter that controls the precise text that appears to indicate the page numbers (if visible). The input to the function is the default string and the slide number, and the result is what is drawn in the bottom right corner. The default parameter value just returns its first argument.

```
(set-spotlight-style! [#:size size  
                      #:color color]) → void?  
size : (or/c #f (>=/c 0)) = #f  
color : (or/c #f string? (is-a?/c color%)) = #f
```

Adjusts the size and color of the “spotlight,” which can be enabled in Slideshow as an alternative to the mouse. Note that the color normally should have alpha value less than 1 (to make it partially transparent). If `size` or `color` is `#f`, the corresponding setting is unchanged.

```
(set-allow-new-slides-after-close! on?) → void?  
on? : any/c
```

Sets whether new slides are allowed after the Slideshow window is closed by the user. By default, an attempt to register a new slide via `slide` after the window is closed triggers an error. Calling this function with `#t` enables new slides to start a new slideshow.

Added in version 1.3 of package `slideshow-lib`.

## 2.4 Constants and Layout Variables

`gap-size` : 24

A width commonly used for layout.

```
(current-gap-size) → real?  
(current-gap-size sep-gap-size) → void?  
  sep-gap-size : real?
```

A parameter whose value is a width used for the separation between items by `slide`, the size and spacing of a bullet for `item`, the space between a slide title and content in `'tall` mode, etc. The default value is `gap-size`.

`bullet` : pict?

A filled bullet used by default by `item`.

It is either `(t "•")`, if that character is available in the font that `t` uses, or it uses an implementation similar to `o-bullet`, but not hollow (using `disk`, not `circle`).

`o-bullet` : pict?

A hollow bullet used by default by `subitem`.

It's implementation is:

```
(baseless  
  (cc-superimpose  
    (circle (/ gap-size 2))  
    (blank 0 gap-size)))
```

`client-w`

Produces the width of the display area, minus margins. The result of the form changes if the margin is adjusted via `set-margin!`.

| `client-h`

Produces the height of the display area, minus margins, but including the title area). The result of the form changes if the margin is adjusted via `set-margin!`.

| `full-page`

Produces an empty pict that is the same size as the client area, which is like `(blank client-w client-h)`.

| `titleless-page`

Produces an empty pict that is the same size as the client area minus the title area in 'top layout mode, which is like `(blank client-w (- client-h title-h (* 2 gap-size)))`.

| `margin`

Produces a number that corresponds to the current margin, which surrounds every side of the slide. The client area for a slide corresponds to the display area (which is always 1024 by 768) minus this margin on each side. The default margin is 20.

The margin can be adjusted via `set-margin!`.

| `title-h`

Produces a number that corresponds to the height of a title created by `titlet`.

If `titlet` is changed via the `current-titlet` parameter, the title height should be updated via `set-title-h!`.

| `printing? : boolean?`

The value is `#t` if slides are being generated for printed output, `#f` for normal on-screen display. Printing mode is normally triggered via the `--print` or `--ps` command-line flag.

| `condense? : boolean?`

The value is `#t` if slides are being generated in condensed mode, `#f` for normal mode. Condensed mode is normally triggered via the `--condense` command-line flag.



## 2.5 Configuration

```
(current-font-size) → exact-nonnegative-integer?  
(current-font-size n) → void?  
  n : exact-nonnegative-integer?
```

Parameter that determines the font size used by `t`, `para`, etc. The default size is 32.

```
(current-main-font) → text-style/c  
(current-main-font style) → void?  
  style : text-style/c
```

Parameter that determines the font size used by `t`, `para`, etc. The default is platform-specific; possible initial values include 'swiss', "Verdana", and "Gill Sans".

```
(current-line-sep) → exact-nonnegative-integer?  
(current-line-sep n) → void?  
  n : exact-nonnegative-integer?
```

Parameter that controls the amount of space used between lines by `para`, `item`, and `subitem`.

```
(current-para-width) → exact-nonnegative-integer?  
(current-para-width n) → void?  
  n : exact-nonnegative-integer?
```

Parameter that controls the width of a pict created by `para`, `item`, and `subitem`.

```
(current-title-color) → (or/c string? (is-a?/c color%))  
(current-title-color color) → void?  
  color : (or/c string? (is-a?/c color%))
```

Parameter used by the default `current-titlet` to colorize the title. The default is "black".

```
(current-slide-assembler) → ((or/c string? #f)  
                               exact-nonnegative-integer?  
                               pict?  
                               . -> .  
                               pict?)  
(current-slide-assembler proc) → void?  
  proc : ((or/c string? #f)  
          exact-nonnegative-integer?  
          pict?  
          . -> .  
          pict?)
```

Parameter whose value is a function for assembling slide content into a single pict; the assembling function takes a string for the title (or #f), a separation for the title (if any) and pict, and a pict for the slide content (not counting the title).

The result is of the assembler is `ct-superimposed` with the client area, but the result pict might draw outside the client region to paint the screen margins, too.

The default assembler uses `titlet` to turn a title string (if any) to a pict. See also `current-titlet` and `set-title-h!`.

The slide assembler is *not* responsible for adding page numbers to the slide; that job belongs to the viewer. See also `current-page-number-font`, `current-page-number-color`, and `set-page-numbers-visible!`.

```
(current-titlet) → (string? . -> . pict?)
(current-titlet proc) → void?
  proc : (string? . -> . pict?)
```

Parameter to configure `titlet`. The default is

```
(lambda (s)
  (colorize (text s (current-main-font) 40)
            (current-title-color)))
```

If this parameter is changed such that the result is a different height, then `set-title-h!` should be called to update the value produced by `title-h`, `titleless-page`, etc.

```
(set-margin! amt) → void?
  amt : real?
```

Changes the margin that surrounds the client area. See also `margin`.

```
(set-title-h! amt) → void?
  amt : real?
```

Changes the expected height of a title, which adjusts `title-h`, `client-h`, `full-page`, and `titleless-page`.

```
(make-slide-inset left-inset
                  top-inset
                  right-inset
                  bottom-inset) → slide-inset?
left-inset : exact-nonnegative-integer?
top-inset  : exact-nonnegative-integer?
right-inset : exact-nonnegative-integer?
bottom-inset : exact-nonnegative-integer?
```

Creates a slide inset, which describes a number of pixels to inset the viewer for a slide on each side.

```
(slide-inset? v) → boolean?  
  v : any/c
```

Returns `#t` if `v` is a slide inset created by `make-slide-inset`, `#f` otherwise.

```
(commentary-on-slide-font-size) → exact-positive-integer?  
(commentary-on-slide-font-size size) → void?  
  size : exact-positive-integer?
```

The font size used for commentary when passing

```
--commentary-on-slide
```

on the command-line.

## 2.6 Pict-Staging Helper

```
(require slideshow/step)      package: slideshow-lib
```

The `slideshow/step` library provides syntax for breaking a complex slide into steps that are more complex than can be handled with `'next` and `'alts` in a `slide` sequence.

```
(with-steps (id ...) body ...)
```

Evaluates the `bodys` once for each `id`, skipping an `id` if its name ends with `~` and `condense?` is true. The results of the last `body` for each iteration are collected into a list, which is the result of the `with-steps` form.

Within the `bodys`, several keywords are bound non-hygenically (using the first `body`'s lexical context):

- `(only? id)` — returns `#t` during the `id` step (i.e., during the evaluation of the `bodys` for `id`), `#f` otherwise.
- `(vonly id)` — returns the identity function during the `id` step, `ghost` otherwise.
- `(only id then-expr)` returns the result of `then-expr` during the `id` step, `values` otherwise.
- `(only id then-expr else-expr)` returns the result of `then-expr` during the `id` step, the result of `else-expr` otherwise.
- `(before? id)` — returns `#t` before the `id` step, `#f` starting for the `id` and afterward.

- `(vbefore id)`, `(before id then-expr)`, or `(before id then-expr else-expr)` — analogous to `vonly` and `only`.
- `(after? id)` — returns `#t` after the `id` step, `#f` through the `id` step.
- `(vafter id)`, `(after id then-expr)`, or `(after id then-expr else-expr)` — analogous to `vonly` and `only`.
- `(between? a-id b-id)` — returns `#t` starting from the `a-id` step through the `b-id` step, `#f` otherwise.
- `(vbetween a-id b-id)`, `(between a-id b-id then-expr)`, or `(between a-id b-id then-expr else-expr)` — analogous to `vonly` and `only`.
- `(between-excel? a-id b-id)` — returns `#t` starting from the `a-id` step through steps before the `b-id` step, `#f` for the `b-id` step and afterward.
- `(vbetween-excl a-id b-id)`, `(between-excl a-id b-id then-expr)`, or `(between-excl a-id b-id then-expr else-expr)` — analogous to `vonly` and `only`.

▮ `(with-steps~ (id ...) body ...)`

Like `with-steps`, but when `condense?` is true, then `expr` is evaluated only for the last `id` (independent of whether the name of the last `id` name ends in `~`).

## 2.7 Text Formatting Helpers

`(require slideshow/text)`      package: `slideshow-lib`

This module provides conveniences functions for formatting text.

▮ `(with-size size expr)`

Sets `current-font-size` to `size` while running `expr`.

Added in version 1.2 of package `slideshow-lib`.

▮ `(with-scale scale expr)`

Multiplies `current-font-size` by `scale` while running `expr`.

Added in version 1.2 of package `slideshow-lib`.

▮ `(big text)`  
▮ `(small text)`

Scale `current-font-size` by  $3/2$  or  $2/3$ , respectively, while running `text`.

Added in version 1.2 of package `slideshow-lib`.

```
(with-font font expr)
```

Sets `current-main-font` to `font` while running `expr`.

Added in version 1.2 of package `slideshow-lib`.

```
(with-style style expr)
```

Adds `style` to `current-main-font` (via `cons`) while running `expr`.

Added in version 1.2 of package `slideshow-lib`.

```
(bold text)
(italic text)
(subscript text)
(superscript text)
(caps text)
```

Adds the attributes for bold, italic, superscript, subscript, or small caps text, respectively, to `current-main-font` while running `text`.

Added in version 1.2 of package `slideshow-lib`.

```
(blank-line) → pict?
```

Adds a blank line of the current font size's height.

Added in version 1.2 of package `slideshow-lib`.

## 2.8 Slides to Picts

```
(require slideshow/slides-to-picts)
package: slideshow-lib
```

```
(get-slides-as-picts path
                    width
                    height
                    condense?
                    [stop-after]) → (listof pict?)
path : path-string?
```

```
width : real?  
height : real?  
condense? : any/c  
stop-after : (or/c #f exact-nonnegative-integer?) = #f
```

Executes the Slideshow program indicated by *path* in a fresh namespace, and returns a list of *picts* for the slides. Each *pict* has the given *width* and *height*, and *condense?* determines whether the Slideshow program is executed in condense mode.

If *stop-after* is not *#f*, then the list is truncated after *stop-after* slides are converted to *picts*.

### 3 Typesetting Racket Code in Slideshow

```
(require slideshow/code)    package: slideshow-lib
```

The `slideshow/code` library provides all of the exports of `pict/code` and also initializes `get-current-code-font-size` to `current-font-size`.

## 4 Animations

```
(require slideshow/play)      package: slideshow-lib
```

The `slideshow/play` module provides tools for generating animations as multiple, automatically advanced slides.

Many of the tools are based on a function that takes a number between `0.0` and `1.0` inclusive and produces a pict. The pict produced for the input `0.0` is the starting image of the animation, and the pict produced for `1.0` is the ending image, while intermediate values produced intermediate images. For example,

```
(lambda (n)
  (cellophane (t "Hello") n))
```

corresponds to an animation that fades in the word “Hello.”

### 4.1 Generating Animated Slides

```
(play gen
  [#:steps steps
   #:delay delay-secs
   #:skip-first? skip-first?
   #:title title
   #:name name
   #:comment comment
   #:layout layout]) → void?
gen : ((real-in 0.0 1.0) . -> . pict?)
steps : exact-positive-integer? = 10
delay-secs : real? = 0.05
skip-first? : any/c = #f
title : (or/c string? pict? #f
         ((real-in 0.0 1.0) . -> . (or/c string? pict? #f)))
      = #f
name : (or/c string? #f
        ((real-in 0.0 1.0) . -> . (or/c string? #f)))
      = title
comment : (or/c comment? #f) = #f
layout : (or/c 'auto 'center 'top 'tall) = 'auto
```

Generates `steps+1` slides by calling `gen` on equally-spaced values from `0.0` (inclusive) to `1.0` (exclusive). Except for the first of the slides, each slide has a timeout of `delay-secs`, so that the next slide appears automatically.



Normally, `play` is called via `play-n`, which effectively calls `gen` on `1.0` without a timeout to complete the animation and stop the auto-advance of slides. The `play-n` function also manages with multi-step animations.

If `skip-first?` is `#f`, then one fewer slide is generated, because `gen` is not called on `0.0`.

The `title`, `name`, and `layout` arguments are passed on to `slide`, at least when `title` and/or `name` are not functions. When `title` or `name` is a function, the function is applied to the value used to produce the slide content, and the resulting title or name is passed on to `slide`.

The `comment` argument is used like a comment argument to `slide`.

In condensed mode (i.e., when `condense?` is `#t`), any slide that would be registered with a timeout is instead skipped.

```
(play-n gen*
  [#:steps steps
   #:delay delay-secs
   #:skip-first? skip-first?
   #:skip-last? skip-last?
   #:title title
   #:name name
   #:comments comment
   #:layout layout]) → void?
gen* : (and/c (unconstrained-domain-> pict?)
              (λ (x) (number? (procedure-arity x))))
steps : (list*of exact-positive-integer? = 10
         (or/c exact-positive-integer? '()))
delay-secs : real? = 0.05
skip-first? : any/c = #f
skip-last? : any/c = #f
title : (or/c string? pict? #f
         ((real-in 0.0 1.0) . -> . (or/c string? pict? #f)))
        = #f
name : (or/c string? #f
        ((real-in 0.0 1.0) . -> . (or/c string? #f)))
        = title
comment : (list*of comment? (or/c comment? #f '())) = #f
layout : (or/c 'auto 'center 'top 'tall) = 'auto
```

Generates a sequence of slides by calling `gen*` with, for each of its arguments, numbers from `0.0` to `1.0`. If `gen*` accepts `n` arguments, then result is a sequence of animations with a pause (i.e., not auto-advanced) between each of `n` segments.

If `gen*` accepts a single argument, then `play-n` is like `play`, except that `gen*` is also called with `1.0` to generate a slide with no timeout. If `gen*` accepts multiple arguments, then slides

are generated by calling `gen*` with the first argument varying from 0.0 to 1.0 while all other arguments are 0.0. Then, the first argument is held at 1.0 while the second argument varies from 0.0 to 1.0, and so on.

For example,

```
(play-n
  (lambda (n1 n2)
    (cellophane (t "Hello")
                 (* n1 (- 1.0 n2))))))
```

generates an animation to fade in the word “Hello,” and then pauses for a manual advance, and then fades “Hello” back out.

If `skip-first?` is `#t`, then the very first slide of the sequence is skipped. Similarly, if `skip-last?` is `#t`, then the last slide of the sequence is skipped.

The `steps` argument controls how many steps happen in each phase on the animation. If it is a number, then that number is used for each phase. If it is a pair of two numbers, then the first number is used for the first phase, and the second number is used for the rest of the phases. Similarly, if it is `(cons num_1 (cons num_2 num_3))`, `num_1` and `num_2` are used for the first two phases and `num_3` is used for the rest.

The elements of the `comment` argument are used like the `steps` argument, except passed as comments instead of used as step counts.

The `delay-msecs`, `title`, `name`, and `layout` arguments are passed on to `play` for each of the `n` segments of animation.

```
(animate-slide element ...)
→ ((() (listof (real-in 0.0 1.0)) . ->* . pict?)
    element : (flat-rec-contract elem/c
               (or/c pict? 'next 'alts
                 (listof (listof elem/c)))))
```

Accepts slide content similar to `slide` with `'next` and `'alts` and produces a procedure suitable for use with `play-n`. The result is similar to using `slide`, but with fades for `'next` and `'alts` transitions (to better fit the style, perhaps, of surrounding animations).

## 5 Legacy Libraries

```
(require slideshow/pict)      package: slideshow-lib
```

The `slideshow/pict` library re-exports `pict`.

```
(require slideshow/code-pict) package: slideshow-lib
```

The `slideshow/code-pict` library re-exports `pict/code`.

## Bibliography

- [Findler06] Robert Bruce Findler and Matthew Flatt, “Slideshow: Functional Presentations,” *Journal of Functional Programming*, 16(4-5), pp. 583–619, 2006. <http://www.cs.utah.edu/plt/publications/jfp05-ff.pdf>

## Index

[animate-slide](#), 26  
Animations, 24  
[big](#), 20  
[bit](#), 9  
[blank-line](#), 21  
[bold](#), 21  
[bt](#), 9  
[bullet](#), 15  
[caps](#), 21  
[clickback](#), 11  
[client-h](#), 16  
[client-w](#), 15  
Command-line Options, 6  
[comment](#), 12  
[comment?](#), 12  
[commentary-on-slide-font-size](#), 19  
[condense?](#), 16  
Configuration, 17  
Constants and Layout Variables, 15  
Creating Slide Presentations, 3  
[current-font-size](#), 17  
[current-gap-size](#), 15  
[current-line-sep](#), 17  
[current-main-font](#), 17  
[current-page-number-adjust](#), 14  
[current-page-number-color](#), 14  
[current-page-number-font](#), 14  
[current-para-width](#), 17  
[current-slide-assembler](#), 17  
[current-title-color](#), 17  
[current-titlet](#), 18  
Display Size and Fonts, 6  
[enable-click-advance!](#), 13  
[full-page](#), 16  
[gap-size](#), 15  
Generating Animated Slides, 24  
[get-slides-as-picts](#), 21  
[interactive](#), 11  
[it](#), 9  
[italic](#), 21  
[item](#), 10  
Legacy Libraries, 27  
[make-outline](#), 12  
[make-slide-inset](#), 18  
Making Slides, 8  
[margin](#), 16  
[most-recent-slide](#), 13  
[o-bullet](#), 15  
[para](#), 10  
[pict->pre-render-pict](#), 12  
Pict-Staging Helper, 19  
[play](#), 24  
[play-n](#), 25  
Primary Slide Functions, 8  
Printing, 7  
[printing?](#), 16  
[re-slide](#), 13  
[retract-most-recent-slide](#), 13  
[rt](#), 9  
[set-allow-new-slides-after-close!](#), 14  
[set-margin!](#), 18  
[set-page-numbers-visible!](#), 14  
[set-spotlight-style!](#), 14  
[set-title-h!](#), 18  
[set-use-background-frame!](#), 14  
[size-in-pixels](#), 12  
[slide](#), 8  
Slide Basics, 3  
Slide Registration, 13  
[slide->pict](#), 13  
[slide-inset?](#), 19  
[slide?](#), 13  
Slides to Picts, 21  
[slideshow](#), 1  
[slideshow/base](#), 8  
[slideshow/code](#), 23  
[slideshow/code-pict](#), 27  
[slideshow/pict](#), 27  
[slideshow/play](#), 24  
[slideshow/slides-to-picts](#), 21  
[slideshow/start](#), 6

`slideshow/step`, 19  
`slideshow/text`, 20  
Slideshow: Figure and Presentation Tools, 1  
`small`, 20  
Staging Slides, 4  
`start-at-recent-slide`, 13  
`subitem`, 11  
`subscript`, 21  
`superscript`, 21  
`t`, 9  
Text Formatting Helpers, 20  
`title-h`, 16  
`titleless-page`, 16  
`titlet`, 10  
`tt`, 9  
Typesetting Racket Code in Slideshow, 23  
Viewer Control, 13  
`with-font`, 21  
`with-scale`, 20  
`with-size`, 20  
`with-steps`, 19  
`with-steps~`, 20  
`with-style`, 21