

Package ‘iSEE’

October 16, 2025

Title Interactive SummarizedExperiment Explorer

Version 2.20.0

Date 2025-04-14

Description Create an interactive Shiny-based graphical user interface for exploring data stored in SummarizedExperiment objects, including row- and column-level metadata. The interface supports transmission of selections between plots and tables, code tracking, interactive tours, interactive or programmatic initialization, preservation of app state, and extensibility to new panel types via S4 classes. Special attention is given to single-cell data in a SingleCellExperiment object with visualization of dimensionality reduction results.

Depends SummarizedExperiment, SingleCellExperiment

Imports methods, BiocGenerics, S4Vectors, utils, stats, shiny, shinydashboard, shinyAce, shinyjs, DT, rintrojs, ggplot2, ggrepel, colourpicker, igraph, vipor, mgcv, graphics, grDevices, viridisLite, shinyWidgets, listviewer, ComplexHeatmap, circlize, grid

Suggests testthat, covr, BiocStyle, knitr, rmarkdown, scRNAseq, TENxPBMCDData, scater, DelayedArray, HDF5Array, RColorBrewer, viridis, htmltools, GenomicRanges

URL <https://isee.github.io/iSEE/>

BugReports <https://github.com/iSEE/iSEE/issues>

biocViews CellBasedAssays, Clustering, DimensionReduction, FeatureExtraction, GeneExpression, GUI, ImmunoOncology, ShinyApps, SingleCell, Transcription, Transcriptomics, Visualization

License MIT + file LICENSE

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.2

git_url <https://git.bioconductor.org/packages/iSEE>

git_branch RELEASE_3_21

git_last_commit fb79638

git_last_commit_date 2025-04-15

Repository Bioconductor 3.21

Date/Publication 2025-10-15

Author Kevin Rue-Albrecht [aut, cre] (ORCID:

<<https://orcid.org/0000-0003-3899-3872>>),

Federico Marini [aut] (ORCID: <<https://orcid.org/0000-0003-3252-7758>>),

Charlotte Soneson [aut] (ORCID:

<<https://orcid.org/0000-0003-3833-2169>>),

Aaron Lun [aut] (ORCID: <<https://orcid.org/0000-0002-3564-4813>>)

Maintainer Kevin Rue-Albrecht <kevinrue67@gmail.com>

Contents

.addCustomLabelsCommands	4
.addLabelCentersCommands	4
.addMultiSelectionPlotCommands	5
.addTourStep	6
.buildLabs	7
.conditionalOnRadio	8
.createCustomDimnamesModalObservers	9
.createUnprotectedParameterObservers	10
.extractAssaySubmatrix	11
.fullName	13
.panelColor	13
.processMultiSelections	15
.removeInvalidChoices	16
.replaceMissingWithFirst	16
.requestUpdate	17
.retrieveOutput	18
.setCachedCommonInfo	19
aes-utils	20
cache-utils	21
checkColormapCompatibility	22
class-utils	23
cleanDataset	24
collapseBox	25
ColumnDataPlot-class	27
ColumnDataTable-class	29
ColumnDotPlot-class	31
columnSelectionColorMap	33
ColumnTable-class	34
ComplexHeatmapPlot-class	35
constants	39
createCustomPanels	40

createLandingPage	42
defaultTour	45
documentation-generics	45
DotPlot-class	46
ExperimentColorMap-class	51
FeatureAssayPlot-class	54
filterDTColumn	57
hidden-inputs	59
interface-generics	60
interface-wrappers	61
iSEE	62
iSEE-pkg	65
iSEEOptions	66
jitterSquarePoints	67
lassoPoints	69
manage_commands	70
metadata-plot-generics	71
multi-select-generics	71
multiSelectionToFactor	74
observer-generics	74
output-generics	76
Panel-class	78
panelDefaults	81
plot-generics	83
plot-utils	87
ReducedDimensionPlot-class	87
registerAppOptions	89
RowDataPlot-class	92
RowDataTable-class	94
RowDotPlot-class	95
RowTable-class	98
SampleAssayPlot-class	99
setup-generics	102
single-select-generics	103
specific-tours	104
subsetPointsByGrid	106
synchronizeAssays	107
Table-class	109
table-generics	111
track-utils	112
validate-utils	113
visual-parameters-generics	114

`.addCustomLabelsCommands`*Add custom label plotting commands*

Description

Add `ggplot` instructions to add custom labels to specified points in a `DotPlot`. This is a utility function that is intended for use in `.generateDotPlot`.

Usage

```
.addCustomLabelsCommands(x, commands, plot_type)
```

Arguments

<code>x</code>	An instance of a <code>DotPlot</code> class.
<code>commands</code>	A character vector representing the sequence of commands to create the <code>ggplot</code> object.
<code>plot_type</code>	String specifying the type of plot, e.g., "scatter", "square", "violin".

Value

A character vector containing commands plus any additional commands required to generate the labels.

Author(s)

Kevin Rue-Albrecht, Aaron Lun

`.addLabelCentersCommands`*Add centered label plotting commands*

Description

Add `ggplot` instructions to label the center of each group on a scatter plot. This is a utility function that is intended for use in `.generateDotPlot`.

Usage

```
.addLabelCentersCommands(x, commands)
```

Arguments

<code>x</code>	An instance of a DotPlot class.
<code>commands</code>	A character vector representing the sequence of commands to create the ggplot object.

Value

A character vector containing `commands` plus any additional commands required to generate the labels.

Author(s)

Aaron Lun

`.addMultiSelectionPlotCommands`*Add multiple selection plotting commands*

Description

Add [ggplot](#) instructions to create brushes and lassos for both saved and active multiple selections in a [DotPlot](#) panel.

Usage

```
.addMultiSelectionPlotCommands(x, envir, commands, flip = FALSE)
```

Arguments

<code>x</code>	An instance of a DotPlot class.
<code>envir</code>	The environment in which the ggplot commands are to be evaluated.
<code>commands</code>	A character vector representing the sequence of commands to create the ggplot object.
<code>flip</code>	A logical scalar indicating whether the x- and y-axes are flipped, only relevant to horizontal violin plots.

Details

This is a utility function that is intended for use in [.generateDotPlot](#). It will modify `envir` by adding `all_active` and `all_saved` variables, so developers should not use these names for their own variables in `envir`.

If no self-selection structures exist in `x`, `commands` is returned directly without modification.

Value

A character vector containing `commands` plus any additional commands required to draw the self selections.

Author(s)

Aaron Lun

<code>.addTourStep</code>	<i>Add a step to the tour</i>
---------------------------	-------------------------------

Description

Utility to add a step to the panel-specific **rintrojs** tour, generating the element tag automatically.

Usage

```
.addTourStep(x, field, text, is_selectize = FALSE)
```

Arguments

<code>x</code>	A Panel object to be toured.
<code>field</code>	String containing the name of the slot of <code>x</code> , itself corresponding to an interface element to highlight.
<code>text</code>	String containing the text to show in the corresponding step of the tour.
<code>is_selectize</code>	Logical scalar indicating whether <code>field</code> corresponds to a selectize element.

Value

Character vector of length two. The first entry contains the element tag to identify the interface element to highlight, while the second entry contains the text.

Alternatively, NULL may be returned if `.hideInterface(x, field)` indicates that the corresponding interface element has been hidden.

Author(s)

Aaron Lun

*.buildLabs**Generate ggplot title and label instructions*

Description

Generate ggplot title and label instructions

Usage

```
.buildLabs(  
  x = NULL,  
  y = NULL,  
  color = NULL,  
  shape = NULL,  
  size = NULL,  
  fill = NULL,  
  group = NULL,  
  title = NULL,  
  subtitle = NULL  
)
```

Arguments

x	The character label for the horizontal axis.
y	x The character label for the vertical axis.
color	The character title for the color scale legend.
shape	The character title for the point shape legend.
size	The character title for the point size legend.
fill	The character title for the color fill legend.
group	The character title for the group legend.
title	The character title for the plot title.
subtitle	The character title for the plot subtitle

Details

If any argument is NULL, the corresponding label is not set.

Value

Title and label instructions for `ggplot` as a character value.

Author(s)

Kevin Rue-Albrecht

Examples

```
cat(.buildLabs(y = "Title for Y axis", color = "Color label"))
```

.conditionalOnRadio Conditional elements on radio or checkbox selection

Description

Creates a conditional UI element that appears when the user picks a certain choice in a radio button, single checkbox or checkbox group interface element.

Usage

```
.conditionalOnRadio(id, choice, ...)

.conditionalOnCheckSolo(id, on_select = TRUE, ...)

.conditionalOnCheckGroup(id, choice, ...)
```

Arguments

<code>id</code>	String containing the ID of the UI element controlling the relevant choice.
<code>choice</code>	String containing the choice for the radio button or checkbox group on which to show the conditional element(s).
<code>...</code>	UI elements to show conditionally.
<code>on_select</code>	Logical scalar specifying whether the conditional element should be shown upon selection in a check box, or upon de-selection (if FALSE).

Details

These functions are just wrappers around [conditionalPanel](#), with the added value coming from the pre-written conditional expressions in Javascript. They are useful for hiding elements that are only relevant when the right radio button or checkbox is selected. This means that we avoid cluttering the UI with options that are not immediately useful to the user.

Value

A HTML object containing interface elements in `...` that only appear when the relevant condition is satisfied.

Author(s)

Aaron Lun

See Also

[conditionalPanel](#), which is used under the hood.

`.createCustomDimnamesModalObservers`*Create observers for a modal for custom dimnames*

Description

Create observers to launch a modal where users can input a list of custom row or column names. These observers register input changes in the app's memory and request an update to the affected panel.

Usage

```
.createCustomDimnamesModalObservers(  
  plot_name,  
  slot_name,  
  button_name,  
  se,  
  input,  
  session,  
  pObjects,  
  rObjects,  
  source_type  
)
```

Arguments

<code>plot_name</code>	String containing the name of the current panel.
<code>slot_name</code>	String specifying the slot of containing the names of the custom features. This will be modified by user interactions with the modal.
<code>button_name</code>	String containing the name of the button in the panel UI that launches the modal.
<code>se</code>	A SummarizedExperiment object after running <code>.cacheCommonInfo</code> .
<code>input</code>	The Shiny input object from the server function.
<code>session</code>	The Shiny session object from the server function.
<code>pObjects</code>	An environment containing global parameters generated in the iSEE app.
<code>rObjects</code>	A reactive list of values generated in the iSEE app.
<code>source_type</code>	String specifying the type of the panel that is source of the selection, either "row" or "column".

Details

This should be called in `.createObservers` for the target panel. It assumes that a button element with the suffix `button_name` is available in the UI (i.e., the full name is created by concatenated `plot_name` with `button_name`).

The modal UI provides options to sort the dimnames, validate them, clear the current text and import a selection from a specified row transmitter. These are all transient until “Apply” is clicked, at which point the app’s memory is modified and an update is requested.

The custom names are stored in the `slot_name` as a single string with names separated by newlines. Hashes are treated as comments and any content after a hash is ignored when interpreting the names. Any leading and trailing whitespace is also ignored during interpretation.

Value

Observers are set up to launch the modal and monitor its UI elements. A NULL is invisibly returned.

Author(s)

Kevin Rue-Albrecht

```
.createUnprotectedParameterObservers
```

Define parameter observers

Description

Define a series of observers to track “protected” or “unprotected” parameters for a given panel. These will register input changes to each specified parameter in the app’s memory and request an update to the output of the affected panel.

Usage

```
.createUnprotectedParameterObservers(
  panel_name,
  fields,
  input,
  pObjects,
  rObjects,
  ignoreInit = TRUE,
  ignoreNULL = TRUE
)

.createProtectedParameterObservers(
  panel_name,
  fields,
  input,
  pObjects,
  rObjects,
  ignoreInit = TRUE,
  ignoreNULL = TRUE
)
```

Arguments

<code>panel_name</code>	String containing the name of the panel.
<code>fields</code>	Character vector of names of parameters for which to set up observers.
<code>input</code>	The Shiny input object from the server function.
<code>pObjects</code>	An environment containing global parameters generated in the iSEE app.
<code>rObjects</code>	A reactive list of values generated in the iSEE app.
<code>ignoreInit, ignoreNULL</code>	Further arguments to pass to observeEvent .

Details

A protected parameter is one that breaks existing multiple selections, e.g., by changing the actual data being plotted. Alterations to protected parameters will clear all active and saved selections in the panel, as those existing selections are assumed to not make any sense in the context of the modified output of that panel.

By comparison, an unprotected parameter only changes the aesthetics and will not clear existing selections.

Value

Observers are set up to monitor the UI elements that can change the protected and non-fundamental parameters. A NULL is invisibly returned.

Author(s)

Aaron Lun

See Also

[.requestUpdate](#) and [.requestCleanUpdate](#), used to trigger updates to the panel output.

`.extractAssaySubmatrix`*Extract assay submatrix*

Description

Extract an assay submatrix based on the multiple row/column selection and any custom specifications from [.createCustomDimnamesModalObservers](#).

Usage

```
.extractAssaySubmatrix(
  x,
  se,
  envir,
  use_custom_row_slot,
  custom_row_text_slot,
  cap_row_selection_slot = NULL,
  as_matrix = TRUE
)
```

Arguments

<code>x</code>	A Panel instance that uses the row selection modal.
<code>se</code>	The current SummarizedExperiment object.
<code>envir</code>	The evaluation environment. This assumes that .processMultiSelections has already been run.
<code>use_custom_row_slot</code>	String specifying the name of the slot indicating whether to use custom rows.
<code>custom_row_text_slot</code>	String specifying the name of the slot holding the custom row names. This is expected to be of the same format as described in ?.createCustomDimnamesModalObservers .
<code>cap_row_selection_slot</code>	String specifying the name of the slot containing the cap on the number of selected rows.
<code>as_matrix</code>	Logical scalar indicating whether to coerce the submatrix into an ordinary matrix. If FALSE, the existing matrix representation is preserved.

Details

This is designed to extract a matrix of assay values for a subset of rows/columns of interest, most typically for a [ComplexHeatmapPlot](#). It assumes that the class of `x` contains a slot indicating whether custom rows should be used, plus a slot to hold the selected custom row names (usually from a modal, see [.createCustomDimnamesModalObservers](#)).

If a multiple row selection is present in `envir` and custom rows are *not* to be used, that selection is used to define the rows of the submatrix. All columns are returned in the submatrix unless a multiple column selection is present in `envir` and the `SelectEffect` in `x` is “Restrict”, in which case only the selected columns are returned.

Value

A character vector of commands to set up the assay submatrix. The submatrix itself is generated within `envir` as the `plot.data` variable.

Author(s)

Kevin Rue-Albrecht

<code>.fullName</code>	<i>Get panel names</i>
------------------------	------------------------

Description

Get panel names

Usage

`.fullName(x)`

`.getEncodedName(x)`

`.getFullName(x)`

Arguments

`x` An instance of a [Panel](#) class.

Details

The encoded name is used internally as the name of various fields in input, output and reactive lists.

The full name is what should be shown in the interface and visible to the end-user.

Value

For `.getEncodedName`, a string containing the encoded panel name of `x`.

For `.fullName`, a string containing the full (plain-English) name of the class.

For `.getFullName`, a string containing the full name of `x`.

Author(s)

Aaron Lun

<code>.panelColor</code>	<i>Get panel colors</i>
--------------------------	-------------------------

Description

Functions to get/set panel colors at the user and developer level. This determines the color of the panel header as well as (for [DotPlots](#)) the color and fill of the brush.

Usage

```
.panelColor(x)

.getPanelColor(x)
```

Arguments

x An instance of a [Panel](#) class.

Details

For developers: `.panelColor` is a method that should be subclassed for each [Panel](#) subclass. This determines the color theme for all instances of that class for use in, e.g., headers and box shadings. Developers should choose a color that is dark enough to serve as a background for white text. We recommend defining colors as hex color codes for full compatibility with both HTML elements and R plots.

For users: by default, `.getPanelColor` will return the default color of each panel as specified by the developer in `.panelColor`. However, users can override this by setting the `panel.color` global option to a named character vector of colors (see Examples). This can be used to customize the color scheme for any given call to [iSEE](#). The names of the vector should be set to the name of class to be overridden; if a class is not named here, its default color is used.

Value

A string containing the color assigned to the class of *x*.

Author(s)

Aaron Lun

Examples

```
rdp <- ReducedDimensionPlot()

# Default color, as specified by the developer:
.panelColor(rdp)

# Still the default color:
.getPanelColor(rdp)

# Overriding the default colors:
sce <- SingleCellExperiment(list(logcounts=matrix(rnorm(1000), ncol=100)))
reducedDim(sce, "PCA") <- matrix(runif(200), ncol=2)

sce <- registerAppOptions(sce, panel.color=c(ReducedDimensionPlot="#1e90ff"))
if (interactive()) {
  iSEE(sce, initial=list(rdp))
}
```

`.processMultiSelections`*Process multiple selections*

Description

Generate and execute commands to process multiple selections, creating variables in the evaluation environment with the identity of the selected rows or columns.

Usage

```
.processMultiSelections(x, all_memory, all_contents, envir)
```

Arguments

<code>x</code>	An instance of a Panel class.
<code>all_memory</code>	A named list of Panel instances containing parameters for the current app state.
<code>all_contents</code>	A named list of arbitrary contents with one entry per panel.
<code>envir</code>	The evaluation environment. This is assumed to already contain <code>se</code> , the SummarizedExperiment object for the current dataset.

Details

This function is primarily intended for use by developers of new panels. It should be called inside [.generateOutput](#) to easily process row/column multiple selections. Developers can check whether `row_selected` or `col_selected` exists in `envir` to determine whether any row or column selection was performed (and adjust the behavior of `.generateOutput` accordingly).

Value

`envir` is populated with one, none or both of `col_selected` and/or `row_selected`, depending on whether `x` is receiving a multiple selection on the rows and/or columns. The return value is the character vector of commands required to construct those variables.

Author(s)

Aaron Lun

See Also

[.generateOutput](#) and its related generic [.renderOutput](#), where this function should generally be used.

`.removeInvalidChoices` *Remove invalid values in multiple choices*

Description

Removes invalid values in a slot of a [Panel](#) object. This is usually called in `.refineParameters`.

Usage

```
.removeInvalidChoices(x, field, choices)
```

Arguments

<code>x</code>	An instance of a Panel class.
<code>field</code>	String containing the name of the relevant slot.
<code>choices</code>	Character vector of permissible values for this slot.

Value

`x` where the slot named `field` is replaced only with the values that exist in `choices`.

Author(s)

Kevin Rue-Albrecht

`.replaceMissingWithFirst`
Replace with first choice

Description

Replace an NA or invalid value in a slot of a [Panel](#) object with the first valid choice. This is usually called in `.refineParameters`.

Usage

```
.replaceMissingWithFirst(x, field, choices)
```

Arguments

<code>x</code>	An instance of a Panel class.
<code>field</code>	String containing the name of the relevant slot.
<code>choices</code>	Character vector of permissible values for this slot.

Value

x where the slot named `field` is replaced with `choices[1]` if its value was previously NA or did not exist in `choices`.

Author(s)

Aaron Lun

<code>.requestUpdate</code>	<i>Request Panel updates</i>
-----------------------------	------------------------------

Description

Request a re-rendering of the [Panel](#) output via reactive variables.

Usage

```
.requestUpdate(panel_name, rObjects)

.requestCleanUpdate(panel_name, pObjects, rObjects)

.requestActiveSelectionUpdate(
  panel_name,
  session,
  pObjects,
  rObjects,
  update_output = TRUE
)
```

Arguments

<code>panel_name</code>	String containing the panel name.
<code>rObjects</code>	A reactive list of values generated in the iSEE app.
<code>pObjects</code>	An environment containing global parameters generated in the iSEE app.
<code>session</code>	The Shiny session object from the server function.
<code>update_output</code>	A logical scalar indicating whether to call <code>.requestUpdate</code> as well.

Details

`.requestUpdate` should be used in various observers to request a re-rendering of the panel, usually in response to user-driven parameter changes in [.createObservers](#).

`.requestCleanUpdate` is used for changes to protected parameters that invalidate existing multiple selections, e.g., if the coordinates change in a [DotPlot](#), existing brushes and lassos are usually not applicable.

`.requestActiveSelectionUpdate` should be used in the observer expression that implements the panel's multiple selection mechanism.

Value

`.requestUpdate` will modify `rObjects` to request a re-rendering of the specified panel.

`.requestCleanUpdate` will also remove all active/saved selections in the chosen panel.

`.requestActiveSelectionUpdate` will modify `rObjects` to indicate that the active multiple selection for `panel_name` has changed. If `update_output=TRUE`, it will also request a re-rendering of the panel.

All functions will invisibly return `NULL`.

Author(s)

Aaron Lun

See Also

[.createProtectedParameterObservers](#), for examples where the update-requesting functions are used.

<code>.retrieveOutput</code>	<i>Retrieve the panel output</i>
------------------------------	----------------------------------

Description

Retrieve the results of a previous [.generateOutput](#) call on this panel.

Usage

```
.retrieveOutput(panel_name, se, pObjects, rObjects)
```

Arguments

<code>panel_name</code>	String containing the panel name.
<code>se</code>	A SummarizedExperiment object containing the current dataset.
<code>pObjects</code>	An environment containing global parameters generated in the iSEE app.
<code>rObjects</code>	A reactive list of values generated in the iSEE app.

Details

This function should be used in the rendering expression in [.renderOutput](#). It takes care of a number of house-keeping tasks required to satisfy [.renderOutput](#)'s requirements, e.g., responding to [.requestUpdate](#) modifications to `rObjects`, setting the commands and contents and varname in `pObjects`.

This function will attempt to retrieve the cached output of [.generateOutput](#) if it was used elsewhere in the app. After retrieval, the cached value is wiped to ensure that it does not go stale. If no cached value is found, [.generateOutput](#) is called directly.

Value

The output of running `.generateOutput` for the current panel. Several fields in `pObjects` are also modified as a side-effect.

Author(s)

Aaron Lun

See Also

`.renderOutput`, where this function should be called.

`.generateOutput`, which is called by this function.

`.setCachedCommonInfo` *Set and get cached commons*

Description

Get and set common cached information for each class. The setter should only ever be called in `.cacheCommonInfo`. The getter can be called anywhere but most usually in `.defineInterface`.

Usage

```
.setCachedCommonInfo(se, cls, ...)
```

```
.getCachedCommonInfo(se, cls)
```

Arguments

<code>se</code>	A <code>SummarizedExperiment</code> object containing the current dataset.
<code>cls</code>	String containing the name of the class for which this information is cached.
<code>...</code>	Any number of named R objects to cache.

Value

`.setCachedCommonInfo` returns `se` with `...` added to its `int_metadata`.

`.getCachedCommonInfo` retrieves the cached common information for class `cls`.

Author(s)

Aaron Lun

See Also

`?"cache-utils"`, for utilities to define some cached variables.

Examples

```
se <- SummarizedExperiment()
se <- .setCachedCommonInfo(se, "SomePanelClass",
  something=1, more_things=TRUE, something_else="A")
.getCachedCommonInfo(se, "SomePanelClass")
```

aes-utils

Generate ggplot aesthetic instructions

Description

Generate ggplot aesthetic instructions

Usage

```
.buildAes(
  x = TRUE,
  y = TRUE,
  color = FALSE,
  shape = FALSE,
  size = FALSE,
  fill = FALSE,
  group = FALSE,
  alt = NULL
)
```

Arguments

<code>x</code>	A logical that indicates whether to enable <code>x</code> in the aesthetic instructions (default: TRUE).
<code>y</code>	A logical that indicates whether to enable <code>y</code> in the aesthetic instructions (default: TRUE).
<code>color</code>	A logical that indicates whether to enable <code>color</code> in the aesthetic instructions (default: FALSE).
<code>shape</code>	A logical that indicates whether to enable <code>shape</code> in the aesthetic instructions (default: FALSE).
<code>size</code>	A logical that indicates whether to enable <code>size</code> in the aesthetic instructions (default: FALSE).
<code>fill</code>	A logical that indicates whether to enable <code>fill</code> in the aesthetic instructions (default: FALSE).
<code>group</code>	A logical that indicates whether to enable <code>group</code> in the aesthetic instructions (default: FALSE).
<code>alt</code>	Alternative aesthetics, supplied as a named character vector.

Value

Aesthetic instructions for [ggplot](#) as a character value.

Author(s)

Kevin Rue-Albrecht

Examples

```
.buildAes()
```

cache-utils	<i>Caching utilities</i>
-------------	--------------------------

Description

Utility functions to be used in a [.cacheCommonInfo](#) method, usually to identify names of elements of the [SummarizedExperiment](#) for later use in [.defineInterface](#) to populate the user interface.

Usage

```
.findAtomicFields(x)

.whichGroupable(x, max_levels = Inf)

.whichNumeric(x)

.isAssayNumeric(se, i)
```

Arguments

x	A data.frame or DataFrame , most typically the rowData or colData .
max_levels	Integer scalar specifying the maximum number unique values for x to be categorical.
se	The SummarizedExperiment object.
i	An integer scalar or string specifying the assay of interest in se.

Details

[.findAtomicFields](#) is necessary as many of the widgets used by [iSEE](#) (e.g., [ggplot](#), [datatable](#)) do not know how to handle more complex types being stored as columns. Similarly, [.whichNumeric](#) and [.whichGroupable](#) can be used to specify options for visualization modes that only make sense for continuous or discrete variables respectively (e.g., sizing, faceting).

Value

For `.findAtomicFields`, a character vector of names of columns in `x` containing atomic R types.

For `.whichNumeric`, an integer vector containing the indices of the numeric columns.

For `.whichGroupable`, an integer vector containing the indices of the categorical columns.

For `.isAssayNumeric`, a logical scalar indicating whether the specified assay is numeric.

Author(s)

Aaron Lun, Kevin Rue-Albrecht, Charlotte Soneson

Examples

```
x <- DataFrame(
  A = rnorm(10),
  B = sample(letters, 10),
  DataFrame = I(DataFrame(
    C = rnorm(10),
    D = sample(letters, 10)
  ))
)

.findAtomicFields(x)
.whichGroupable(x)
.whichNumeric(x)
```

checkColormapCompatibility

Check compatibility between ExperimentColorMap and Summarized-Experiment objects

Description

This function compares a pair of [ExperimentColorMap](#) and [SingleCellExperiment](#) objects, and examines whether all of the assays, colData, and rowData defined in the ExperimentColorMap object exist in the SingleCellExperiment object.

Usage

```
checkColormapCompatibility(ecm, se)
```

Arguments

ecm	An ExperimentColorMap .
se	A SingleCellExperiment .

Value

A character vector of incompatibility error messages, if any.

Author(s)

Kevin Rue-Albrecht

Examples

```
# Example colormaps ----

count_colors <- function(n){
  c("black", "brown", "red", "orange", "yellow")
}

qc_color_fun <- function(n){
  qc_colors <- c("forestgreen", "firebrick1")
  names(qc_colors) <- c("Y", "N")
  return(qc_colors)
}

ecm <- ExperimentColorMap(
  assays = list(
    tophat_counts = count_colors
  ),
  colData = list(
    passes_qc_checks_s = qc_color_fun
  )
)

# Example SingleCellExperiment ----

library(scRNAseq)
sce <- ReprocessedAllenData(assays="tophat_counts")

# Test for compatibility ----

checkColormapCompatibility(ecm, sce)
```

class-utils

*Set default slot values***Description**

A utility function to set slots to default values if their values are not provided to `initialize` methods.

Usage

```
.emptyDefault(args, field, default)
```

Arguments

args	A named list of arguments to pass to the <code>initialize</code> method for a given class.
field	String specifying the field to set.
default	The default value of the slot in <code>field</code> .

Details

A more natural approach would be to have the default values in the arguments of the `initialize` method. However, this would require us to hard-code the slot names in the function signature, which would break our current DRY model of only specifying the slot names once.

Value

args is returned with the named `field` set to `default` if it was previously absent.

Author(s)

Aaron Lun, Kevin Rue-Albrecht

Examples

```
showMethods("initialize", classes = "ReducedDimensionPlot", includeDefs = TRUE)
```

cleanDataset

Clean the dataset

Description

Clean the `SummarizedExperiment` by making sure that names of various fields are available and unique. Also transfer any information stored in `rowRanges` into `rowData`.

Usage

```
cleanDataset(se)

## S4 method for signature 'SummarizedExperiment'
cleanDataset(se)

## S4 method for signature 'SingleCellExperiment'
cleanDataset(se)
```

Arguments

se A [SummarizedExperiment](#) object or one of its subclasses.

Details

Various [Panels](#) assume that the row and column names of the input SummarizedExperiment are available and unique. This function enforces that, adding consecutive integer names if not available and calling [make.unique](#) if they are duplicated.

Various [Panels](#) further assume that the [assay](#), [rowData](#), [colData](#) names are unique; if this is not the case, [selectInput](#) behaves in unexpected (and incorrect) ways. This function enforces that as well by running them through [make.unique](#).

Finally, positional information in [rowRanges](#) is not accessible to [iSEE](#). This function moves this information into [rowData](#), prefixing the column names with [rowRanges_](#).

For [SingleCellExperiment](#) object, we enforce uniqueness in the [reducedDims](#).

All changes result in warnings as a “sensible” object is not expected to require any work.

Value

A cleaned version of `se`.

Author(s)

Aaron Lun, Charlotte Soneson

Examples

```
# Creating a very naughty SE.
se <- SummarizedExperiment(list(cbind(1:10, 2:11), cbind(2:11, 3:12)),
  colData=DataFrame(A=1:2, A=3:4, check.names=FALSE),
  rowData=DataFrame(B=1:10, B=1:10, check.names=FALSE))
se

cleanDataset(se)
```

collapseBox

A collapsible box

Description

A custom collapsible box with Shiny inputs upon collapse, more or less stolen from [shinyBS](#).

Usage

```
collapseBox(id, title, ..., open = FALSE, style = NULL)
```

Arguments

<code>id</code>	String specifying the identifier for this object, to use as a field of the Shiny input.
<code>title</code>	String specifying the title of the box for use in the UI.
<code>...</code>	Additional UI elements to show inside the box.
<code>open</code>	Logical scalar indicating whether this box should be open upon initialization.
<code>style</code>	String specifying the box style, defaults to "default".

Details

Collapsible boxes are used to hold parameters in the “parameter boxes” described in [.defineInterface](#). It is recommended to format the id as PANEL_SLOT where PANEL is the name of the panel associated with the box and SLOT is the name of the slot that specifies whether this box should be open or not at initialization. (See [Panel](#) for some examples with DataBoxOpen.)

Do not confuse these boxes with the shinydashboard::boxes, which are used to hold the plot and table panels. Adding to the nomenclature confusion is the fact that our collapsible boxes are implemented in Javascript using the Bootstrap “panel” classes, which in turn has nothing to do with our [Panel](#) classes.

Value

A HTML tag object containing a collapsible box.

Comments on shinyBS

We would have preferred to use bsCollapse from **shinyBS**. However, that package does not seem to be under active maintenance, and there are several aspects that make it difficult to use. Specifically, it does not seem to behave well with conditional elements inside the box, and it also does needs a Depends: relationship with **shinyBS**.

For these reasons, we created our own collapsible box, taking code from shinyBS where appropriate. The underlying Javascript code for this object is present in inst/www and is attached to the search path for Shiny resources upon loading **iSEE**.

Author(s)

Aaron Lun

See Also

shinyBS, from which the Javascript code was derived.

[.defineInterface](#), which should return a list of these collapsible boxes.

Examples

```
library(shiny)
collapseBox("SomePanelType1_ParamBoxOpen",
  title="Custom parameters",
  open=FALSE,
  selectInput("SomePanelType1_Thing",
    label="What thing?",
    choices=LETTERS, selected="A"
  )
)
```

ColumnDataPlot-class *The ColumnDataPlot panel*

Description

The ColumnDataPlot is a panel class for creating a [ColumnDotPlot](#) where the y-axis represents a variable from the [colData](#) of a [SummarizedExperiment](#) object. It provides slots and methods for specifying which variable to use on the y-axis (and, optionally, also the x-axis), as well as a method to create the data.frame in preparation for plotting.

Slot overview

The following slots control the column data information that is used:

- **YAxis**, a string specifying the column of the [colData](#) to show on the y-axis. If NA, defaults to the first valid field (see ?"[.refineParameters, ColumnDotPlot-method](#)").
- **XAxis**, string specifying what should be plotting on the x-axis. This can be any one of "None", "Column data" or "Column selection". Defaults to "None".
- **XAxisColumnData**, string specifying the column of the [colData](#) to show on the x-axis. If NA, defaults to the first valid field.

In addition, this class inherits all slots from its parent [ColumnDotPlot](#), [DotPlot](#) and [Panel](#) classes.

Constructor

`ColumnDataPlot(...)` creates an instance of a ColumnDataPlot class, where any slot and its value can be passed to ... as a named argument.

Supported methods

In the following code snippets, `x` is an instance of a [ColumnDataPlot](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- [.refineParameters\(x, se\)](#) returns `x` after replacing any NA value in **YAxis** or **XAxisColumnData** with the name of the first valid [colData](#) variable. This will also call the equivalent [ColumnDotPlot](#) method for further refinements to `x`. If no valid column metadata variables are available, NULL is returned instead.

For defining the interface:

- [.defineDataInterface\(x, se, select_info\)](#) returns a list of interface elements for manipulating all slots described above.
- [.panelColor\(x\)](#) will return the specified default color for this panel class.
- [.allowableXAxisChoices\(x, se\)](#) returns a character vector specifying the acceptable variables in [colData\(se\)](#) that can be used as choices for the x-axis. This consists of all variables with atomic values.

- `.allowableYAxisChoices(x, se)` returns a character vector specifying the acceptable variables in `colData(se)` that can be used as choices for the y-axis. This consists of all variables with atomic values.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots described above and in the parent classes. This will also call the equivalent `ColumnDotPlot` method.

For controlling selections:

- `.multiSelectionInvalidated(x)` returns TRUE if the x-axis uses multiple column selections, such that the point coordinates may change upon updates to upstream selections in transmitting panels. Otherwise, it dispatches to the `ColumnDotPlot` method.

For defining the panel name:

- `.fullName(x)` will return "Column data plot".

For creating the plot:

- `.generateDotPlotData(x, envir)` will create a data.frame of column metadata variables in `envir`. It will return the commands required to do so as well as a list of labels.

For documentation:

- `.definePanelTour(x)` returns an data.frame containing a panel-specific tour.

Subclass expectations

Subclasses do not have to provide any methods, as this is a concrete class.

Author(s)

Aaron Lun

See Also

`ColumnDotPlot`, for the immediate parent class.

Examples

```
#####
# For end-users #
#####

x <- ColumnDataPlot()
x[["XAxis"]]
x[["XAxis"]] <- "Column data"

#####
# For developers #
#####
```

```

library(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)

old_cd <- colData(sce)
colData(sce) <- NULL

# Spits out a NULL and a warning if there is nothing to plot.
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

# Replaces the default with something sensible.
colData(sce) <- old_cd
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

```

ColumnDataTable-class *The ColumnDataTable panel*

Description

The ColumnDataTable is a panel class for creating a [ColumnTable](#) where the value of the table is defined as the [colData](#) of the [SummarizedExperiment](#). It provides functionality to extract the [colData](#) to coerce it into an appropriate data.frame in preparation for rendering.

Slot overview

This class inherits all slots from its parent [ColumnTable](#) and [Table](#) classes.

Constructor

`ColumnDataTable(...)` creates an instance of a ColumnDataTable class, where any slot and its value can be passed to ... as a named argument.

Note that ColSearch should be a character vector of length equal to the total number of columns in the [colData](#), though only the entries for the atomic fields will actually be used.

Supported methods

In the following code snippets, x is an instance of a [ColumnDataTable](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.cacheCommonInfo(x)` adds a "ColumnDataTable" entry containing `valid.colData.names`, a character vector of names of atomic columns of the [colData](#). This will also call the equivalent [ColumnTable](#) method.

- `.refineParameters(x, se)` adjusts ColSearch to a character vector of length equal to the number of atomic fields in the `colData`. This will also call the equivalent `ColumnTable` method for further refinements to `x`.

For defining the interface:

- `.fullName(x)` will return "Column data table".
- `.panelColor(x)` will return the specified default color for this panel class.

For creating the output:

- `.generateTable(x, envir)` will modify `envir` to contain the relevant data.frame for display, while returning a character vector of commands required to produce that data.frame. Each row of the data.frame should correspond to a column of the SummarizedExperiment.

For documentation:

- `.definePanelTour(x)` returns an data.frame containing the steps of a panel-specific tour.

Unless explicitly specialized above, all methods from the parent class `Panel` are also available.

Author(s)

Aaron Lun

Examples

```
#####
# For end-users #
#####

x <- ColumnDataTable()
x[["Selected"]]
x[["Selected"]] <- "SOME_SAMPLE_NAME"

#####
# For developers #
#####

library(scater)
sce <- mockSCE()

# Search column refinement works as expected.
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)
```

ColumnDotPlot-class *The ColumnDotPlot virtual class*

Description

The ColumnDotPlot is a virtual class where each column in the [SummarizedExperiment](#) is represented by no more than one point (i.e., a “dot”) in a brushable [ggplot](#) plot. It provides slots and methods to extract [colData](#) fields to control the per-point aesthetics on the plot. This panel will transmit column identities in both its single and multiple selections, and it can receive multiple column selections but not multiple row selections.

Slot overview

The following slots control coloring of the points:

- `ColorByColumnData`, a string specifying the [colData](#) field for controlling point color, if `ColorBy="Column data"` (see the [Panel](#) class). Defaults to the first valid field (see `.cacheCommonInfo` below).
- `ColorByFeatureNameAssay`, a string specifying the assay of the [SummarizedExperiment](#) object containing values to use for coloring, if `ColorBy="Feature name"`. Defaults to "logcounts" in `getPanelDefault`, falling back to the name of the first valid assay (see `?".cacheCommonInfo, DotPlot-method"` for the definition of validity).
- `ColorBySampleNameColor`, a string specifying the color to use for coloring an individual sample on the plot, if `ColorBy="Sample name"`. Defaults to "red" in `getPanelDefault`.

The following slots control other metadata-related aesthetic aspects of the points:

- `ShapeByColumnData`, a string specifying the [colData](#) field for controlling point shape, if `ShapeBy="Column data"` (see the [Panel](#) class). The specified field should contain categorical values; defaults to the first such valid field.
- `SizeByColumnData`, a string specifying the [colData](#) field for controlling point size, if `SizeBy="Column data"` (see the [Panel](#) class). The specified field should contain continuous values; defaults to the first such valid field.
- `TooltipColumnData`, a character vector specifying [colData](#) fields to show in the tooltip. Defaults to `'character(0)'`, which displays only the `'colnames'` value of the data point.

In addition, this class inherits all slots from its parent [DotPlot](#) and [Panel](#) classes.

Supported methods

In the following code snippets, `x` is an instance of a [ColumnDotPlot](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.cacheCommonInfo(x)` adds a "ColumnDotPlot" entry containing `valid.colData.names`, a character vector of names of columns that are valid (i.e., contain atomic values); `discrete.colData.names`, a character vector of names for columns with discrete atomic values; and `continuous.colData.names`, a character vector of names of columns with continuous atomic values. This will also call the equivalent [DotPlot](#) method.

- `.refineParameters(x, se)` replaces NA values in `ColorByFeatAssay` with the first valid assay name in `se`. This will also call the equivalent `DotPlot` method.

For defining the interface:

- `.hideInterface(x, field)` returns a logical scalar indicating whether the interface element corresponding to `field` should be hidden. This returns `TRUE` for row selection parameters ("`RowSelectionSource`" and "`RowSelectionRestrict`"), otherwise it dispatches to the `Panel` method.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots described above and in the parent classes. This will also call the equivalent `DotPlot` method.

For controlling selections:

- `.multiSelectionRestricted(x)` returns a logical scalar indicating whether `x` is restricting the plotted points to those that were selected in a transmitting panel.
- `.multiSelectionDimension(x)` returns "`column`" to indicate that a multiple column selection is being transmitted.
- `.multiSelectionInvalidated(x)` returns `TRUE` if the faceting options use multiple column selections, such that the point coordinates/domain may change upon updates to upstream selections in transmitting panels.
- `.singleSelectionDimension(x)` returns "`sample`" to indicate that a sample identity is being transmitted.

For documentation:

- `.definePanelTour(x)` returns a `data.frame` containing the steps of a tour relevant to subclasses, mostly tuning the more generic descriptions from the same method of the parent `DotPlot`.
- `.getDotPlotColorHelp(x, color_choices)` returns a `data.frame` containing the documentation for the "`ColorBy`" UI element, specialized for column-based dot plots.

Unless explicitly specialized above, all methods from the parent classes `DotPlot` and `Panel` are also available.

Subclass expectations

Subclasses are expected to implement methods for, at least:

- `.generateDotPlotData`
- `.fullName`
- `.panelColor`

The method for `.generateDotPlotData` should create a `plot.data` `data.frame` with one row per column in the `SummarizedExperiment` object.

Author(s)

Aaron Lun

See Also

[DotPlot](#), for the immediate parent class that contains the actual slot definitions.

columnSelectionColorMap

Define the selection colormap

Description

Define the colormap when coloring points in a [DotPlot](#) based on their assigned multiple row/column selection.

Usage

```
columnSelectionColorMap(x, levels)
```

```
rowSelectionColorMap(x, levels)
```

Arguments

x	An ExperimentColorMap object.
levels	Character vector containing the available levels of a ColorBy column derived from a series of multiple selections, usually generated by multiSelectionToFactor on the selection information in row_selected or col_selected.

Details

The "unselected" level is always assigned the grey color; colors for all other levels are generated by [colDataColorMap\(x\)](#) or [rowDataColorMap\(x\)](#). The "active" level is always assigned the first color from these functions, regardless of whether it is present in levels. This aims to provide some consistency in the coloring when the selections change.

Value

A named character vector of colors for each level in levels.

Author(s)

Aaron Lun

Examples

```
ecm <- ExperimentColorMap()
columnSelectionColorMap(ecm, c("active", "unselected"))
columnSelectionColorMap(ecm, c("active", "saved1", "unselected"))
columnSelectionColorMap(ecm, c("saved1", "unselected"))
columnSelectionColorMap(ecm, c("saved1"))
```

ColumnTable-class

The ColumnTable class

Description

The ColumnTable is a virtual class where each column in the [SummarizedExperiment](#) is represented by no more than row in a [datatable](#) widget. In panels of this class, single and multiple selections can only be transmitted on the samples.

Slot overview

No new slots are added. All slots provided in the [Table](#) parent class are available.

Supported methods

In the following code snippets, `x` is an instance of a [ColumnTable](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.refineParameters(x, se)` replaces NA values in Selected with the first column name of `se`. This will also call the equivalent [Table](#) method.

For defining the interface:

- `.hideInterface(x, field)` returns a logical scalar indicating whether the interface element corresponding to `field` should be hidden. This returns TRUE for row selection parameters ("RowSelectionSource" and "RowSelectionRestrict"), otherwise it dispatches to the [Panel](#) method.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers to propagate changes in the Selected to linked plots. This will also call the equivalent [Table](#) method.

For controlling selections:

- `.multiSelectionDimension(x)` returns "column" to indicate that a column selection is being transmitted.
- `.singleSelectionDimension(x)` returns "sample" to indicate that a sample identity is being transmitted.

For rendering output:

- `.showSelectionDetails(x)` returns a HTML element containing details about the selected row. This requires a function to be registered by `registerAppOptions` under the option name `"ColumnTable.select.details"`. The function should take a string containing the name of a feature (i.e., the current selection in the `ColumnTable`) and returns a HTML element. If no function is registered, NULL is returned.

Unless explicitly specialized above, all methods from the parent classes `Table` and `Panel` are also available.

Subclass expectations

Subclasses are expected to implement methods for:

- `.generateTable`
- `.fullName`
- `.panelColor`

The method for `.generateTable` should create a tab data.frame where each row corresponds to a column in the `SummarizedExperiment` object.

Author(s)

Aaron Lun

See Also

`Table`, for the immediate parent class that contains the actual slot definitions.

ComplexHeatmapPlot-class

The ComplexHeatmapPlot panel

Description

The ComplexHeatmapPlot is a panel class for creating a `Panel` that displays an assay of a `SummarizedExperiment` object as a `Heatmap` with features as rows and samples and columns, respectively. It provides slots and methods for specifying the features of interest, which assay to display in the main heatmap, any transformations to perform on the data, and which metadata variables to display as row and column heatmap annotations.

ComplexHeatmapPlot slot overview

The following slots control the rows that are used:

- CustomRows, a logical scalar indicating whether the custom list of rows should be used. If FALSE, the incoming selection is used instead. Defaults to TRUE.
- CustomRowsText, string containing newline-separated row names. This specifies which rows of the [SummarizedExperiment](#) object are to be shown in the heatmap. If NA, defaults to the first row name of the SummarizedExperiment.
- CapRowSelection, an integer specifying the cap on the number of rows from a multiple selection to show in the heatmap, to avoid a frozen state when the application attempts to process a very large heatmap. Ignored if CustomRows = TRUE. Defaults to 200.

The following slots control the metadata variables that are used:

- ColumnData, a character vector specifying columns of the [colData](#) to show as [columnAnnotation](#). Defaults to character(0).
- RowData, a character vector specifying columns of the [rowData](#) to show as [rowAnnotation](#). Defaults to character(0).
- ShowColumnSelection, a logical vector indicating whether the column selection should be shown as an extra annotation bar. Defaults to TRUE.
- OrderColumnSelection, a logical vector indicating whether the column selection should be used to order columns in the heatmap. Defaults to TRUE.

The following slots control the choice of assay values:

- Assay, string specifying the name of the assay to use for obtaining expression values. Defaults to "logcounts" in [getPanelDefault](#), falling back to the first valid assay name (see [.cacheCommonInfo](#) below).

The following slots control the clustering of rows:

- ClusterRows, a logical scalar indicating whether rows should be clustered by their assay values. Defaults to FALSE.
- ClusterRowsDistance, string specifying a distance measure to use. This can be any one of "euclidean", "maximum", "manhattan", "canberra", "binary", "minkowski", "pearson", "spearman", or "kendall". Defaults to "spearman".
- ClusterRowsMethod, string specifying a linkage method to use. This can be any one of "ward.D", "ward.D2", "single", "complete", "average", "mcquitty", "median", or "centroid". Defaults to "ward.D2".

The following control transformations applied to rows:

- AssayCenterRows is a logical scalar indicating whether assay values should be centered for each row.
- AssayScaleRows is a logical scalar indicating whether assay values should be scaled for each row. This transformation is only applicable if AssayCenterRows is TRUE.

The following slots control the color scale:

- CustomBounds is logical scale indicating whether the color scale should be constrained by an upper and/or a lower bounds.
- LowerBound is a numerical value setting the lower bound of the color scale; or NA to disable the lower bound when CustomBounds is TRUE.
- UpperBound is a numerical value setting the lower bound of the color scale; or NA to disable the upper bound when CustomBounds is TRUE.
- DivergentColormap is a character scalar indicating a 3-color divergent colormap to use when AssayCenterRows is TRUE.

The following slots refer to general plotting parameters:

- ShowDimNames, a character vector specifying the dimensions for which to display names. This can contain zero or more of "Rows" and "Columns". Defaults to "Rows".
- NamesRowFontSize, a numerical value setting the font size of the row names.
- NamesColumnFontSize, a numerical value setting the font size of the column names.
- LegendPosition, string specifying the position of the legend on the plot. Defaults to "Bottom" in [getPanelDefault](#) but can also be "Right".
- LegendDirection, string specifying the orientation of the legend on the plot for continuous covariates. Defaults to "Horizontal" in [getPanelDefault](#) but can also be "Vertical".

The following slots control some aspects of the user interface:

- DataBoxOpen, a logical scalar indicating whether the data parameter box should be open. Defaults to FALSE.
- VisualBoxOpen, a logical scalar indicating whether the visual parameter box should be open. Defaults to FALSE.

In addition, this class inherits all slots from its parent [Panel](#) class.

Constructor

`ComplexHeatmapPlot(...)` creates an instance of a `ComplexHeatmapPlot` class, where any slot and its value can be passed to ... as a named argument.

Supported methods

In the following code snippets, `x` is an instance of a [ComplexHeatmapPlot](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.cacheCommonInfo(x)` adds a "ComplexHeatmapPlot" entry containing `valid.assay.names`, a character vector of valid (i.e., non-empty) assay names; `discrete.assay.names`, a character vector of valid assay names with discrete atomic values; `continuous.assay.names`, a character vector of valid assay names with continuous atomic values; `valid.colData.names`, a character vector of names of columns in `colData` that are valid; `discrete.colData.names`, a character vector of names for columns in `colData` with discrete atomic values; `continuous.colData.names`, a character vector of names of columns in `colData` with continuous atomic values; `valid.rowData.names`, a character vector of names of columns in `rowData` that are valid; `discrete.rowData.names`,

a character vector of names for columns in `rowData` with discrete atomic values; `continuous.rowData.names`, a character vector of names of columns in `rowData` with continuous atomic values. Valid assay names are defined as those that are non-empty, i.e., not `""`; valid columns in `colData` and `rowData` are defined as those that contain atomic values. This will also call the equivalent `Panel` method.

- `.refineParameters(x, se)` replaces any NA value in "Assay" with the first valid assay name; and NA value in "CustomRowsText" with the first row name. This will also call the equivalent `Panel` method for further refinements to `x`. If no valid column metadata fields are available, NULL is returned instead.

For defining the interface:

- `.defineInterface(x, se, select_info)` defines the user interface for manipulating all slots described above and in the parent classes. TODO It will also create a data parameter box that can respond to specialized `.defineDataInterface`, and a visual parameter box and a selection parameter box both specific to the `ComplexHeatmapPlot` panel. This will *override* the `Panel` method.
- `.defineDataInterface(x, se, select_info)` returns a list of interface elements for manipulating all slots described above.
- `.defineOutput(x)` returns a UI element for a brushable plot.
- `.panelColor(x)` will return the specified default color for this panel class.
- `.hideInterface(x, field)` returns a logical scalar indicating whether the interface element corresponding to `field` should be hidden. This returns TRUE for the selection history ("SelectionHistory"), otherwise it dispatches to the `Panel` method.

For generating the output:

- `.generateOutput(x, se, all_memory, all_contents)` returns a list containing plot, a `Heatmap` object; commands, a list of character vector containing the R commands required to generate contents and plot; and contents and varname, both set to NULL as this is not a transmitting panel.
- `.exportOutput(x, se, all_memory, all_contents)` will create a PDF file containing the current plot, and return a string containing the path to that PDF. This assumes that the plot field returned by `.generateOutput` is a `Heatmap` object.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots described above and in the parent classes. This will also call the equivalent `Panel` method.
- `.renderOutput(x, se, output, pObjects, rObjects)` will add a rendered plot element to output. The reactive expression will add the contents of the plot to `pObjects$contents` and the relevant commands to `pObjects$commands`. This will also call the equivalent `Panel` method to render the panel information text boxes.

For defining the panel name:

- `.fullName(x)` will return "Complex heatmap".

For documentation:

- `.definePanelTour(x)` returns an `data.frame` containing a panel-specific tour.

Author(s)

Kevin Rue-Albrecht

See Also[Panel](#), for the immediate parent class.**Examples**

```
#####
# For end-users #
#####

x <- ComplexHeatmapPlot()
x[["ShowDimNames"]]
x[["ShowDimNames"]] <- c("Rows", "Columns")

#####
# For developers #
#####

library(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)

old_cd <- colData(sce)
colData(sce) <- NULL

# Spits out a NULL and a warning if there is nothing to plot.
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

# Replaces the default with something sensible.
colData(sce) <- old_cd
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)
```

constants

*Constants***Description**

Constant values used throughout iSEE panels and extensions.

Panel slot names

.dataParamBoxOpen Name of slot that indicates whether the 'Data parameter' box is open.

.multiSelectHistory Name of slot that stores the list of saved selections.

`.organizationHeight` Name of slot that stores the panel height.

`.organizationWidth` Name of slot that stores the panel width.

Multiple selection parameters

`.noSelection` Value displayed in the absence of selection.

Author(s)

Kevin Rue-Albrecht

createCustomPanels	<i>Create custom panels</i>
--------------------	-----------------------------

Description

Helper functions for quick-and-dirty creation of custom panels, usually in the context of a one-off application. This creates a new class with specialized methods for showing content based on a user-specified function.

Usage

```
createCustomTable(
  FUN,
  restrict = NULL,
  className = "CustomTable",
  fullName = "Custom table",
  FUN.selection.details = NULL,
  where = topenv(parent.frame())
)

createCustomPlot(
  FUN,
  restrict = NULL,
  className = "CustomPlot",
  fullName = "Custom plot",
  where = topenv(parent.frame())
)
```

Arguments

<code>FUN</code>	A function that generates a <code>data.frame</code> or a ggplot , for <code>createCustomTable</code> and <code>createCustomPlot</code> respectively. See Details for the expected arguments.
<code>restrict</code>	Character vector of names of optional arguments in <code>FUN</code> to which the UI is restricted. If specified, only the listed arguments receive UI elements in the interface.
<code>className</code>	String containing the name of the new Panel class.

fullName	String containing the full name of the new class.
FUN.selection.details	Function generating a UI element that displays details about the current selection, if any.
where	An environment indicating where the class and method definitions should be stored.

Details

FUN is expected to have the following first 3 arguments:

- se, a [SummarizedExperiment](#) object for the current dataset of interest.
- rows, a list of row selections received from the transmitting panel. This contains one or more character vectors of row names in active and saved selections. Alternatively, this may be NULL if no selection has been made in the transmitter.
- columns, a list of column selections received from the transmitting panel. This contains one or more character vectors of column names in active and saved selections. Alternatively, this may be NULL if no selection has been made in the transmitter.

Any number of additional named arguments may also be present in FUN. All such arguments should have default values, as these are used to automatically generate UI elements in the panel:

- Character vectors will get a [selectInput](#).
- Strings will get a [textInput](#).
- Numeric scalars will get a [numericInput](#).
- Logical scalars will get a [checkboxInput](#).

Arguments with other types of default values are ignored. If `restrict` is specified, arguments will only have corresponding UI elements if they are listed in `restrict`. All user interactions with these elements will automatically trigger regeneration of the panel contents.

Classes created via these functions are extremely limited. Only scalar inputs are supported via the UI and all panels cannot transmit to the rest of the app. We recommend only using these functions for one-off applications to quickly prototype concepts; serious [Panel](#) extensions should be done explicitly.

Value

A new class and its methods are defined in the global environment. A generator function for creating new instances of the class is returned.

Author(s)

Aaron Lun

Examples

```

library(scater)
CUSTOM_DIMRED <- function(se, rows, columns, ntop=500, scale=TRUE,
  mode=c("PCA", "TSNE", "UMAP"))
{
  if (is.null(columns)) {
    return(
      ggplot() + theme_void() + geom_text(
        aes(x, y, label=label),
        data.frame(x=0, y=0, label="No column data selected."),
        size=5)
    )
  }

  mode <- match.arg(mode)
  if (mode=="PCA") {
    calcFUN <- runPCA
  } else if (mode=="TSNE") {
    calcFUN <- runTSNE
  } else if (mode=="UMAP") {
    calcFUN <- runUMAP
  }

  kept <- se[, unique(unlist(columns))]
  kept <- calcFUN(kept, ncomponents=2, ntop=ntop,
    scale=scale, subset_row=unique(unlist(rows)))
  plotReducedDim(kept, mode)
}

GEN <- createCustomPlot(CUSTOM_DIMRED)
GEN()

if (interactive()) {
  library(scRNAseq)
  sce <- ReprocessedAllenData("tophat_counts")
  library(scater)
  sce <- logNormCounts(sce, exprs_values="tophat_counts")

  iSEE(sce, initial=list(
    ColumnDataPlot(PanelId=1L),
    GEN(ColumnSelectionSource="ColumnDataPlot1")
  ))
}

```

Description

Define a function to create a landing page in which users can specify or upload [SummarizedExperiment](#) objects.

Usage

```
createLandingPage(  
  seUI = NULL,  
  seLoad = NULL,  
  initUI = NULL,  
  initLoad = NULL,  
  requireButton = TRUE  
)
```

Arguments

seUI	Function that accepts a single id argument and returns a UI element for specifying the SummarizedExperiment.
seLoad	Function that accepts the input value of the UI element from seUI and returns a SummarizedExperiment object.
initUI	Function that accepts a single id argument and returns a UI element for specifying the initial state.
initLoad	Function that accepts the input value of the UI element from initUI and returns a list of Panels .
requireButton	Logical scalar indicating whether the app should require an explicit button press to initialize, or if it should initialize upon any modification to the UI element in seUI.

Details

By default, this function creates a landing page in which users can upload an RDS file containing a SummarizedExperiment, which is subsequently read by [readRDS](#) to launch an instance of [iSEE](#). However, any source of SummarizedExperiment objects can be used; for example, we can retrieve them from databases by modifying seUI and seLoad appropriately.

The default landing page also allows users to upload a RDS file containing a list of [Panels](#) that specifies the initial state of the [iSEE](#) instance (to be used as the initial argument in [iSEE](#)). Again, any source can be used to create this list if initUI and initLoad are modified appropriately.

The UI elements for the SummarizedExperiment and the initial state are named "se" and "initial" respectively. This can be used in Shiny bookmarking to initialize an [iSEE](#) in a desired state by simply clicking a link, provided that requireButton=FALSE so that reactive expressions are immediately triggered upon setting se= and initial= in the URL. We do not use bookmarking to set all individual [iSEE](#) parameters as we will run afoul of URL character limits.

Value

A function that generates a landing page upon being passed to [iSEE](#) as the landingPage argument.

Defining a custom landing page

We note that `createLandingPage` is just a limited wrapper around the landing page API. In `iSEE`, `landingPage` can be any function that accepts the following arguments:

- `FUN`, a function to initialize the `iSEE` observer architecture. This function expects to be passed:
 - `SE`, a `SummarizedExperiment` object.
 - `INITIAL`, a list of `Panel` objects describing the initial application state. If `NULL`, the initial state from `initial` in the top-level `iSEE` call is used instead.
 - `TOUR`, a `data.frame` containing a tour to be attached to the app - see `defaultTour` for an example. If `NULL` (the default), no tour is added.
 - `COLORMAP`, an `ExperimentColorMap` object that defines the colormaps to use in the application.
- `input`, the Shiny input list.
- `output`, the Shiny output list.
- `session`, the Shiny session object.

The `landingPage` function should define a `renderUI` expression that is assigned to `output$allPanels`. This should define a UI that contains all widgets necessary for a user to set up an `iSEE` session interactively. We suggest that all UI elements have IDs prefixed with `"initialize_INTERNAL"` to avoid conflicts.

The function should also define observers to respond to user interactions with the UI elements. These are typically used to define a `SummarizedExperiment` object and an input state as a list of `Panels`; any one of these observers may then call `FUN` on those arguments to launch the main `iSEE` instance.

Note that, once the main app is launched, the UI elements constructed here are lost and observers will never be called again. There is no explicit “unload” mechanism to return to the landing page from the main app, though a browser refresh is usually sufficient.

Author(s)

Aaron Lun

Examples

```
createLandingPage()

# Alternative approach, to create a landing page
# that opens one of the datasets from the scRNAseq package.
library(scRNAseq)
all.data <- ls("package:scRNAseq")
all.data <- all.data[grepl("Data$", all.data)]

lpfun <- createLandingPage(
  seUI=function(id) selectInput(id, "Dataset:", choices=all.data),
  seLoad=function(x) get(x, as.environment("package:scRNAseq"))()
)

app <- iSEE(landingPage=lpfun)
```

```
if (interactive()) {  
  shiny::runApp(app, port=1234)  
}
```

`defaultTour`*Define the default tour*

Description

Define the default tour for the subset of the Allen brain dataset. This is only available when run on the `iSEE(sce)` example in ?"[iSEE](#)".

Usage

```
defaultTour()
```

Value

A `data.frame` where each row is a tour step. The first column specifies the UI element to be highlighted by the tour, while the second column contains the tour text.

Author(s)

Aaron Lun

Examples

```
defaultTour()
```

`documentation-generics`*Documentation generics*

Description

The generics power the creation of panel-specific documentation within the [iSEE](#) app. Users can click on an icon next to the panel name to open a self-guided tour for that panel's functionality.

Defining the panel tour

`.definePanelTour(x)` takes a [Panel](#) `x` and is expected to return a `data.frame` with the character fields "element" and "intro". Each row corresponds to a step of an **rintrojs** tour; the "element" specifies the active UI element to be highlighted in that step, while the "intro" element contains the HTML-formatted text to show in the tour pop-up.

It is a good idea to `callNextMethod()` to obtain the tour steps for the parent class to append onto the current class's `data.frame`. In some cases, modification of the parent class's tour steps may be necessary if some of the parent's functionality has been overwritten. Some communication with the parent's maintainers may be necessary to establish a stable way to identify the rows corresponding to the steps to be written, e.g., based on the row names of the `data.frame`.

A tour for a Panel `x` is expected to only highlight UI elements *on the same panel*. This is very important as other panels cannot be assumed to exist in an arbitrary instance of [iSEE](#). As such, these tours are not well-suited to highlighting interactions between different panels.

The observer set-up for the panel tour is done in `.createObservers` for the base [Panel](#) class. No further action is required on behalf of developers to set up the triggers to launch the tour.

Defining UI-specific tours

It is possible to provide tours for individual UI elements, which can be more helpful than a single large tour for the entire panel. A documented element has a clickable icon (usually generated by functions like `.selectInput.iSEE`) that launches a specific tour, typically explaining the behavior and effects of the associated parameter. The tours themselves should be registered by `.addSpecificTour`. This is best done inside the various interface-defining functions (e.g., `.defineInterface` and related methods) where the documentation can be written adjacent to the definition of the UI element itself.

For a [DotPlot](#) instance `x`, the `.getDotPlotColorHelp(x, color_choices)` generic should return a function that returns a `data.frame` containing the **rintrojs** tour for the color choice UI element, i.e., "ColorBy". This allows downstream Panels to tune the wording of the color documentation, given that this is commonly specialized. `color_choices` is a character vector that contains the valid choices for the "ColorBy" radio button; some input datasets will not have, e.g., any column data, so the corresponding button will not be shown and its associated tour can be omitted.

Author(s)

Aaron Lun

DotPlot-class

The DotPlot virtual class

Description

The [DotPlot](#) is a virtual class for all panels where each row or column in the [SummarizedExperiment](#) is represented by no more than one point (i.e., a "dot") in a brushable [ggplot](#) plot. It provides slots and methods to create the plot, to control various aesthetics of the dots, and to store the brush or lasso selection.

Slot overview

The following slots are relevant to coloring of the points:

- **ColorBy**, a string specifying how points should be colored. This should be one of "None", "Feature name", "Sample name" and either "Column data" (for [ColumnDotPlots](#)) or "Row data" (for [RowDotPlots](#)). Defaults to "None".
- **ColorByDefaultColor**, a string specifying the default color to use for all points if ColorBy="None". Defaults to "black" in [getPanelDefault](#).
- **ColorByFeatureName**, a string specifying the feature to be used for coloring points when ColorBy="Feature name". For [RowDotPlots](#), this is used to highlight the point corresponding to the selected feature; for [ColumnDotPlots](#), this is used to color each point according to the expression of that feature. If NA, this defaults to the name of the first row.
- **ColorByFeatureSource**, a string specifying the name of the panel to use for transmitting the feature selection to ColorByFeatureName. Defaults to "---".
- **ColorBySampleName**, a string specifying the sample to be used for coloring points when ColorBy="Sample name". For [RowDotPlots](#), this is used to color each point according to the expression of that sample; for [ColumnDotPlots](#), this is used to highlight the point corresponding to the selected sample. If NA, this defaults to the name of the first column.
- **ColorBySampleSource**, a string specifying the name of the panel to use for transmitting the sample selection to ColorBySampleNameColor. Defaults to "---".
- **ColorByFeatureDynamicSource**, a logical scalar indicating whether x should dynamically change its selection source when coloring by feature. Defaults to FALSE in [getPanelDefault](#).
- **ColorBySampleDynamicSource**, a logical scalar indicating whether x should dynamically change its selection source when coloring by feature. Defaults to FALSE in [getPanelDefault](#).
- **SelectionAlpha**, a numeric scalar in [0, 1] specifying the transparency to use for non-selected points. Defaults to 0.1 in [getPanelDefault](#).

The following slots control other metadata-related aesthetic aspects of the points:

- **ShapeBy**, a string specifying how the point shape should be determined. This should be one of "None" and either "Column data" (for [ColumnDotPlots](#)) or "Row data" (for [RowDotPlots](#)). Defaults to "None".
- **SizeBy**, a string specifying the metadata field for controlling point size. This should be one of "None" and either "Column data" (for [ColumnDotPlots](#)) or "Row data" (for [RowDotPlots](#)). Defaults to "None".

The following slots control the faceting:

- **FacetRowBy**, a string indicating what to use for creating row facets. For [RowDotPlots](#), this should be one of "None", "Row data" or "Row selection". For [ColumnDotPlots](#), this should be one of "None", "Column data" or "Column selection". Defaults to "None", i.e., no row faceting.
- **FacetByColumn**, a string indicating what to use for creating column facets. For [RowDotPlots](#), this should be one of "None", "Row data" or "Row selection". For [ColumnDotPlots](#), this should be one of "None", "Column data" or "Column selection". Defaults to "None", i.e., no column faceting.

The following slots control any text to be shown on the plot:

- `LabelCenters`, a logical scalar indicating whether the label the centers (technically medoids) of all cells in each group, where groups are defined by a discrete covariate in the relevant metadata field. Defaults to `FALSE`.
- `LabelCentersBy`, a string specifying the metadata field to define the groups when `LabelCenters` is `TRUE`. This should be a discrete variable in `rowData` or `colData` for `RowDotPlots` and `ColumnDotPlots`, respectively. Defaults to the name of the first column.
- `LabelCentersColor`, a string specifying the color used for the labels at the center of each group. Only used when `LabelCenters` is `TRUE`. Defaults to `"black"`.
- `CustomLabels`, a logical scalar indicating whether custom labels should be inserted on specific points. Defaults to `FALSE`.
- `CustomLabelsText`, a (possibly multi-line) string with the names of the points to label when `CustomLabels` is set to `TRUE`. Each line should contain the name of a row or column for `RowDotPlots` and `ColumnDotPlots`, respectively. Leading and trailing whitespace are stripped, and all text on a line after `#` is ignored. Defaults to the name of the first row/column.

The following slots control interactions with the plot image:

- `ZoomData`, a named numeric vector of plot coordinates with `"xmin"`, `"xmax"`, `"ymin"` and `"ymax"` elements parameterizing the zoom boundaries. Defaults to an empty vector, i.e., no zoom.
- `BrushData`, a list containing either a Shiny brush (see `?brushedPoints`) or an `iSEE` lasso (see `?lassoPoints`). Defaults to an empty list, i.e., no brush or lasso.
- `HoverInfo`, a logical scalar indicating whether the feature/sample name should be shown upon mouse-over of the point. Defaults to `TRUE`.

The following slots control some aspects of the user interface:

- `DataBoxOpen`, a logical scalar indicating whether the data parameter box should be open. Defaults to `FALSE`.
- `VisualBoxOpen`, a logical scalar indicating whether the visual parameter box should be open. Defaults to `FALSE`.
- `VisualChoices`, a character vector specifying the visible interface elements upon initialization. This can contain zero or more of `"Color"`, `"Shape"`, `"Size"`, `"Point"`, `"Facet"`, `"Text"`, and `"Other"`. Defaults to `"Color"`.

The following slots control the addition of a contour:

- `ContourAdd`, logical scalar indicating whether a contour should be added to a (scatter) plot. Defaults to `FALSE`.
- `ContourColor`, string specifying the color to use for the contour lines. Defaults to `"blue"`.

The following slot controls whether the aspect ratio is fixed to 1 or not:

- `FixAspectRatio`, logical scalar indicating whether the aspect ratio of a scatter plot should be fixed to 1. Defaults to `FALSE`.

The following slot controls whether the violin boundaries are plotted or not:

- `ViolinAdd`, logical scalar indicating whether the violin boundaries should be plotted. Defaults to `TRUE`.

The following slots control the general appearance of the points.

- `PointSize`, positive numeric scalar specifying the relative size of the points. Defaults to 1.
- `PointAlpha`, non-negative numeric scalar specifying the transparency of the points. Defaults to 1, i.e., not transparent.
- `Downsample`, logical scalar indicating whether to downsample points for faster plotting. Defaults to `FALSE` in `getPanelDefault`.
- `DownsampleResolution`, numeric scalar specifying the resolution of the downsampling grid (see `?subsetPointsByGrid`) if `Downsample=TRUE`. Larger values correspond to reduced downsampling at the cost of plotting speed. Defaults to 200 in `getPanelDefault`.

The following slots refer to general plotting parameters:

- `FontSize`, positive numeric scalar specifying the relative font size. Defaults to 1 in `getPanelDefault`.
- `PointSize`, positive numeric scalar specifying the relative point size. Defaults to 1 in `getPanelDefault`.
- `LegendPosition`, string specifying the position of the legend on the plot. Defaults to "Bottom" in `getPanelDefault`. Other valid choices include "Right" and "None".

In addition, this class inherits all slots from its parent `Panel` class.

Supported methods

In the following code snippets, `x` is an instance of a `DotPlot` class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up the objects:

- `.cacheCommonInfo(x)` adds a "DotPlot" entry containing `valid.assay.names`, a character vector of valid assay names. Valid names are defined as those that are non-empty, i.e., not `""`. This method will also call the equivalent `Panel` method.
- `.refineParameters(x, se)` replaces NA values in `ColorByFeatureName` and `ColorBySampleNameColor` with the first row and column name, respectively, of `se`. This will also call the equivalent `Panel` method.

For defining the interface:

- `.defineInterface(x, se, select_info)` defines the user interface for manipulating all slots described above and in the parent classes. It will also create a data parameter box that can respond to specialized `.defineDataInterface`. This will *override* the `Panel` method.
- `.defineVisualColorInterface(x, se, select_info)` defines the user interface subpanel for manipulating the color of the points.
- `.defineVisualShapeInterface(x, se)` defines the user interface subpanel for manipulating the shape of the points.
- `.defineVisualSizeInterface(x, se)` defines the user interface subpanel for manipulating the size of the points.

- `.defineVisualPointInterface(x, se)` defines the user interface subpanel for manipulating other point-related parameters.
- `.defineVisualFacetInterface(x, se)` defines the user interface subpanel for manipulating facet-related parameters.
- `.defineVisualTextInterface(x, se)` defines the user interface subpanel for manipulating text-related parameters.
- `.defineVisualOtherInterface(x, se)` defines the user interface subpanel for manipulating other parameters. Currently this returns NULL.
- `.defineOutput(x)` returns a UI element for a brushable plot.
- `.allowableColorByDataChoices(x, se)` returns a character vector containing all atomic variables in the relevant *Data dimension.

For generating the output:

- `.generateOutput(x, se, all_memory, all_contents)` returns a list containing contents, a data.frame with one row per point currently present in the plot; plot, a `ggplot` object; commands, a list of character vector containing the R commands required to generate contents and plot; and varname, a string containing the name of the variable in commands that was used to obtain contents.
- `.generateDotPlot(x, labels, envir)` returns a list containing plot and commands, as described above. This is called within `.generateOutput` for all `DotPlot` instances by default. Methods are also guaranteed to generate a `dot.plot` variable in `envir` containing the `ggplot` object corresponding to plot.
- `.prioritizeDotPlotData(x, envir)` returns NULL.
- `.colorByNoneDotPlotField(x)` returns NULL.
- `.colorByNoneDotPlotScale(x)` returns NULL.
- `.exportOutput(x, se, all_memory, all_contents)` will create a PDF file containing the current plot, and return a string containing the path to that PDF. This assumes that the plot field returned by `.generateOutput` is a `ggplot` object.

For defining reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for some (but not all!) of the slots. This will also call the equivalent `Panel` method.
- `.renderOutput(x, se, output, pObjects, rObjects)` will add a rendered plot element to output. The reactive expression will add the contents of the plot to `pObjects$contents` and the relevant commands to `pObjects$commands`. This will also call the equivalent `Panel` method to render the panel information text boxes.

For controlling selections:

- `.multiSelectionCommands(x, index)` returns a character vector of R expressions that - when evaluated - returns a character vector of the names of selected points in the active and/or saved selections of x. The active selection is returned if `index=NA`, otherwise one of the saved selection is returned.
- `.multiSelectionActive(x)` returns `x[["BrushData"]]` or NULL if there is no brush or closed lasso.

- `.multiSelectionClear(x)` returns `x` after setting the BrushData slot to an empty list.
- `.singleSelectionValue(x, contents)` returns the name of the first selected element in the active brush. If no brush is active, NULL is returned instead.
- `.singleSelectionSlots(x)` will return a list specifying the slots that can be updated by single selections in transmitter panels, mostly related to the choice of coloring parameters. This includes the output of `callNextMethod`.

For documentation:

- `.definePanelTour(x)` returns an data.frame containing the steps of a tour relevant to subclasses, mostly describing the specification of visual effects and the creation of a brush or lasso.

Unless explicitly specialized above, all methods from the parent class `Panel` are also available.

Subclass expectations

The DotPlot is a rather vaguely defined class for which the only purpose is to avoid duplicating code for `ColumnDotPlots` and `RowDotPlots`. We recommend extending those subclasses instead.

Author(s)

Aaron Lun

See Also

`RowDotPlot` and `ColumnDotPlot`, which are more amenable to extension.

ExperimentColorMap-class

ExperimentColorMap class

Description

ExperimentColorMap class

Usage

```
ExperimentColorMap(
  assays = list(),
  colData = list(),
  rowData = list(),
  all_discrete = list(assays = NULL, colData = NULL, rowData = NULL),
  all_continuous = list(assays = NULL, colData = NULL, rowData = NULL),
  global_discrete = NULL,
  global_continuous = NULL,
  ...
)
```

Arguments

<code>assays</code>	List of colormaps for assays.
<code>colData</code>	List of colormaps for colData.
<code>rowData</code>	List of colormaps for rowData.
<code>all_discrete</code>	Colormaps applied to all undefined categorical assays, colData, and rowData, respectively.
<code>all_continuous</code>	Colormaps applied to all undefined continuous assays, colData, and rowData, respectively.
<code>global_discrete</code>	Colormap applied to all undefined categorical covariates.
<code>global_continuous</code>	Colormap applied to all undefined continuous covariates.
<code>...</code>	additional arguments passed on to the ExperimentColorMap constructor

Details

Colormaps must all be functions that take at least one argument: the number of (named) colours to return as a character vector. This argument may be ignored in the body of the colormap function to produce constant colormaps.

Value

An object of class ExperimentColorMap

Categorical colormaps

The default categorical colormap emulates the default ggplot2 categorical color palette (Credit: <https://stackoverflow.com/questions/8197559/emulate-ggplot2-default-color-palette>). This palette returns a set of colors sampled in steps of equal size that correspond to approximately equal perceptual changes in color:

```
function(n) {
  hues=seq(15, 375, length=(n + 1))
  hcl(h=hues, l=65, c=100)[seq_len(n)]
}
```

To change the palette for all categorical variables, users must supply a colormap that returns a similar value; namely, an unnamed character vector of length n. For instance, using the base R palette `rainbow.colors`

```
function(n) {
  rainbow(n)
}
```

Accessors

In the following code snippets, *x* is an ExperimentColorMap object.

`assayColorMap(x, i, ..., discrete=FALSE)`: Get an assays colormap for the specified assay *i*.

`colDataColorMap(x, i, ..., discrete=FALSE)`: Get a colData colormap for the specified colData column *i*.

`rowDataColorMap(x, i, ..., discrete=FALSE)`: Get a rowData colormap for the specified rowData column *i*.

If the colormap for *i* cannot be found, one of the default colormaps is returned. In this case, *discrete* is a logical scalar that indicates whether the colormap should be categorical. The more specialized default is first attempted - e.g., for `assayColorMap`, this would be the assay colormap specified in assays of `all_discrete` or `all_continuous` - before falling back to the global default in `global_discrete` or `global_continuous`. Similarly, if *i* is missing, the default discrete/continuous colormap is returned.

Setters

In the following code snippets, *x* is an ExperimentColorMap object, and *i* is a character or numeric index.

`assayColorMap(x, i, ...) <- value`: Set an assays colormap.

`colDataColorMap(x, i, ...) <- value`: Set a colData colormap.

`rowDataColorMap(x, i, ...) <- value`: Set a rowData colormap.

`assay(x, i, ...) <- value`: Alias. Set an assays colormap.

Examples

```
# Example colormaps ----

count_colors <- function(n){
  c("black", "brown", "red", "orange", "yellow")
}
fpkm_colors <- viridis::inferno
tpm_colors <- viridis::plasma

qc_color_fun <- function(n){
  qc_colors <- c("forestgreen", "firebrick1")
  names(qc_colors) <- c("Y", "N")
  return(qc_colors)
}

# Constructor ----

ecm <- ExperimentColorMap(
  assays=list(
    counts=count_colors,
    tophat_counts=count_colors,
```

```

        cufflinks_fpkms=fpkm_colors,
        rsem_tpm=tpm_colors
    ),
    colData=list(
        passes_qc_checks_s=qc_color_fun
    )
)

# Accessors ----

# assay colormaps
assayColorMap(ecm, "logcounts") # [undefined --> default]
assayColorMap(ecm, "counts")
assayColorMap(ecm, "cufflinks_fpkms")
assay(ecm, "cufflinks_fpkms") # alias

# colData colormaps
colDataColorMap(ecm, "passes_qc_checks_s")
colDataColorMap(ecm, "undefined")

# rowData colormaps
rowDataColorMap(ecm, "undefined")

# generic accessors
assays(ecm)
assayNames(ecm)

# Setters ----

assayColorMap(ecm, "counts") <- function(n){c("blue", "white", "red")}
assay(ecm, 1) <- function(n){c("blue", "white", "red")}

colDataColorMap(ecm, "passes_qc_checks_s") <- function(n){NULL}
rowDataColorMap(ecm, "undefined") <- function(n){NULL}

# Categorical colormaps ----

# Override all discrete colormaps using the base rainbow palette
ecm <- ExperimentColorMap(global_discrete = rainbow)
n <- 10
plot(1:n, col=assayColorMap(ecm, "undefined", discrete = TRUE)(n), pch=20, cex=3)

```

FeatureAssayPlot-class

The FeatureAssayPlot panel

Description

The FeatureAssayPlot is a panel class for creating a [ColumnDotPlot](#) where the y-axis represents the expression of a feature of interest, using the [assay](#) values of the [SummarizedExperiment](#). It

provides slots and methods to specify the feature and what to plot on the x-axis, as well as a method to actually create a data.frame containing those pieces of data in preparation for plotting.

Slot overview

The following slots control the values on the y-axis:

- `YAxisFeatureName`, a string specifying the name of the feature to plot on the y-axis. If NA, defaults to the first row name of the `SummarizedExperiment` object.
- `Assay`, string specifying the name of the assay to use for obtaining expression values. Defaults to "logcounts" in `getPanelDefault`, falling back to the name of the first valid assay (see `?".cacheCommonInfo, DotPlot-method"` for the definition of validity).
- `YAxisFeatureSource`, string specifying the encoded name of the transmitting panel to obtain a single selection that replaces `YAxisFeatureName`. Defaults to "---", i.e., no transmission is performed.
- `YAxisFeatureDynamicSource`, a logical scalar indicating whether x should dynamically change its selection source for the y-axis. Defaults to FALSE in `getPanelDefault`.

The following slots control the values on the x-axis:

- `XAxis`, string specifying what should be plotting on the x-axis. This can be any one of "None", "Feature name", "Column data" or "Column selection". Defaults to "None".
- `XAxisColumnData`, string specifying which column of the `colData` should be shown on the x-axis, if `XAxis`="Column data". Defaults to the first valid `colData` field (see `?".refineParameters, ColumnDotPlot-m"` for details).
- `XAxisFeatureName`, string specifying the name of the feature to plot on the x-axis, if `XAxis`="Feature name". Defaults to the first row name.
- `XAxisFeatureSource`, string specifying the encoded name of the transmitting panel to obtain a single selection that replaces `XAxisFeatureName`. Defaults to "---", i.e., no transmission is performed.
- `XAxisFeatureDynamicSource`, a logical scalar indicating whether x should dynamically change its selection source for the x-axis. Defaults to FALSE in `getPanelDefault`.

In addition, this class inherits all slots from its parent `ColumnDotPlot`, `DotPlot` and `Panel` classes.

Constructor

`FeatureAssayPlot(...)` creates an instance of a `FeatureAssayPlot` class, where any slot and its value can be passed to ... as a named argument.

Supported methods

In the following code snippets, `x` is an instance of a `FeatureAssayPlot` class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.refineParameters(x, se)` replaces any NA values in `XAxisFeatureName` and `YAxisFeatureName` with the first row name; any NA value in `Assay` with the first valid assay name; and any NA value in `XAxisColumnData` with the first valid column metadata field. This will also call the equivalent `ColumnDotPlot` method for further refinements to `x`. If no rows or assays are present, `NULL` is returned instead.

For defining the interface:

- `.defineDataInterface(x, se, select_info)` returns a list of interface elements for manipulating all slots described above.
- `.panelColor(x)` will return the specified default color for this panel class.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots described above and in the parent classes. This will also call the equivalent `ColumnDotPlot` method.

For defining the panel name:

- `.fullName(x)` will return "Feature assay plot".

For creating the plot:

- `.generateDotPlotData(x, envir)` will create a `data.frame` of feature expression values in `envir`. It will return the commands required to do so as well as a list of labels.

For managing selections:

- `.singleSelectionSlots(x)` will return a list specifying the slots that can be updated by single selections in transmitter panels, mostly related to the choice of feature on the x- and y-axes. This includes the output of the method for the parent `ColumnDotPlot` class.
- `.multiSelectionInvalidated(x)` returns `TRUE` if the x-axis uses multiple column selections, such that the point coordinates may change upon updates to upstream selections in transmitting panels. Otherwise, it dispatches to the `ColumnDotPlot` method.

For documentation:

- `.definePanelTour(x)` returns an `data.frame` containing a panel-specific tour.

Author(s)

Aaron Lun

See Also

`ColumnDotPlot`, for the immediate parent class.

Examples

```
#####
# For end-users #
#####

x <- FeatureAssayPlot()
x[["XAxis"]]
x[["Assay"]] <- "logcounts"
x[["XAxisColumnData"]] <- "stuff"

#####
# For developers #
#####

library(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)

old_assay_names <- assayNames(sce)
assayNames(sce) <- character(length(old_assay_names))

# Spits out a NULL and a warning if no assays are named.
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

# Replaces the default with something sensible.
assayNames(sce) <- old_assay_names
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)
```

filterDTColumn

Filter DT columns

Description

Filter a data.frame based on the **DT** [datatable](#) widget column search string.

Usage

```
filterDTColumn(x, search)

filterDT(df, column, global)
```

Arguments

x	A numeric or character vector, usually representing a column of a data.frame.
search	A string specifying the search filter to apply to x.
df	A data.frame that was used in the datatable widget.

column	A character vector of per-column search strings to apply to df. If any entry is an empty string, the corresponding column is not used for any filtering.
global	String containing a regular expression to search for across all columns in df (and row names, if present). If an empty string, no filtering is performed.

Details

For character *x*, search is treated as a regular expression.

For numeric *x*, search should have the form LOWER ... UPPER where all elements in [LOWER, UPPER] are retained.

For factor *x*, search should have the form ["choice_1", "choice_2", etc.]. This is also the case for logical *x*, albeit with the only choices being "true" or "false".

filterDT will retain all rows where (i) any value in any column (after coercion to a string) matches global, and (ii) the value in each column satisfies the filter specified in the corresponding entry of column. Setting global to an empty string will skip requirement (i) while setting any entry of column to an empty string will skip requirement (ii) for the affected column.

Ideally, ncol(df) and length(searches) would be the same, but if not, filterDT will simply filter on the first N entries where N is the smaller of the two.

Any NA element in *x* will be treated as a no-match. The same applies for each column of df that has non-empty column. Note that a no-match in one column does not preclude a successful match in another column by global.

Value

A logical vector indicating which entries of *x* or rows of df are to be retained.

Author(s)

Aaron Lun

See Also

[datatable](#) and associated documentation for more details about column searches.

Examples

```
# Regular expression:
filterDTColumn(LETTERS, "A|B|C")

# Range query:
filterDTColumn(runif(20), "0.1 ... 0.5")

# Factor query:
filterDTColumn(factor(letters), "['a', 'b', 'c']")

# Works on DataFrames:
X <- data.frame(row.names=LETTERS, thing=runif(26),
  stuff=sample(letters[1:3], 26, replace=TRUE),
  stringsAsFactors=FALSE)
```

```
filterDT(X, c("0 ... 0.5", "a|b"), global="")
filterDT(X, "", global="A")
```

hidden-inputs

*Hidden interface elements***Description**

Returns an interface element with or without the CSS class `shinyjs-hide`, depending on whether the element is hidden based on `.hideInterface`. This allows panels to hide interface elements that are provided by parent classes but are not needed in the subclass.

Usage

```
.selectInputHidden(x, field, ...)
```

Arguments

<code>x</code>	An instance of a Panel class.
<code>field</code>	String specifying the name of the suffix of the ID of the interface element.
<code>...</code>	Further arguments to pass to the Shiny function responsible for producing the interface element.

Details

`.selectInputHidden(x, field, ...)` produces a Shiny [selectInput](#) element that will be hidden if `.hideInterface(x)` is TRUE.

Value

The output of `FUN(id, ...)` is returned where `id` is defined by concatenating `.getEncodedName(x)` and `field` (separated by an underscore).

Author(s)

Kevin Rue-Albrecht

Examples

```
.selectInputHidden(ComplexHeatmapPlot(), "SelectionHistory",
  choices = c(1, 2, 3), label = "Saved selection (hidden)")

.selectInputHidden(ReducedDimensionPlot(), "Type",
  choices = c("UMAP", "PCA"), label = "Reduced dimension type (hidden)")
```

Description

An overview of the generics for defining the user interface (UI) for each panel as well as some recommendations on their implementation.

Defining the parameter interface

`.defineInterface(x, se, select_info)` defines the UI for modifying all parameters for a given panel. The required arguments are:

- `x`, an instance of a [Panel](#) class.
- `se`, a [SummarizedExperiment](#) object containing the current dataset. This can be assumed to have been produced by running `.refineParameters(x, se)`.
- `select_info`, a list of two lists, `single` and `multiple`, each of which contains the character vectors `row` and `column`. This specifies the panels available for transmitting single/multiple selections on the rows/columns, see [?.multiSelectionDimension](#) and [?.singleSelectionDimension](#) for more details.

Methods for this generic are expected to return a list of [collapseBox](#) elements. Each parameter box can contain arbitrary numbers of additional UI elements, each of which is expected to modify one slot of `x` upon user interaction.

The ID of each interface element should follow the form of `PANEL_SLOT` where `PANEL` is the panel name (from `.getEncodedName(x)`) and `SLOT` is the name of the slot modified by the interface element, e.g., `"ReducedDimensionPlot1_Type"`. Each interface element should have an equivalent observer in `.createObservers` unless they are hidden by `.hideInterface` (see below).

It is the developer's responsibility to call [callNextMethod](#) to obtain interface elements for parent classes. A common strategy is to combine the output of `callNextMethod` with additional [collapseBox](#) elements to achieve the desired UI structure.

Defining the data parameter interface

`.defineDataInterface(x, se, select_info)` defines the UI for data-related (i.e., non-aesthetic) parameters. The required arguments are the same as those for `.defineInterface`. Methods for this generic are expected to return a list of UI elements for altering data-related parameters, which are automatically placed inside the "Data parameters" collapsible box. Each element's ID should still follow the `PANEL_SLOT` pattern described above.

This generic aims to provide a simpler alternative to specializing `.defineInterface` for the most common use case. New panels can write methods for this generic to add their own interface elements for altering the contents of the panel, without needing to reimplement other UI elements in the parent class's `.defineInterface` method. Conversely, there is no obligation to write a method for this generic if one is planning to specialize `.defineInterface`.

It is the developer's responsibility to call [callNextMethod](#) to obtain interface elements for parent classes.

Hiding interface elements

`.hideInterface(x, field)` determines whether certain UI elements should be hidden from the user. The required arguments are:

- `x`, an instance of a [Panel](#) class.
- `field`, string containing the name of a slot of `x`.

Methods for this generic are expected to return a logical scalar indicating whether the interface element corresponding to `field` should be hidden from the user. This is useful for hiding UI elements that cannot be changed or have no effect, especially in highly specialized subclasses where some concepts in the parent class may no longer be relevant. (The alternative would be to reimplement all of the parent's `.defineInterface` method just to omit a handful of UI elements!)

It is the developer's responsibility to call [callNextMethod](#) to hide the same interface elements as parent classes. This is not strictly required if one wishes to expose previously hidden elements.

Author(s)

Aaron Lun

interface-wrappers	iSEE <i>UI element wrappers</i>
--------------------	--

Description

Wrapper functions to create the standard **shiny** user interface elements, accompanied by an optional help icon that opens an interactive tour describing the purpose of the element. Also responds to requests to hide a particular element via [.hideInterface](#).

Usage

```
.selectInput.iSEE(x, field, label, ..., help = TRUE)
.selectizeInput.iSEE(x, field, label, ..., help = TRUE)
.checkboxInput.iSEE(x, field, label, ..., help = TRUE)
.checkboxGroupInput.iSEE(x, field, label, ..., help = TRUE)
.sliderInput.iSEE(x, field, label, ..., help = TRUE)
.numericInput.iSEE(x, field, label, ..., help = TRUE)
.radioButton.iSEE(x, field, label, ..., help = TRUE)
```

Arguments

<code>x</code>	A Panel object for which to construct an interface element.
<code>field</code>	String containing the name of the parameter controlled by the interface element.
<code>label</code>	String specifying the label to be shown.
<code>...</code>	Further arguments to be passed to the corresponding shiny function.
<code>help</code>	Logical scalar indicating whether a help icon should be added to the label.

Value

The output of `FUN(id, ...)` is returned where `FUN` is set the corresponding **shiny** function, e.g., [selectInput](#) for `.selectInput.iSEE`. `id` is defined by concatenating `.getEncodedName(x)` and `field` (separated by an underscore).

If `.hideInterface(x, field)` is `TRUE`, the output is wrapped inside a [hidden](#) call.

Author(s)

Aaron Lun

iSEE

iSEE: interactive SummarizedExperiment Explorer

Description

Interactive and reproducible visualization of data contained in a [SummarizedExperiment](#) object, using a Shiny interface.

Usage

```
iSEE(
  se,
  initial = NULL,
  extra = NULL,
  colormap = ExperimentColorMap(),
  landingPage = createLandingPage(),
  tour = NULL,
  appTitle = NULL,
  tabTitle = NULL,
  runLocal = TRUE,
  voice = FALSE,
  bugs = FALSE,
  saveState = NULL,
  ...
)
```

Arguments

se	A SummarizedExperiment object, ideally with named assays. If missing, an app is launched with a landing page generated by the <code>landingPage</code> argument.
initial	A list of Panel objects specifying the initial state of the app. The order of panels determines the sequence in which they are laid out in the interface. Defaults to one instance of each panel class available from iSEE .
extra	A list of additional Panel objects that might be added after the app has started. Defaults to one instance of each panel class available from iSEE .
colormap	An ExperimentColorMap object that defines custom colormaps to apply to individual assays, <code>colData</code> and <code>rowData</code> covariates.
landingPage	A function that renders a landing page when iSEE is started without any specified se. Ignored if se is supplied.
tour	A data.frame with the content of the interactive tour to be displayed after starting up the app. Ignored if se is not supplied.
appTitle	A string indicating the title to be displayed in the app. If not provided, the app displays the version info of iSEE .
tabTitle	A string indicating the title to be displayed in the browser tab. If not provided, the tab is named "iSEE".
runLocal	A logical indicating whether the app is to be run locally or remotely on a server, which determines how documentation will be accessed.
voice	A logical indicating whether the voice recognition should be enabled.
bugs	Set to TRUE to enable the bugs Easter egg. Alternatively, a named numeric vector control the respective number of each bug type (e.g., <code>c(bugs=3L, spiders=1L)</code>).
saveState	A function that accepts a single argument containing the current application state and saves it to some appropriate location.
...	Further arguments to pass to shinyApp .

Details

Configuring the initial state of the app is as easy as passing a list of [Panel](#) objects to `initial`. Each element represents one panel and is typically constructed with a command like `ReducedDimensionPlot()`. Panels are filled from left to right in a row-wise manner depending on the available width. Each panel can be easily customized by modifying the parameters in each object.

The `extra` argument should specify [Panel](#) classes that might not be shown during initialization but can be added interactively by the user after the app has started. The first instance of each new class in `extra` will be used as a template when the user adds a new panel of that class. Note that `initial` will automatically be appended to `extra` to form the final set of available panels, so it is not strictly necessary to re-specify instances of those initial panels in `extra`. (unless we want the parameters of newly created panels to be different from those at initialization).

Value

A Shiny app object is returned for interactive data exploration of se, either by simply printing the object or by explicitly running it with [runApp](#).

Setting up a tour

The `tour` argument allows users to specify a custom tour to walk their audience through various panels. This is useful for describing different aspects of the dataset and highlighting interesting points in an interactive manner.

We use the format expected by the `rintrojs` package - see <https://github.com/carlganz/rintrojs#usage> for more information. There should be two columns, `element` and `intro`, with the former describing the element to highlight and the latter providing some descriptive text. The `defaultTour` also provides the default tour that is used in the Examples below.

Creating a landing page

If `se` is not supplied, a landing page is generated that allows users to upload their own RDS file to initialize the app. By default, the maximum request size for file uploads defaults to 5MB (<https://shiny.rstudio.com/reference/shiny/0.14/shiny-options.html>). To raise the limit (e.g., 50MB), run `options(shiny.maxRequestSize=50*1024^2)`.

The `landingPage` argument can be used to alter the landing page, see `createLandingPage` for more details. This is useful for creating front-ends that can retrieve `SummarizedExperiments` from a database on demand for interactive visualization.

Saving application state

If users want to record the application state, they can download an RDS file containing a list with the entries:

- `memory`, a list of `Panel` objects containing the current state of the application. This can be directly re-used as the `initial` argument in a subsequent `iSEE` call.
- `se`, the `SummarizedExperiment` object of interest. This is optional and may not be present in the list, depending on the user specifications.
- `colormap`, the `ExperimentColorMap` object being used. This is optional and may not be present in the list, depending on the user specifications.

We can also provide a custom function in `saveState` that accepts a single argument containing this list. This is most useful when `iSEE` is deployed in an enterprise environment where sessions can be saved in a persistent location; combined with a suitable `landingPage` specification, this allows users to easily reload sessions of interest. The idea is very similar to Shiny bookmarks but is more customizable and can be used in conjunction with URL-based bookmarking.

References

Rue-Albrecht K, Marini F, Soneson C, Lun ATL. iSEE: Interactive SummarizedExperiment Explorer *F1000Research* 7.

Javascript code for bugs was based on <https://github.com/Auz/Bug>.

Examples

```
library(scRNAseq)

# Example data ----
```

```
sce <- ReprocessedAllenData(assays="tophat_counts")
class(sce)

library(scater)
sce <- logNormCounts(sce, exprs_values="tophat_counts")

sce <- runPCA(sce, ncomponents=4)
sce <- runTSNE(sce)
rowData(sce)$ave_count <- rowMeans(assay(sce, "tophat_counts"))
rowData(sce)$n_cells <- rowSums(assay(sce, "tophat_counts") > 0)
sce

# launch the app itself ----

app <- iSEE(sce)
if (interactive()) {
  shiny::runApp(app, port=1234)
}
```

iSEE-pkg*iSEE: interactive SummarizedExperiment/SingleCellExperiment Explorer*

Description

iSEE is a Bioconductor package that provides an interactive Shiny-based graphical user interface for exploring data stored in SummarizedExperiment objects, including row- and column-level metadata. Particular attention is given to single-cell data in a SingleCellExperiment object with visualization of dimensionality reduction results, e.g., from principal components analysis (PCA) or t-distributed stochastic neighbour embedding (t-SNE)

Author(s)

Aaron Lun <infinite.monkeys.with.keyboards@gmail.com>

Charlotte Soneson <charlottesoneson@gmail.com>

Federico Marini <marinif@uni-mainz.de>

Kevin Rue-Albrecht <kevinrue67@gmail.com>

See Also

Useful links:

- <https://isee.github.io/iSEE/>
- Report bugs at <https://github.com/iSEE/iSEE/issues>

iSEEOptions

*Global iSEE options***Description**

Get or set global values that are used by relevant panels during construction and application initialization. This has been deprecated in favor of [panelDefaults](#) (for options that apply during [Panel](#) construction) and [registerAppOptions](#) (for options that apply during application runtime).

Usage

```
iSEEOptions
```

Format

An object of class `list` of length 4.

Commands

```
str(iSEEOptions$get())
```

 will show the default values for all options.

```
iSEEOptions$set(name=value)
```

 will set the named option to value.

```
iSEEOptions$restore()
```

 will reset the global options to the package default values.
Available options

`point.color` Default color of data points in `DotPlot` panels (character).

`point.size` Default size of data points in `DotPlot` panels (numeric).

`point.alpha` Default alpha level controlling transparency of data points in `DotPlot` panels (numeric).

`downsample` Enable visual downsampling in `DotPlot` panels (logical).

`downsample.resolution` Resolution of the visual downsampling, if active (numeric).

`selected.color` Color of selected data points in `DotPlot` panels (character).

`selected.alpha` Alpha level controlling transparency of data points *not* selected in `DotPlot` panels (numeric).

`selection.dynamic.single` Toggle dynamic single selections for all panels (logical).

`selection.dynamic.multiple` Toggle dynamic multiple selections for all panels (logical).

`contour.color` Color of the 2d density estimation contour in `DotPlot` panels (character).

`font.size` Global multiplier controlling the magnification of plot title and text elements in `DotPlot` panels (numeric).

`legend.position` Position of the legend in `DotPlot` and `ComplexHeatmapPlot` panels (one of "Bottom", "Right", or "None").

`legend.direction` Position of the legend in `DotPlot` and `ComplexHeatmapPlot` panels (one of "Horizontal", "Vertical").

- `panel.width` Default panel grid width (must be between 1 and 12).
- `panel.height` Default panel height (in pixels).
- `panel.color` Named character vector of colors. The names of the vector should be set to the name of class to be overridden; if a class is not named here, its default color is used. It is highly recommended to define colors as hex color codes (e.g., `"#1e90ff"`), for full compatibility with both HTML elements and R plots.
- `color.maxlevels` Maximum number of levels for a categorical variable used for coloring. Variables with more levels are coerced to numeric to avoid problems with an overly-large legend. Defaults to 24.
- `factor.maxlevels` Maximum number of levels for a categorical variable to be used anywhere in the app. Variables with more levels are coerced to numeric to avoid rendering delays. Defaults to 100.
- `assay` Character vector of assay names to use if available, in order of preference.
- `RowTable.select.details` A function that takes a string containing the name of a feature (i.e., the current selection in the [RowTable](#)) and returns a HTML element with more details.
- `ColumnTable.select.details` A function that takes a string containing the name of a sample (i.e., the current selection in the [ColumnTable](#)) and returns a HTML element with more details.

Author(s)

Kevin Rue-Albrecht

Examples

```
iSEEOptions$get('downsample'); iSEEOptions$get('selected.color')
```

`jitterSquarePoints` *Jitter points for categorical variables*

Description

Add quasi-random jitter on the x-axis for violin plots when the x-axis variable is categorical. Add random jitter within a rectangular area for square plots when both x- and y-axis variables are categorical.

Usage

```
jitterSquarePoints(X, Y, grouping = NULL)

jitterViolinPoints(X, Y, grouping = NULL, ...)
```

Arguments

<code>X</code>	A factor corresponding to a categorical variable.
<code>Y</code>	A numeric vector of the same length as <code>X</code> for <code>jitterViolinPoints</code> , or a factor of the same length as <code>X</code> for <code>jitterSquarePoints</code> .
<code>grouping</code>	A named list of factors of the same length as <code>X</code> , specifying how elements should be grouped.
<code>...</code>	Further arguments to be passed to <code>offsetX</code> .

Details

The `jitterViolinPoints` function calls `offsetX` to obtain quasi-random jittered x-axis values. This reflects the area occupied by a violin plot, though some tuning of arguments in `...` may be required to get an exact match.

The `jitterSquarePoints` function will uniformly jitter points on both the x- and y-axes. The jitter area is a square with area proportional to the frequency of the paired levels in `X` and `Y`. If either factor only has one level, the jitter area becomes a rectangle that can be interpreted as a bar plot.

If `grouping` is specified, the values corresponding to each point defines a single combination of levels. Both functions will then perform jittering separately within each unique combination of levels. This is useful for obtaining appropriate jittering when points are split by group, e.g., during faceting.

If `grouping!=NULL` for `jitterSquarePoints` the statistics in the returned summary data.frame will be stratified by unique combinations of levels. To avoid clashes with existing fields, the names in `grouping` should not be `"X"`, `"Y"`, `"Freq"`, `"XWidth"` or `"YWidth"`.

Value

For `jitterViolinPoints`, a numeric vector is returned containing the jittered x-axis coordinates for all points.

For `jitterSquarePoints`, a list is returned with numeric vectors `X` and `Y`, containing jittered coordinates on the x- and y-axes respectively for all points; and `summary`, a data.frame of frequencies and side lengths for each unique pairing of `X/Y` levels.

Author(s)

Aaron Lun

Examples

```
X <- factor(sample(LETTERS[1:4], 100, replace=TRUE))
Y <- rnorm(100)
(out1 <- jitterViolinPoints(X=X, Y=Y))

Y2 <- factor(sample(letters[1:3], 100, replace=TRUE))
(out2 <- jitterSquarePoints(X=X, Y=Y2))

grouped <- sample(5, 100, replace=TRUE)
(out3 <- jitterViolinPoints(X=X, Y=Y, grouping=list(FacetRow=grouped)))
(out4 <- jitterSquarePoints(X=X, Y=Y2, grouping=list(FacetRow=grouped)))
```

lassoPoints	<i>Find rows of data within a closed lasso</i>
-------------	--

Description

Identify the rows of a data.frame lying within a closed lasso polygon, analogous to [brushedPoints](#).

Usage

```
lassoPoints(df, lasso)
```

Arguments

df	A data.frame from which to select rows.
lasso	A list containing data from a lasso.

Details

This function uses [in.out](#) from the **mgecv** package to identify points within a polygon. This involves a boundary crossing algorithm that may not be robust in the presence of complex polygons with intersecting edges.

Value

A subset of rows from df with coordinates lying within lasso.

Author(s)

Aaron Lun

See Also

[brushedPoints](#)

Examples

```
lasso <- list(coord=rbind(c(0, 0), c(0.5, 0), c(0, 0.5), c(0, 0)),
  closed=TRUE, mapping=list(x="X", y="Y"))
values <- data.frame(X=runif(100), Y=runif(100),
  row.names=sprintf("VALUE_%i", seq_len(100)))
lassoPoints(values, lasso)

# With faceting information:
lasso <- list(coord=rbind(c(0, 0), c(0.5, 0), c(0, 0.5), c(0, 0)),
  panelvar1="A", panelvar2="B", closed=TRUE,
  mapping=list(x="X", y="Y",
  panelvar1="FacetRow", panelvar2="FacetColumn"))
values <- data.frame(X=runif(100), Y=runif(100),
  FacetRow=sample(LETTERS[1:2], 100, replace=TRUE),
```

```
FacetColumn=sample(LETTERS[1:4], 100, replace=TRUE),
row.names=sprintf("VALUE_%i", seq_len(100)))
lassoPoints(values, lasso)
```

manage_commands*Manage commands to be evaluated*

Description

Functions to manage commands to be evaluated.

Usage

```
.textEval(cmd, envir)
```

Arguments

<code>cmd</code>	A character vector containing commands to be executed.
<code>envir</code>	An environment in which to execute the commands.

Value

`.textEval` returns the output of `eval(parse(text=cmd), envir)`, unless `cmd` is empty in which case it returns `NULL`.

Author(s)

Aaron Lun, Kevin Rue-Albrecht

Examples

```
myenv <- new.env()
myenv$x <- "Hello world!"
.textEval("print(x)", myenv)
```

metadata-plot-generics*Generics for row/column metadata plots*

Description

These generics allow subclasses to refine the choices of allowable variables on the x- and y-axes of a [ColumnDataPlot](#) or [RowDataPlot](#). This is most useful for restricting the visualization to a subset of variables, e.g., only taking log-fold changes in a y-axis of a MA plot.

Allowable y-axis choices

`.allowableYAxisChoices(x, se)` takes `x`, a [Panel](#) instance; and `se`, the [SummarizedExperiment](#) object. It is expected to return a character vector containing the names of acceptable variables to show on the y-axis. For [ColumnDataPlots](#), these should be a subset of the variables in `colData(se)`, while for [RowDataPlots](#), these should be a subset of the variables in `rowData(se)`.

Given a constant `se`, the output of this function should be constant for all instances of the same panel class. As such, it is a good idea to make use of information precomputed by `.cacheCommonInfo`. For example, `.cacheCommonInfo,ColumnDotPlot-method` will add vectors specifying whether a variable in the `colData` is valid and discrete or continuous, which can be intersected with any additional requirements in a subclass's method for this generic.

This generic is called by `.defineDataInterface` for [ColumnDataPlots](#) and [RowDataPlots](#). Thus, developers wanting to restrict those choices for subclasses can simply specialize `.allowableYAxisChoices` rather than reimplementing `.defineDataInterface`.

Allowable x-axis choices

`.allowableXAxisChoices(x, se)` is the same as above but controls the variables that can be shown on the x-axis. This need not return the same subset of variables as `.allowableYAxisChoices`. However, again, the output of this function should be constant for all instances of the same class and a constant `se`.

Author(s)

Aaron Lun

multi-select-generics *Generics for controlling multiple selections*

Description

A panel can create a multiple selection on either the rows or columns and transmit this selection to another panel to affect its set of displayed points. For example, users can brush on a [DotPlots](#) to select a set of points, and then the panel can transmit the identities of those points to another panel for highlighting.

This suite of generics controls the behavior of these multiple selections. In all of the code chunks shown below, `x` is assumed to be an instance of the [Panel](#) class.

Possibility of selection

`.isBrushable(x)` should return a logical specifying whether the panel supports selection using a Shiny brush or lasso waypoints. The output should be constant for all instances of `x` and is used to govern the reporting of multiple selections in the code tracker.

Specifying the dimension

`.multiSelectionDimension(x)` should return a string specifying whether the selection contains rows ("row"), columns ("column") or if the Panel in `x` does not perform multiple selections at all ("none"). The output should be constant for all instances of `x` and is used to govern the interface choices for the selection parameters.

Specifying the active selection

`.multiSelectionActive(x)` should return some structure containing all parameters required to identify all points in the active multiple selection of `x`. For example, the `DotPlot` method for this generic would return the contents of the `BrushData` slot, usually a list containing a Shiny brush or lasso waypoints for `DotPlot` classes. If `.multiSelectionActive(x)` returns `NULL`, `x` is assumed to have no active multiple selection.

The active selection is considered to be the one that can be directly changed by the user, as compared to saved selections that are not modifiable (other than being deleted on a first-in-last-out basis). This generic is primarily used to bundle up selection parameters to be stored in the `SelectionHistory` slot when the user saves the current active selection.

Evaluating the selection

`.multiSelectionCommands(x, index)` is expected to return a character vector of commands to generate a character vector of row or column names in the desired multiple selection of `x`. If `index=NA`, the desired selection is the currently active one; developers can assume that `.multiSelectionActive(x)` returns a non-`NULL` value in this case. Otherwise, for an integer `index`, it refers to the corresponding saved selection in the `SelectionHistory`.

The commands will be evaluated in an environment containing:

- `select`, a variable of the same type as returned by `.multiSelectionActive(x)`. This will contain the active selection if `index=NA` and one of the saved selections otherwise. For example, for `DotPlots`, `select` will be either a Shiny brush or a lasso structure.
- `contents`, some arbitrary content saved by the rendering expression in `.renderOutput(x)`. This is most often a `data.frame` but can be anything as long as `.multiSelectionCommands` knows how to process it. For example, a `data.frame` of coordinates is stored by `DotPlots` to identify the points selected by a brush/lasso.
- `se`, the `SummarizedExperiment` object containing the current dataset.

The output commands are expected to produce a character vector named `selected` in the evaluation environment. All other variables generated by the commands should be prefixed with `.` to avoid name clashes.

Determining the available points for selection

`.multiSelectionAvailable(x, contents)` is expected to return an integer scalar specifying the number of points available for selection in the the current instance of the panel `x`. The `contents` field in the output of `.generateOutput` is passed to the `contents` argument of this generic.

The default method for this generic returns `nrow(contents)` for all `Panel` subclasses, assuming that `contents` is a `data.frame` where each row represents a point. If not, this method needs to be specialized in order to return an accurate total of available points, which is ultimately used to compute the percentage selected in the multiple selection information panels.

Destroying selections

`.multiSelectionClear(x)` should return `x` after removing the active selection, i.e., so that nothing is selected. This is used internally to remove multiple selections that do not make sense after protected parameters have changed. For example, a brush or lasso made on a PCA plot in `Reduced-DimensionPlots` would not make sense after switching to t-SNE coordinates, so the application will automatically erase those selections to avoid misleading conclusions.

Responding to selections

These generics control how `x` responds to a transmitted multiple selection, not how `x` itself transmits selections.

`.multiSelectionRestricted(x)` should return a logical scalar indicating whether `x`'s displayed contents will be restricted to the selection transmitted from *another panel*. This is used to determine whether child panels of `x` need to be re-rendered when `x`'s transmitter changes its multiple selection. For example, the method for `RowDotPlots` and `ColumnDotPlots` would return `TRUE` if `RowSelectionRestrict=TRUE` or `ColumnSelectionRestrict=TRUE`, respectively. Otherwise, it would be `FALSE` as the transmitted selection is only used for aesthetics, not for changing the identity of the displayed points.

`.multiSelectionInvalidated(x)` should return a logical scalar indicating whether a transmission of a multiple selection to `x` invalidates `x`'s own existing selections. This should only be `TRUE` in special circumstances, e.g., if receipt of a new multiple selection causes recalculation of coordinates in a `DotPlot`.

`.multiSelectionResponsive(x, dim)` should return a logical scalar indicating whether `x` is responsive to an incoming multiple selection on dimension `dim`. `dim` is equal to "row" if the multiple selection is of rows, otherwise it is equal to "column" for a multiple selection of columns. For example, the method for `ComplexHeatmapPlot` would return `TRUE` when an incoming selection originates from a row-oriented panel and `CustomRows=FALSE`. Otherwise, it would be `FALSE` as the dimension of the transmitted selection is dismissed by the options of the child panel.

Author(s)

Aaron Lun

`multiSelectionToFactor`*Convert multiple selections into a factor*

Description

Convert multiple selection information into a factor, typically for use as a covariate or for coloring.

Usage

```
multiSelectionToFactor(selected, all.names)
```

Arguments

<code>selected</code>	A named list of character vectors, containing the names of selected observations for different selections. Vectors for different selections may overlap.
<code>all.names</code>	Character vector of all observations.

Value

A factor containing the set(s) to which each observation is assigned. Multiple sets are encoded as comma-separated strings. Unselected observations are listed as "unselected".

Author(s)

Aaron Lun

Examples

```
multiSelectionToFactor(list(active=c("A", "B"),
  saved1=c("B", "C"), saved2=c("D", "E", "F")),
  all.names=LETTERS[1:10])
```

`observer-generics`*Generic for the panel observers*

Description

The workhorse generic for defining the Shiny observers for a given panel, along with recommendations on its implementation.

Creating parameter observers

In `.createObservers(x, se, input, session, pObjects, rObjects)`, the required arguments are:

- `x`, an instance of a [Panel](#) class.
- `se`, a [SummarizedExperiment](#) object containing the current dataset. This can be assumed to have been produced by running `.refineParameters(x, se)`.
- `input`, the Shiny input object from the server function.
- `session`, the Shiny session object from the server function.
- `pObjects`, an environment containing global parameters generated in the [iSEE](#) app.
- `rObjects`, a reactive list of values generated in the [iSEE](#) app.

Methods for this generic are expected to set up all observers required to respond to changes in the interface elements set up by `.defineInterface`. Recall that each interface element has an ID of the form of `PANEL_SLOT`, where `PANEL` is the panel name (from `.getEncodedName`) and `SLOT` is the name of the slot modified by the interface element. Thus, observers should respond to changes in those elements in `input`. The return value of this generic is not used; only the side-effect of observer set-up is relevant.

It is the developer's responsibility to call `callNextMethod` to set up the observers required by the parent class. This is best done by calling `callNextMethod` at the top of the method before defining up additional observers. Each parent class should implement observers for its slots, so it is usually only necessary to implement observers for any newly added slots in a particular class.

Modifying the memory

Consider an observer for an interface element that modifies a slot of `x`. The code within this observer is expected to modify the “memory” of the app state in `pObjects`, via:

```
new_value <- input[[paste0(PANEL, "_", SLOT)]]
pObjects$memory[[PANEL]][[SLOT]] <- new_value
```

This enables [iSEE](#) to keep a record of the current state of the application. In fact, any changes must go through `pObjects$memory` before they change the output in `.renderOutput`; there is no direct interaction between `input` and `output` in this framework.

We suggest using `.createProtectedParameterObservers` and `.createUnprotectedParameterObservers`, which create simple observers that update the memory in response to changes in the UI elements. For handling selectize elements filled with server-side row/column names, we can use `.createCustomDimnamesModalObserver`.

Developers should not attempt to modify `x` in any observer expression. This value does not have pass-by-reference semantics and any changes will not propagate to other parts of the application. Rather, modifications should occur to the version of `x` in `pObjects$memory`, as described in the code chunk above.

Triggering re-rendering

To trigger re-rendering of an output, observers should call `.requestUpdate(PANEL, rObjects)` where `PANEL` is the name of the current panel. This will request a re-rendering of the output with no additional side effects and is most useful for responding to aesthetic parameters.

In some cases, changes to some parameters may invalidate existing multiple selections, e.g., brushes and lassos are no longer valid if the variable on the axes are altered. Observers responding to such changes should instead call `.requestCleanUpdate(PANEL, pObjects, rObjects)`, which will destroy all existing selections in order to avoid misleading conclusions.

Author(s)

Aaron Lun

output-generics

Generics for Panel outputs

Description

An overview of the generics for defining the panel outputs, along with recommendations on their implementation.

Defining the output element

`.defineOutput(x)` defines the output element of the panel (e.g., a plot or table widget), given an instance of a [Panel](#) subclass in `x`.

Methods for this generic are expected to return a HTML element containing the visual output of the panel, such as the return value of `plotOutput` or `dataTableOutput`. This element will be shown in the **iSEE** interface above the parameter boxes for `x`. Multiple elements can be returned via a `tagList`.

The IDs of the output elements are expected to be prefixed with the panel name from `.getEncodedName(x)` and an underscore, e.g., "ReducedDimensionPlot1_someOutput". One of the output elements may simply have the ID set to PANEL alone; this is usually the case for simple panels with one primary output like a [DotPlot](#).

Defining the rendered output

`.renderOutput(x, se, ..., output, pObjects, rObjects)` will create an expression to render the panel's output. The following arguments are required:

- `x`, an instance of a [Panel](#) class.
- `se`, a [SummarizedExperiment](#) object containing the current dataset.
- `...`, further arguments that may be used by specific methods.
- `output`, the Shiny output object from the server function.
- `pObjects`, an environment containing global parameters generated in the **iSEE** app.
- `rObjects`, a reactive list of values generated in the **iSEE** app.

It is expected to attach one or more reactive expressions to `output` to render the output element(s) defined by `.defineOutput`. This is typically done by calling **shiny** rendering functions like `renderPlot` or the most appropriate equivalent for the panel's output. The return value of this generic is not used; only the side-effect of the reactive output set-up is relevant.

The rendering expression inside the chosen rendering function is expected to:

1. Call `force(rObjects[[PANEL]])`, where `PANEL` is the output of `.getEncodedName(x)`. This ensures that the output is rerendered upon requesting changes in `.requestUpdate`.
2. Call `.generateOutput` to generate the output content to be rendered.
3. Fill `pObjects$contents[[PANEL]]` with some content related to the displayed output that allows cross-referencing with single/multiple selection structures. This will be used in other generics like `.multiSelectionCommands` and `.singleSelectionValue` to determine the identity of the selected point(s). As a result, it is only strictly necessary if the panel is a potential transmitter, as determined by the return value of `.multiSelectionDimension`.
4. Fill `pObjects$commands[[PANEL]]` with a character vector of commands required to produce the displayed output. This should minimally include the commands required to generate `pObjects$contents[[PANEL]]`; for plotting panels, the vector should also include code to create the plot.
5. Fill `pObjects$varname[[PANEL]]` with a string containing the R expression in `pObjects$commands[[PANEL]]` that holds the contents stored in `pObjects$contents[[PANEL]]`. This is used for code reporting, and again, is only strictly necessary if the panel is a potential transmitter.

We strongly recommend calling `.retrieveOutput` within the rendering expression, which will automatically perform all of the tasks above, rather than calling `.generateOutput` manually. By doing so, the only extra work required of the rendering expression is to actually render the output (e.g., by printing a `ggplot` object). Of course, the rendering expression must itself be encapsulated by an appropriate rendering function assigned to output.

Developers should not attempt to modify `x` in any rendering expression. This does not have pass-by-reference semantics and any changes will not propagate to other parts of the application. Similarly, the rendering expression should treat `pObjects$memory` as read-only. Any adjustment of parameters should be handled elsewhere, e.g., by the observer expressions in `.createObservers`.

Generating content

`.generateOutput(x, se, all_memory, all_contents)` actually generates the panel's output to be used in the rendering expression. The following arguments are required:

- `x`, an instance of a `Panel` class.
- `se`, a `SummarizedExperiment` object containing the current dataset.
- `all_memory`, a named list containing `Panel` objects parameterizing the current state of the app.
- `all_contents`, a named list containing the contents of each panel.

Methods for this generic should return a list containing:

- `contents`, some arbitrary content for the panel (usually a `data.frame`). The values therein are used by `.multiSelectionCommands` to determine the multiple row/column selection in `x` to be transmitted to other (child) panels. The app will ensure that the `pObjects$contents` of each panel is populated before attempting to render their children. `contents` may be set to `NULL` if `x` does not transmit, i.e., `.multiSelectionDimension` returns `"none"`.
- `commands`, a list of character vectors of R commands that, when executed, produces the contents of the panel and any displayed output (e.g., a `ggplot` object). Developers should write these commands as if the evaluation environment only contains the `SummarizedExperiment` `se` and `ExperimentColorMap` `colormap`. It may also contain `col_selected`, if a multiple column selection is being transmitted to `x`; and possibly `row_selected`, if a multiple row selection is being transmitted to `x`.

- `varname`, a string specifying the name of the variable in commands used to generate contents. This is used to fulfill code tracking obligations. If the current panel is not a transmitter, this may be set to `NULL` instead.

The output list may contain any number of other fields that can be used by `.renderOutput` but are otherwise ignored.

We suggest implementing this method using `eval(parse(text=...))` calls, which enables easy construction and evaluation of the commands and contents at the same time. A convenient wrapper for this call is provided by the `.textEval` utility.

The `all_memory` and `all_contents` arguments are provided for the sole purpose of determining what multiple selections are being received by `x`. We strongly recommend passing them onto `.processMultiSelections` to do the heavy lifting. It would be unusual and inadvisable to use these arguments for any other information sharing across panels.

Exporting content

`.exportOutput(x, se, all_memory, all_contents)` converts the panel output into a downloadable form. The following arguments are required:

- `x`, an instance of a `Panel` class.
- `se`, a `SummarizedExperiment` object containing the current dataset.
- `all_memory`, a named list containing `Panel` objects parameterizing the current state of the app.
- `all_contents`, a named list containing the contents of each panel.

Methods for this generic should generate appropriate files containing the content of `x`. (For example, plots may create PDFs while tables may create CSV files.) All files should be created in the working directory at the time of the function call, possibly in further subdirectories. Each file name should be prefixed with the `.getEncodedName`. The method itself should return a character vector containing *relative* paths to all newly created files.

To implement this method, we suggest simply passing all arguments onto `.generateOutput` and then converting the output into an appropriate file.

Author(s)

Aaron Lun

Panel-class

The Panel virtual class

Description

The `Panel` is a virtual base class for all **iSEE** panels. It provides slots and methods to control the height and width of each panel, as well as functionality to control the choice of “transmitting” panels from which to receive a multiple row/column selection.

Slot overview

The following slots are relevant to panel organization:

- `PanelId`, an integer scalar specifying the identifier for the panel. This should be unique across panels of the same concrete class.
- `PanelWidth`, an integer scalar specifying the width of the panel. Bootstrap coordinates are used so this value should lie between 2 and 12; defaults to 4 in `getPanelDefault`.
- `PanelHeight`, an integer scalar specifying the height of the panel in pixels. This is expected to lie between 400 and 1000; defaults to 500 in `getPanelDefault`.

The following slots are relevant to *receiving* a multiple selection on the rows:

- `RowSelectionSource`, a string specifying the name of the transmitting panel from which to receive a multiple row selection (e.g., `"RowDataPlot1"`). Defaults to `"---"`.
- `RowSelectionDynamicSource`, a logical scalar indicating whether `x` should dynamically change its selection source for multiple row selections. Defaults to `FALSE` in `getPanelDefault`.
- `RowSelectionRestrict`, a logical scalar indicating whether the display of `x` should be restricted to the rows in the multiple selection received from a transmitting panel. Defaults to `FALSE`.

The following slots are relevant to *receiving* a multiple selection on the columns:

- `ColumnSelectionSource`, a string specifying the name of the transmitting panel from which to receive a multiple column selection (e.g., `"ColumnDataPlot1"`). Defaults to `"---"`.
- `ColumnSelectionDynamicSource`, a logical scalar indicating whether `x` should dynamically change its selection source for multiple column selections. Defaults to `FALSE` in `getPanelDefault`.
- `ColumnSelectionRestrict`, a logical scalar indicating whether the display of `x` should be restricted to the columns in the multiple selection received from a transmitting panel. Defaults to `FALSE`.

There are also the following miscellaneous slots:

- `SelectionBoxOpen`, a logical scalar indicating whether the selection parameter box should be open at initialization. Defaults to `FALSE`.
- `SelectionHistory`, a list of arbitrary elements that contain parameters for saved multiple selections. Each element of this list corresponds to one saved selection in the current panel. Defaults to an empty list.
- `VersionInfo`, a named list of `package_version` objects specifying the versions of packages used to create a given `Panel` instance. This information is used to inform `updateObject` of any updates that need to be applied. By default, it is filled with a single `"iSEE"` entry containing the current version of `iSEE`.

Getting and setting slots

In all of the following code chunks, `x` is an instance of a `Panel`, and `i` is a string containing the slot name:

- `x[[i]]` returns the value of a slot named `i`.
- `x[[i]] <- value` modifies `x` so that the value in slot `i` is replaced with `value`.
- `show(x)` will print a summary of all (non-hidden) slots and their values.

Supported methods

In the following code snippets, `x` is an instance of a `ColumnDotPlot` class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.refineParameters(x, se)` calls `updateObject(x)`. If `x` is up to date, this operation is a no-op and returns `x` without modification.
- `.cacheCommonInfo(x, se)` is a no-op, returning `se` without modification.

For defining the interface:

- `.defineInterface(x, se, select_info)` will return a list of collapsible boxes for changing data and selection parameters. The data parameter box will be populated based on `.defineDataInterface`.
- `.defineDataInterface(x, se, select_info)` will return an empty list.
- `.hideInterface(x, field)` will always return `FALSE`.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` will add observers to respond to changes in multiple selection options. It will also call `.singleSelectionSlots(x)` to set up observers for responding to transmitted single selections.
- `.renderOutput(x, se, output, pObjects, rObjects)` will add elements to output for rendering the information textboxes at the bottom of each panel. Each panel should specialize this method to add rendering expressions for the actual output (e.g., plots, tables), followed by a `callNextMethod` to create the textboxes.

For generating output:

- `.exportOutput(x, se, all_memory, all_contents)` is a no-op, i.e., it will return an empty character vector and create no files.

For documentation:

- `.definePanelTour(x)` returns a data.frame containing the selection-related steps of the tour.

For controlling selections:

- `.multiSelectionRestricted(x)` will always return `TRUE`.
- `.multiSelectionDimension(x)` will always return `"none"`.
- `.multiSelectionActive(x)` will always return `NULL`.
- `.multiSelectionClear(x)` will always return `x`.
- `.multiSelectionInvalidated(x)` will always return `FALSE`.
- `.multiSelectionAvailable(x, contents)` will return `nrow(contents)`.
- `.multiSelectionResponsive(x, dim)` will always return `TRUE`.
- `.singleSelectionDimension(x)` will always return `"none"`.
- `.singleSelectionValue(x)` will always return `NULL`.
- `.singleSelectionSlots(x)` will always return an empty list.

Subclass expectations

Subclasses are required to implement methods for:

- [.defineOutput](#)
- [.generateOutput](#)
- [.renderOutput](#)
- [.fullName](#)
- [.panelColor](#)

Subclasses that transmit selections should also implement specialized methods for selection-related parameters listed above.

Author(s)

Aaron Lun

See Also

[DotPlot](#) and [Table](#), for examples of direct subclasses.

panelDefaults

Panel defaults

Description

Get or set default parameters that are used by certain [Panel](#) during their construction. This allows users to easily change the defaults for multiple panels simultaneously without having to manually specify them in the constructor.

Usage

```
panelDefaults(...)
```

```
getPanelDefault(name, error = TRUE)
```

Arguments

...	Named options to set. Alternatively a single named list containing the options to set.
name	String containing the name of the option to retrieve. Alternatively NULL, in which case the current values of all options are returned as a named list.
error	Logical scalar indicating whether an error should be raised if name cannot be found.

Details

All options set by `panelDefaults` will only affect `Panel` construction and have no effect on the behavior of `Panels` that are already constructed. Most options are named after the affected slot in the relevant `Panel` subclass.

For general `Panels`:

- `PanelWidth`, defaults to 4.
- `PanelHeight`, defaults to 500.

For `DotPlots`:

- `ColorByDefaultColor`, defaults to "black".
- `PointSize`, defaults to 1.
- `PointAlpha`, defaults to 1.
- `Downsample`, defaults to FALSE.
- `DownsampleResolution`, defaults to 200.
- `SelectionAlpha`, defaults to 0.1.
- `ContourColor`, defaults to "blue".
- `FontSize`, defaults to 1.
- `LegendPointSize`, defaults to 1.

For `RowDotPlots`:

- `TooltipRowData`, defaults to `character(0)`.

For `ColumnDotPlots`:

- `TooltipColumnData`, defaults to `character(0)`.

For `ComplexHeatmapPlots`:

- `LegendDirection`, defaults to "Horizontal".

A few options affect multiple subclasses that independently define the same slot:

- `LegendPosition`, defaults to "Bottom". Affects `DotPlots` and `ComplexHeatmapPlots`.
- `Assay`, defaults to "logcounts". Affects `FeatureAssayPlots`, `SampleAssayPlots` and `ComplexHeatmapPlots`.

A few options are not named after any particular slot as they affect different slots in different subclasses:

- `ColorByNameColor`, defaults to "red". This affects `ColorByFeatureNameColor` in `RowDotPlots` and `ColorBySampleNameColor` in `ColumnDotPlots`.
- `ColorByNameAssay`, defaults to "logcounts". This affects `ColorByFeatureNameAssay` in `RowDotPlots` and `ColorBySampleNameAssay` in `ColumnDotPlots`.
- `SingleSelectionDynamicSource`, defaults to FALSE. This affects `ColorByFeatureDynamicSource`, `ColorBySampleDynamicSource`, `XAxisFeatureDynamicSource`, `YAxisFeatureDynamicSource`, `XAxisSampleDynamicSource` and `YAxisSampleDynamicSource` in the relevant panels.
- `MultipleSelectionDynamicSource`, defaults to FALSE. This affects `RowSelectionDynamicSource` and `ColumnSelectionDynamicSource`.

Value

`panelDefaults` will return a named list of the values of all options. If `...` is non-empty, `panelDefaults` will modify the global options that are used during the constructors for the relevant [Panel](#) classes. (Note that the return value still contains the values *before* the modification is applied.)

`getPanelDefault` will return the current value of the requested option. If `error=TRUE` and `name` is not present, an error is raised; otherwise `NULL` is returned.

For developers

Developers of `Panel` subclasses may add more options to this list, typically by calling `panelDefaults` in the `.onLoad` expressions of the package containing the subclass. We recommend prefixing any options with the name of the package in the form of `<PACKAGE>_<OPTION>`, so as to avoid conflicts with other options (in the base classes, or in other downstream packages) that have the same name. Any options added in this manner should correspond to parameters that are already present as slots in the panel class. If this is not the case, consider using [registerAppOptions](#) instead.

Author(s)

Kevin Rue-Albrecht

Examples

```
old <- panelDefaults(Assay="WHEE")
getPanelDefault("Assay")

old <- panelDefaults(Assay="F00", PointSize=5)
getPanelDefault("Assay")
getPanelDefault("PointSize")

# We can also list out all options:
panelDefaults()

# Restoring the previous defaults.
panelDefaults(old)
getPanelDefault("Assay")
getPanelDefault("PointSize")
```

Description

A series of generics for controlling how plotting is performed in [DotPlot](#) panels. [DotPlot](#) subclasses can specialize one or more of them to modify the behavior of [.generateOutput](#).

Generating plotting data

`.generateDotPlotData(x, envir)` sets up the data to use in the [DotPlot](#) plot. The following arguments are required:

- `x`, an instance of a [DotPlot](#) subclass.
- `envir`, the evaluation environment in which the `data.frame` is to be constructed. This can be assumed to have `se`, the [SummarizedExperiment](#) object containing the current dataset; possibly `col_selected`, if a multiple column selection is being transmitted to `x`; and possibly `row_selected`, if a multiple row selection is being transmitted to `x`.

A method for this generic should add a `plot.data` variable in `envir` containing a `data.frame` with columns named "X" and "Y", denoting the variables to show on the x- and y-axes respectively. It should return a list with `commands`, a character vector of commands that produces `plot.data` when evaluated in `envir`; and `labels`, a list of strings containing labels for the x-axis (named "X"), y-axis ("Y") and plot ("title").

Each row of the `plot.data` `data.frame` should correspond to one row or column in the [SummarizedExperiment](#) `envir$se` for [RowDotPlots](#) and [ColumnDotPlots](#) respectively. Note that, even if only a subset of rows/columns in the [SummarizedExperiment](#) are to be shown, there must always be one row in the `data.frame` per row/column of the [SummarizedExperiment](#), and in the same order. All other rows of the `data.frame` should be filled in with NAs rather than omitted entirely. This is necessary for correct interactions with later methods that add other variables to `plot.data`.

Any internal variables that are generated by the commands in `commands` should be prefixed with `.` to avoid potential clashes with reserved variable names in the rest of the application.

This generic is called by `.generateDotPlot` (see below), which is in turn called by `.generateOutput`. The idea is that developers can specialize `.generateDotPlotData` to change the data source for a [DotPlot](#) subclass without needing to reimplement the entirety of `.generateDotPlot`.

Generating the ggplot object

`.generateDotPlot(x, labels, envir)` creates the plot to be shown in the interface. The following arguments are required:

- `x`, an instance of a [DotPlot](#) subclass.
- `labels`, a list of labels corresponding to the columns of `plot.data`. This is typically used to define axis or legend labels in the plot.
- `envir`, the evaluation environment in which the [ggplot](#) object is to be constructed. This can be assumed to have `plot.data`, a `data.frame` of plotting data.

Note that `se`, `row_selected` and `col_selected` will still be present in `envir`, but it is simplest to only use information that has already been incorporated into `plot.data` where possible. This is because the order and number of rows in `plot.data` may have changed since `.generateDotPlotData`.

Methods for this generic should return a list with `plot`, a [ggplot](#) object; and `commands`, a character vector of commands to produce that object when evaluated inside `envir`. This plot will subsequently be the rendered output in `.renderOutput`. Note that `envir` should contain a copy of the plot object in a variable named `dot.plot` - see below for details.

Methods are expected to respond to the presence of various fields in the `plot.data`. The `data.frame` will contain, at the very least, the fields "X" and "Y" from `.generateDotPlotData`. Depending on the parameters of `x`, it may also have the following columns:

- "ColorBy", the values of the covariate to use to color each point.
- "ShapeBy", the values of the covariate to use for shaping each point. This is guaranteed to be categorical.
- "SizeBy", the values of the covariate to use for sizing each point. This is guaranteed to be continuous.
- "FacetRow", the values of the covariate to use to create row facets. This is guaranteed to be categorical.
- "FacetColumn", the values of the covariate to use to create column facets. This is guaranteed to be categorical.
- "SelectBy", a logical field indicating whether the point was included in a multiple selection (i.e., transmitted from another plot with `x` as the receiver). Note that if `RowSelectionRestrict=TRUE` or `ColumnSelectionRestrict=TRUE` (for `RowDotPlots` and `ColumnDotPlots`, respectively), `plot.data` will already have been subsetting to only retain TRUE values of this field.

`envir` may also contain the following variables:

- `plot.data.all`, present when a multiple selection is transmitted to `x` and `RowSelectionRestrict=TRUE` or `ColumnSelectionRestrict=TRUE` (for `RowDotPlots` and `ColumnDotPlots`, respectively). This is a `data.frame` that contains all points prior to subsetting and is useful for defining the boundaries of the plot such that they do not change when the transmitted multiple selection changes.
- `plot.data.pre`, present when downsampling is turned on. This is a `data.frame` that contains all points prior to downsampling (but after subsetting, if that was performed) and is again mainly used to fix the boundaries of the plot.

Developers may wish to use the `.addMultiSelectionPlotCommands` utility to draw brushes and lassos of `x`. Note that this refers to the brushes and lassos made on `x` itself, not those transmitted from another panel to `x`.

It would be very unwise for methods to alter the x-axis, y-axis or faceting values in `plot.data`. This will lead to unintuitive discrepancies between apparent visual selections for a brush/lasso and the actual multiple selection that is evaluated by downstream functions like `.processMultiSelections`.

In certain situations, a `DotPlot` subclass may be able to build off a `ggplot` generated by its parent class. This is easily done by exploiting the fact that methods for this generic are expected to store a copy of their plot `ggplot` object as a `dot.plot` variable in `envir`. A specialized method for the subclass can `callNextMethod()` to populate `envir` with the initial `dot.plot`, and then just construct and execute commands to add more `ggplot2` layers as desired.

This generic is called by `.generateOutput` for `DotPlot` subclasses. Again, the idea here is that developers can specialize `.generateDotPlot` to change the plot aesthetics without needing to reimplement the entirety of `.generateOutput`.

Prioritizing points

`.prioritizeDotPlotData(x, envir)` specifies the "priority" of points to be plotted, where high-priority points are plotted last so that they will not be masked by other points. The following arguments are required:

- `x`, an instance of a [DotPlot](#) subclass.
- `envir`, the evaluation environment in which the [ggplot](#) object is to be constructed. This can be assumed to have `plot.data`, a `data.frame` of plotting data.

Again, note that `se`, `row_selected` and `col_selected` will still be present in `envir`, but it is simplest to only use information that has already been incorporated into `plot.data` where possible. This is because the order and number of rows in `plot.data` may have changed since [.generateDotPlotData](#).

Methods for this generic are expected to generate a `.priority` variable in `envir`, an ordered factor of length equal to `nrow(plot.data)` indicating the priority of each point. They may also generate a `.rescaled` variable, a named numeric vector containing the scaling factor to apply to the downsampling resolution for each level of `.priority`.

The method itself should return a list containing `commands`, a character vector of R commands required to generate these variables; and `rescaled`, a logical scalar indicating whether a `.rescaled` variable was produced.

Points assigned the highest level in `.priority` are regarded as having the highest visual importance. Such points will be shown on top of other points if there are overlaps on the plot, allowing developers to specify that, e.g., DE genes should be shown on top of non-DE genes. Scaling of the resolution enables developers to perform more aggressive downsampling for unimportant points.

Methods for this generic may also return `NULL`, in which case no special action is taken.

This generic is called by `.generateDotPlot`, which is in turn called by [.generateOutput](#). Thus, developers of `DotPlot` subclasses can specialize this generic to change the point priority without needing to reimplement the entirety of `.generateDotPlot`.

Controlling the “None” color scale

In some cases, it is desirable to insert a default scale when `ColorBy="None"`. This is useful for highlighting points in a manner that is integral to the nature of the plot, e.g., up- or down-regulated genes in a MA plot. We provide a few generics to help control which points are highlighted and how they are colored.

`.colorByNoneDotPlotField(x)` expects `x`, an instance of a [DotPlot](#) subclass, and returns a string containing a name of a column in `plot.data` to use for coloring in the `ggplot` mapping. This assumes that the relevant field was added to `plot.data` by a method for [.generateDotPlotData](#).

`.colorByNoneDotPlotScale(x)` expects `x`, an instance of a [DotPlot](#) subclass, and returns a string containing a **ggplot2** `scale_color_*` call, e.g., [scale_color_manual](#). This string should end with a `"+"` operator as additional **ggplot2** layers will be added by [iSEE](#).

This generic is called by `.generateDotPlot`, which is in turn called by [.generateOutput](#). Thus, developers of `DotPlot` subclasses can specialize this generic to change the default color scheme without needing to reimplement the entirety of `.generateDotPlot`.

Author(s)

Kevin “K-pop” Rue-Albrecht, Aaron “A-bomb” Lun

plot-utils*Process faceting choices*

Description

Generate ggplot instructions to facet a plot by row and/or column

Usage

```
.addFacets(x)
```

Arguments

x A single-row DataFrame that contains all the input settings for the current panel.

Value

A string containing a command to define the row and column faceting covariates.

Author(s)

Kevin Rue-Albrecht.

Examples

```
x <- ReducedDimensionPlot(  
  FacetRowBy = "Column data", FacetRowByColData="Covariate_1",  
  FacetColumnBy = "Column data", FacetColumnByColData="Covariate_2")  
.addFacets(x)
```

ReducedDimensionPlot-class*The ReducedDimensionPlot panel*

Description

The ReducedDimensionPlot is a panel class for creating a [ColumnDotPlot](#) where the coordinates of each column/sample are taken from the [reducedDims](#) of a [SingleCellExperiment](#) object. It provides slots and methods to specify which dimensionality reduction result to use and to create the data.frame with the coordinates of the specified results for plotting.

ReducedDimensionPlot slot overview

The following slots control the dimensionality reduction result that is used:

- `Type`, a string specifying the name of the dimensionality reduction result. If NA, defaults to the first entry of `reducedDims`.
- `XAxis`, integer scalar specifying the dimension to plot on the x-axis. Defaults to 1.
- `YAxis`, integer scalar specifying the dimension to plot on the y-axis. Defaults to 2.

In addition, this class inherits all slots from its parent `ColumnDotPlot`, `DotPlot` and `Panel` classes.

Constructor

`ReducedDimensionPlot(...)` creates an instance of a `ReducedDimensionPlot` class, where any slot and its value can be passed to `...` as a named argument.

Supported methods

In the following code snippets, `x` is an instance of a `ReducedDimensionPlot` class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.cacheCommonInfo(x)` adds a "ReducedDimensionPlot" entry containing `valid.reducedDim.names`, a character vector of names of valid dimensionality reduction results (i.e., at least one dimension). This will also call the equivalent `ColumnDotPlot` method.
- `.refineParameters(x, se)` replaces NA values in `RedDimType` with the first valid dimensionality reduction result name in `se`. This will also call the equivalent `ColumnDotPlot` method for further refinements to `x`. If no dimensionality reduction results are available, NULL is returned instead.

For defining the interface:

- `.defineDataInterface(x, se, select_info)` returns a list of interface elements for manipulating all slots described above.
- `.panelColor(x)` will return the specified default color for this panel class.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots described above and in the parent classes. This will also call the equivalent `ColumnDotPlot` method.

For defining the panel name:

- `.fullName(x)` will return "Reduced dimension plot".

For creating the plot:

- `.generateDotPlotData(x, envir)` will create a data.frame of reduced dimension coordinates in `envir`. It will return the commands required to do so as well as a list of labels.

For documentation:

- `.definePanelTour(x)` returns an `data.frame` containing a panel-specific tour.

Subclasses do not have to provide any methods, as this is a concrete class.

Author(s)

Aaron Lun

See Also

[ColumnDotPlot](#), for the immediate parent class.

Examples

```
#####
# For end-users #
#####

x <- ReducedDimensionPlot()
x[["Type"]]
x[["Type"]] <- "TSNE"

#####
# For developers #
#####

library(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Spits out a NULL and a warning if no reducedDims are available.
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

# Replaces the default with something sensible.
sce <- runPCA(sce)
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)
```

registerAppOptions	<i>Set and get app-level options</i>
--------------------	--------------------------------------

Description

Set and get global options for the [iSEE](#) application. These are options that do not correspond to any [Panel](#) slot and cannot be changed by the user after initialization.

Usage

```
registerAppOptions(se, ...)

getOption(name, se, default = NULL)

getAllAppOptions(se)
```

Arguments

<code>se</code>	The SummarizedExperiment object to be supplied to iSEE .
<code>...</code>	Named options to register. Alternatively a single named list containing the options to register.
<code>name</code>	String containing the name of the option to retrieve.
<code>default</code>	Value to return if name is not present in the available options.

Details

`registerAppOptions` provides an alternative mechanism for setting global options, separate from [panelDefaults](#). The primary difference is that `registerAppOptions` allows tuning of options that do not have a corresponding slot in any [Panel](#) subclass. This makes it useful for parameters that the user should not or cannot change within the application, as well as for fine-tuning parameters that are too rarely used to have their own interface elements.

Known options include:

`panel.color` Named character vector of colors. The names of the vector should be set to the name of class to be overridden; if a class is not named here, its default color is used. It is highly recommended to define colors as hex color codes (e.g., "#1e90ff"), for full compatibility with both HTML elements and R plots.

`color.maxlevels` Maximum number of levels for a categorical variable used for coloring. Variables with more levels are coerced to numeric to avoid problems with an overly-large legend. Defaults to 24.

`factor.maxlevels` Maximum number of levels for a categorical variable to be used anywhere in the app. Variables with more levels are coerced to numeric to avoid rendering delays. Defaults to 100.

`RowTable.select.details` A function that takes a string containing the name of a feature (i.e., the current selection in the [RowTable](#)) and returns a HTML element with more details.

`ColumnTable.select.details` A function that takes a string containing the name of a sample (i.e., the current selection in the [ColumnTable](#)) and returns a HTML element with more details.

`tooltip.signif` Number of *significant* digits to display in the tooltip. Defaults to 6.

The registered options are stored in the `SummarizedExperiment` to ensure that we can recover the application state with the combination of the `SummarizedExperiment` and list of `Panel` settings. By comparison, if we had used a global cache as in [panelDefaults](#), we would need to save them separately to ensure that we can recover a particular application state.

By default, `registerAppOptions` will add or replace individual arguments specified by `...`. This means that users can call the function multiple times to accumulate registered options in `se`. The exception is if `...` contains a single list, in which case the entire set of options is directly replaced by that list. For example, one could supply a single empty list to clear `se` of all existing options.

Value

registerAppOptions will return `se`, modified with the application-level options.

getOption will return the value of the specified option, or default if that option is not available.

getAllAppOptions will return a named list of all registered options.

For developers

Developers of Panel subclasses can add arbitrary options to `...` to help control the behavior of their Panel instances. We recommend prefixing any options with the name of the package in the form of `<PACKAGE>_<OPTION>`, so as to avoid conflicts with other options (in the base classes, or in other downstream packages) that have the same name.

For calls to `getOption` that occur after the iSEE app has started, it is not actually necessary to supply `se`. The options in `se` are transferred to a global option store when the app starts, allowing us to call `getOption` without `se` in various Panel methods. This is useful for some generics where `se` is not part of the function signature. Developers can mimic this state (e.g., for unit testing) by calling `.activateAppOptionRegistry` on the SummarizedExperiment produced by `registerAppOptions`. Conversely, calling `.deactivateAppOptionRegistry` will reset the global option store.

Author(s)

Aaron Lun

Examples

```
se <- SummarizedExperiment()
se <- registerAppOptions(se, factor.maxlevels=10, color.maxlevels=10)

getOption("factor.maxlevels", se)
getOption("color.maxlevels", se)
getOption("random.other.thing", se, default=10)

getAllAppOptions(se)

# For developers: you don't actually need to pass 'se' to the getters
# if they are being called inside Panel methods.
.activateAppOptionRegistry(se)
getOption("factor.maxlevels")
getOption("color.maxlevels")
.deactivateAppOptionRegistry()

# Wiping out all options.
se <- registerAppOptions(se, list())
getAllAppOptions(se)
```

RowDataPlot-class *The RowDataPlot panel*

Description

The RowDataPlot is a panel class for creating a [RowDotPlot](#) where the y-axis represents a variable from the [rowData](#) of a [SummarizedExperiment](#) object. It provides slots and methods for specifying which variable to use on the y-axis (and, optionally, also the x-axis), as well as a method to create the data.frame in preparation for plotting.

Slot overview

The following slots control the variables to be shown:

- **YAxis**, a string specifying the row of the [rowData](#) to show on the y-axis. If NA, defaults to the first valid field (see ?"[.refineParameters, RowDotPlot-method](#)").
- **XAxis**, string specifying what should be plotting on the x-axis. This can be any one of "None", "Row data" and "Row selection". Defaults to "None".
- **XAxisRowData**, string specifying the row of the [rowData](#) to show on the x-axis. If NA, defaults to the first valid field.

In addition, this class inherits all slots from its parent [RowDotPlot](#), [DotPlot](#) and [Panel](#) classes.

Constructor

`RowDataPlot(...)` creates an instance of a RowDataPlot class, where any slot and its value can be passed to ... as a named argument.

Supported methods

In the following code snippets, `x` is an instance of a [RowDataPlot](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.refineParameters(x, se)` returns `x` after replacing any NA value in **YAxis** or **XAxisRowData** with the name of the first valid [rowData](#) variable. This will also call the equivalent [RowDotPlot](#) method for further refinements to `x`. If no valid row metadata variables are available, NULL is returned instead.

For defining the interface:

- `.defineDataInterface(x, se, select_info)` returns a list of interface elements for manipulating all slots described above.
- `.panelColor(x)` will return the specified default color for this panel class.
- `.allowableXAxisChoices(x, se)` returns a character vector specifying the acceptable variables in [rowData](#)(`se`) that can be used as choices for the x-axis. This consists of all variables with atomic values.

- `.allowableYAxisChoices(x, se)` returns a character vector specifying the acceptable variables in `rowData(se)` that can be used as choices for the y-axis. This consists of all variables with atomic values.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots described above and in the parent classes. This will also call the equivalent `RowDotPlot` method.

For controlling selections:

- `.multiSelectionInvalidated(x)` returns TRUE if the x-axis uses multiple row selections, such that the point coordinates may change upon updates to upstream selections in transmitting panels. Otherwise, it dispatches to the `RowDotPlot` method.

For defining the panel name:

- `.fullName(x)` will return "Row data plot".

For creating the plot:

- `.generateDotPlotData(x, envir)` will create a data.frame of row metadata variables in `envir`. It will return the commands required to do so as well as a list of labels.

For documentation:

- `.definePanelTour(x)` returns an data.frame containing a panel-specific tour.

Subclass expectations

Subclasses do not have to provide any methods, as this is a concrete class.

Author(s)

Aaron Lun

See Also

`RowDotPlot`, for the immediate parent class.

Examples

```
#####
# For end-users #
#####

x <- RowDataPlot()
x[["XAxis"]]
x[["XAxis"]] <- "Row data"

#####
# For developers #
#####
```

```

library(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)

# Spits out a NULL and a warning if is nothing to plot.
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

# Replaces the default with something sensible.
rowData(sce)$Stuff <- runif(nrow(sce))
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

```

RowDataTable-class	<i>The RowDataTable panel</i>
--------------------	-------------------------------

Description

The RowDataTable is a panel class for creating a [ColumnTable](#) where the value of the table is defined as the [rowData](#) of the [SummarizedExperiment](#). It provides functionality to extract the [rowData](#) to coerce it into an appropriate data.frame in preparation for rendering.

Slot overview

This class inherits all slots from its parent [ColumnTable](#) and [Table](#) classes.

Constructor

`RowDataTable(...)` creates an instance of a RowDataTable class, where any slot and its value can be passed to ... as a named argument.

Note that `ColSearch` should be a character vector of length equal to the total number of columns in the [rowData](#), though only the entries for the atomic fields will actually be used.

Supported methods

In the following code snippets, `x` is an instance of a [RowDataTable](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.cacheCommonInfo(x)` adds a "RowDataTable" entry containing `valid.rowData.names`, a character vector of names of atomic columns of the [rowData](#). This will also call the equivalent [ColumnTable](#) method.
- `.refineParameters(x, se)` adjusts `ColSearch` to a character vector of length equal to the number of atomic fields in the [rowData](#). This will also call the equivalent [ColumnTable](#) method for further refinements to `x`.

For defining the interface:

- `.fullName(x)` will return "Row data table".
- `.panelColor(x)` will return the specified default color for this panel class.

For creating the output:

- `.generateTable(x, envir)` will modify `envir` to contain the relevant data.frame for display, while returning a character vector of commands required to produce that data.frame. Each row of the data.frame should correspond to a row of the SummarizedExperiment.

For documentation:

- `.definePanelTour(x)` returns an data.frame containing the steps of a panel-specific tour.

Unless explicitly specialized above, all methods from the parent class `Panel` are also available.

Author(s)

Aaron Lun

Examples

```
#####
# For end-users #
#####

x <- RowDataTable()
x[["Selected"]]
x[["Selected"]] <- "SOME_ROW_NAME"

#####
# For developers #
#####

library(scater)
sce <- mockSCE()

# Sets the search columns appropriately.
sce <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce)
```

RowDotPlot-class

The RowDotPlot virtual class

Description

The RowDotPlot is a virtual class where each row in the `SummarizedExperiment` is represented by no more than one point (i.e., a “dot”) in a brushable `ggplot` plot. It provides slots and methods to extract `rowData` fields to control the per-point aesthetics on the plot. This panel will transmit row identities in both its single and multiple selections, and it can receive multiple row selections but not multiple column selections.

Slot overview

The following slots control coloring of the points:

- `ColorByRowData`, a string specifying the `rowData` field for controlling point color, if `ColorBy="Row data"` (see the `Panel` class). Defaults to the first valid field (see `.cacheCommonInfo` below).
- `ColorBySampleNameAssay`, a string specifying the assay of the `SummarizedExperiment` object containing values to use for coloring, if `ColorBy="Sample name"`. Defaults to "logcounts" in `getPanelDefault`, falling back to the name of the first valid assay (see `?".cacheCommonInfo, DotPlot-method"` for the definition of validity).
- `ColorByFeatureNameColor`, a string specifying the color to use for coloring an individual sample on the plot, if `ColorBy="Feature name"`. Defaults to "red" in `getPanelDefault`.

The following slots control other metadata-related aesthetic aspects of the points:

- `ShapeByRowData`, a string specifying the `rowData` field for controlling point shape, if `ShapeBy="Row data"` (see the `Panel` class). The specified field should contain categorical values; defaults to the first such field.
- `SizeByRowData`, a string specifying the `rowData` field for controlling point size, if `SizeBy="Row data"` (see the `Panel` class). The specified field should contain continuous values; defaults to the first such field.
- `TooltipRowData`, a character vector specifying `rowData` fields to show in the tooltip. Defaults to `'character(0)'`, which displays only the 'rownames' value of the data point.

In addition, this class inherits all slots from its parent `DotPlot` and `Panel` classes.

Supported methods

In the following code snippets, `x` is an instance of a `RowDotPlot` class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.cacheCommonInfo(x)` adds a "RowDotPlot" entry containing `valid.rowData.names`, a character vector of valid column data names (i.e., containing atomic values); `discrete.rowData.names`, a character vector of names for discrete columns; and `continuous.rowData.names`, a character vector of names of continuous columns. This will also call the equivalent `DotPlot` method.
- `.refineParameters(x, se)` replaces NA values in `ColorByFeatAssay` with the first valid assay name in `se`. This will also call the equivalent `DotPlot` method.

For defining the interface:

- `.hideInterface(x, field)` returns a logical scalar indicating whether the interface element corresponding to `field` should be hidden. This returns TRUE for row selection parameters ("RowSelectionSource" and "RowSelectionRestrict"), otherwise it dispatches to the `Panel` method.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots in the `RowDotPlot`. This will also call the equivalent `DotPlot` method.

For controlling selections:

- `.multiSelectionRestricted(x)` returns a logical scalar indicating whether `x` is restricting the plotted points to those that were selected in a transmitting panel.
- `.multiSelectionDimension(x)` returns "row" to indicate that a multiple row selection is being transmitted.
- `.multiSelectionInvalidated(x)` returns TRUE if the faceting options use the multiple row selections, such that the point coordinates/domain may change upon updates to upstream selections in transmitting panels.
- `.singleSelectionDimension(x)` returns "feature" to indicate that a feature identity is being transmitted.

For documentation:

- `.definePanelTour(x)` returns an `data.frame` containing the steps of a tour relevant to subclasses, mostly tuning the more generic descriptions from the same method of the parent `DotPlot`.
- `.getDotPlotColorHelp(x, color_choices)` returns a `data.frame` containing the documentation for the "ColorBy" UI element, specialized for row-based dot plots.

Unless explicitly specialized above, all methods from the parent classes `DotPlot` and `Panel` are also available.

Subclass expectations

Subclasses are expected to implement methods for, at least:

- `.generateDotPlotData`
- `.fullName`
- `.panelColor`

The method for `.generateDotPlotData` should create a `plot.data` `data.frame` with one row per row in the `SummarizedExperiment` object.

Author(s)

Aaron Lun

See Also

`DotPlot`, for the immediate parent class that contains the actual slot definitions.

RowTable-class

*The RowTable class***Description**

The RowTable is a virtual class where each row in the [SummarizedExperiment](#) is represented by no more than one row in a [datatable](#) widget. In panels of this class, single and multiple selections can only be transmitted on the features.

Slot overview

No new slots are added. All slots provided in the [Table](#) parent class are available.

Supported methods

In the following code snippets, x is an instance of a [RowTable](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- [.refineParameters](#)(x, se) replaces NA values in Selected with the first row name of se. This will also call the equivalent [Table](#) method.

For defining the interface:

- [.hideInterface](#)(x, field) returns a logical scalar indicating whether the interface element corresponding to field should be hidden. This returns TRUE for column selection parameters ("ColumnSelectionSource" and "ColumnSelectionRestrict"), otherwise it dispatches to the [Panel](#) method.

For monitoring reactive expressions:

- [.createObservers](#)(x, se, input, session, pObjects, rObjects) sets up observers to propagate changes in the Selected to linked plots. This will also call the equivalent [Table](#) method.

For controlling selections:

- [.multiSelectionDimension](#)(x) returns "row" to indicate that a row selection is being transmitted.
- [.singleSelectionDimension](#)(x) returns "feature" to indicate that a feature identity is being transmitted.

For rendering output:

- [.showSelectionDetails](#)(x) returns a HTML element containing details about the selected row. This requires a function to be registered by [registerAppOptions](#) under the option name "RowTable.select.details". The function should take a string containing the name of a feature (i.e., the current selection in the [RowTable](#)) and returns a HTML element. If no function is registered, NULL is returned.

Unless explicitly specialized above, all methods from the parent classes [DotPlot](#) and [Panel](#) are also available.

Subclass expectations

Subclasses are expected to implement methods for:

- [.generateTable](#)
- [.fullName](#)
- [.panelColor](#)

The method for [.generateTable](#) should create a tab data.frame where each row corresponds to a row in the [SummarizedExperiment](#) object.

Author(s)

Aaron Lun

See Also

[Table](#), for the immediate parent class that contains the actual slot definitions.

SampleAssayPlot-class *The SampleAssayPlot panel*

Description

The SampleAssayPlot is a panel class for creating a [RowDotPlot](#) where the y-axis represents the expression of a sample of interest, using the [assay](#) values of the [SummarizedExperiment](#). It provides slots and methods for specifying the sample and what to plot on the x-axis, as well as a method to actually create a data.frame containing those pieces of data in preparation for plotting.

Slot overview

The following slots control the values on the y-axis:

- `YAxisSampleName`, a string specifying the name of the sample to plot on the y-axis. If NA, defaults to the first column name of the [SummarizedExperiment](#) object.
- `Assay`, string specifying the name of the assay to use for obtaining expression values. Defaults to "logcounts" in [getPanelDefault](#), falling back to the name of the first valid assay (see [?".cacheCommonInfo, DotPlot-method"](#) for the definition of validity).
- `YAxisSampleSource`, string specifying the encoded name of the transmitting panel to obtain a single selection that replaces `YAxisSampleName`. Defaults to "---", i.e., no transmission is performed.
- `YAxisSampleDynamicSource`, a logical scalar indicating whether x should dynamically change its selection source for the y-axis. Defaults to FALSE in [getPanelDefault](#).

The following slots control the values on the x-axis:

- `XAxis`, string specifying what should be plotted on the x-axis. This can be any one of "None", "Sample name", "Row data" or "Row selection". Defaults to "None".

- `XAxisColumnData`, string specifying which column of the `colData` should be shown on the x-axis, if `XAxis="Column data"`. Defaults to the first valid `colData` field (see `?".refineParameters,ColumnDotPlot-m` for details).
- `XAxisSampleName`, string specifying the name of the sample to plot on the x-axis, if `XAxis="Sample name"`. Defaults to the first column name.
- `XAxisSampleSource`, string specifying the encoded name of the transmitting panel to obtain a single selection that replaces `XAxisSampleName`. Defaults to `"---"`, i.e., no transmission is performed.
- `XAxisSampleDynamicSource`, a logical scalar indicating whether x should dynamically change its selection source for the x-axis. Defaults to `FALSE` in `getPanelDefault`.

In addition, this class inherits all slots from its parent `ColumnDotPlot`, `DotPlot` and `Panel` classes.

Constructor

`SampleAssayPlot(...)` creates an instance of a `SampleAssayPlot` class, where any slot and its value can be passed to `...` as a named argument.

Supported methods

In the following code snippets, `x` is an instance of a `SampleAssayPlot` class. Refer to the documentation for each method for more details on the remaining arguments.

For setting up data values:

- `.refineParameters(x, se)` replaces any NA values in `XAxisSampleName` and `YAxisSampleName` with the first column name; any NA value in `Assay` with the first valid assay name; and any NA value in `XAxisColumnData` with the first valid column metadata field. This will also call the equivalent `ColumnDotPlot` method for further refinements to `x`. If no columns or assays are present, `NULL` is returned instead.

For defining the interface:

- `.defineDataInterface(x, se, select_info)` returns a list of interface elements for manipulating all slots described above.
- `.panelColor(x)` will return the specified default color for this panel class.

For monitoring reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all slots described above and in the parent classes. This will also call the equivalent `ColumnDotPlot` method.

For defining the panel name:

- `.fullName(x)` will return `"Sample assay plot"`.

For creating the plot:

- `.generateDotPlotData(x, envir)` will create a data.frame of sample assay values in `envir`. It will return the commands required to do so as well as a list of labels.

For managing selections:

- `.singleSelectionSlots(x)` will return a list specifying the slots that can be updated by single selections in transmitter panels, mostly related to the choice of sample on the x- and y-axes. This includes the output of the method for the parent [RowDotPlot](#) class.
- `.multiSelectionInvalidated(x)` returns TRUE if the x-axis uses multiple row selections, such that the point coordinates may change upon updates to upstream selections in transmitting panels. Otherwise, it dispatches to the [RowDotPlot](#) method.

For documentation:

- `.definePanelTour(x)` returns an `data.frame` containing a panel-specific tour.

Author(s)

Aaron Lun

See Also

[ColumnDotPlot](#), for the immediate parent class.

Examples

```
#####
# For end-users #
#####

x <- SampleAssayPlot()
x[["XAxis"]]
x[["Assay"]] <- "logcounts"
x[["XAxisRowData"]] <- "stuff"

#####
# For developers #
#####

library(scater)
sce <- mockSCE()
sce <- logNormCounts(sce)

old_assay_names <- assayNames(sce)
assayNames(sce) <- character(length(old_assay_names))

# Spits out a NULL and a warning if no assays are named.
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)

# Replaces the default with something sensible.
assayNames(sce) <- old_assay_names
sce0 <- .cacheCommonInfo(x, sce)
.refineParameters(x, sce0)
```

Description

These generics are related to the initial setup of the **iSEE** application.

Caching common information

`.cacheCommonInfo(x, se)` computes common values that can be re-used for all panels with the same class as `x`. The following arguments are required:

- `x`, an instance of a [Panel](#) class.
- `se`, the [SummarizedExperiment](#) object containing the current dataset.

It is expected to return `se` with (optionally) extra fields added to `int_metadata(se)$iSEE`. Each field should be named according to the class name and contain some common information that is constant for all instances of the class of `x` - see `.setCachedCommonInfo` for an appropriate setter utility. The goal is to avoid repeated recomputation of required values when creating user interface elements or observers that respond to those elements.

Methods for this generic should start by checking whether the metadata already contains the class name, and returning `se` without modification if this is the case. Otherwise, it should [callNextMethod](#) to fill in the cache values from the parent classes, before adding cached values under the class name for `x`. This means that any modification to `se` will only be performed once per class, so any cached values should be constant for all instances of the same class.

Values from the cache can also be [deparsed](#) and used to assemble rendering commands in `.generateOutput`. However, those same commands should not make any use of the cache itself, i.e., they should not call `.getCachedCommonInfo`. This is because the code tracker does not capture the code used to construct the cache, so the commands that are shown to the user will make use of a cache that is not present in the original `se` object.

Refining parameters

`.refineParameters(x, se)` enforces appropriate settings for each parameter in `x`. The following arguments are required:

- `x`, an instance of a [Panel](#) class.
- `se`, the [SummarizedExperiment](#) object containing the current dataset.

Methods for this generic should return a copy of `x` where slots with invalid values are replaced with appropriate entries from `se`. This is necessary because the constructor and validity methods for `x` do not know about `se`; thus, certain slots (e.g., for the row/column names) cannot be set to a reasonable default or checked by the validity method. By comparison, `.refineParameters` can catch and correct invalid values as it has access to `se`.

We recommend specializing `initialize` to fill any yet-to-be-determined slots with NA defaults. `.refineParameters` can then be used to sweep across these slots and replace them with appropriate entries, typically by using `.getCachedCommonInfo` to extract the cached set of potential valid

values. Of course, any slots that are not se-dependent should just be set at construction and checked by the validity method.

It is also possible for this generic to return NULL, which is used as an indicator that se does not contain information to meaningfully show any instance of the class of x in the **iSEE** app. For example, the method for [ReducedDimensionPlot](#) will return NULL if se is not a [SingleCellExperiment](#) containing some dimensionality reduction results.

Author(s)

Aaron Lun

single-select-generics

Generics for controlling single selections

Description

A panel can create a single selection on either the rows or columns and transmit this selection to another panel for use as an aesthetic parameter. For example, users can click on a [RowTable](#) to select a gene of interest, and then the panel can transmit the identities of that row to another panel for coloring by that selected gene's expression. This suite of generics controls the behavior of these single selections.

Specifying the nature of the selection

Given an instance of the [Panel](#) class x, `.singleSelectionDimension(x)` should return a string specifying whether the panel's single selection would contain a "feature", "sample", or if the Panel in x does not perform single selections at all ("none"). The output should be constant for all instances of x.

Obtaining the selected element

`.singleSelectionValue(x, contents)` should return a string specifying the selected row or column in x that is to be transmitted to other panels. If no row or column is selected, it should return NULL.

contents is any arbitrary structure returned by `.generateOutput` for x in the field of the same name. This should contain all of the information necessary to determine the name of the selected row/column. For example, a data.frame of coordinates is stored by [DotPlots](#) to identify the point selected by a brush/lasso.

Indicating the receiving slots

`.singleSelectionSlots(x)` controls how x should *respond* to a single selection. It should return a list of lists, where each internal list describes a set of slots in x that might respond to a single selection from a transmitting panel. This internal list should contain at least entries with the following names:

- `param`, the name of the slot of `x` that can potentially respond to a single selection in a transmitting panel, e.g., `ColorByFeatureName` in [DotPlots](#).
- `source`, the name of the slot of `x` that indicates which transmitting panel to respond to, e.g., `ColorByFeatureSource` in [DotPlots](#).

For each set of responsive slots, the expected paradigm is that the user interface will contain two [selectInput](#) elements, one for each of the `param` and `source` slots. Users are free to manually alter the choice of feature/sample in the `param`'s `selectInput`. Users are also allowed to change the identity of the transmitting panel via the `source`'s `selectInput`, which will automatically update the chosen entry in the `param`'s `selectInput` when the transmitter's single selection changes.

Developers are strongly recommended to follow the above paradigm. In fact, the observers to perform these updates are automatically set up by [.createObservers, Panel-method](#) if the internal list also contains the following named entries:

- `dimension`, a string set to either `"feature"` or `"sample"`. This specifies whether the slot specified by `param` contains the identity of a single feature or a single sample. If this is not present, no observers will be set up.
- `dynamic`, the name of the slot indicating whether the choice of transmitting panel should change dynamically. One example would be `"ColorByFeatureDynamicSource"` for [DotPlots](#). If supplied, a [checkboxInput](#) should also be present in the UI to turn on/off dynamic choices for this parameter. This field can be missing if the current panel does not support dynamic selection sources.
- `use_mode`, the name of the slot of `x` containing the current usage mode. This is used in cases where there are multiple choices of which only one involves using information held in `source`. An example would be `ColorBy` in [DotPlots](#) where coloring by feature name is only one of many options, such that the panel should only respond to transmitted single selections when the user intends to color by feature name. If the value of this field is `NA`, the usage mode for `x` is assumed to be such that the panel should always respond to transmitted single selections.
- `use_value`, a string containing the relevant value of the slot specified by `use_mode` in order for the panel to respond to transmitted single selections. An example would be `"Feature name"` in [DotPlots](#). This field can be missing if `use_mode` is `NA`.
- `protected`, a logical scalar indicating whether the slot specified by `param` is "protected", i.e., changing this value will cause all existing selections to be invalidated and will trigger re-rendering of the children receiving multiple selections. This is `FALSE` for purely aesthetic parameters (e.g., coloring) and `TRUE` for data-related parameters (e.g., `XAxisFeatureName` in [FeatureAssayPlot](#)).

Author(s)

Aaron Lun

Description

Utilities to manage the tours specific to individual UI elements for each [Panel](#). This is done via a global tour cache that is updated by each Panel's interface-generating methods, so that developers can easily put the UI documentation next to the element definition.

Usage

```
.addSpecificTour(cls, field, fun, force = FALSE)

.getSpecificTours(cls)

.clearSpecificTours()
```

Arguments

<code>cls</code>	String containing the name of the Panel class containing the relevant UI element.
<code>field</code>	String containing the slot of the <code>cls</code> class that is controlled by the UI element.
<code>fun</code>	Function that accepts a string containing the encoded Panel name, and returns a <code>data.frame</code> compatible with rintrojs , i.e., with a <code>element</code> and <code>intro</code> column.
<code>force</code>	Logical scalar indicating whether <code>fun</code> should forcibly overwrite an existing function in the tour cache for <code>cls</code> and <code>field</code> .

Details

By default, `.addSpecificTour` will have no effect if a function is already registered for a particular combination of `cls` and `field`. However, users can force a replacement with `force=TRUE`.

`.clearSpecificTours` is intended for use by the **iSEE** app itself and should not be used by Panel methods.

Value

`.addSpecificTour` registers the provided function in the tour cache for the provided `cls` and `field`. A `NULL` is invisibly returned.

`.getSpecificTours` returns a list of registered tour-generating functions for the specified `cls`.

`.clearSpecificTours` removes all functions in the tour cache.

Author(s)

Aaron Lun

See Also

[.selectInput](#), [iSEE](#) and friends, which provide modified UI elements that can be clicked to launch a helpful tour.

subsetPointsByGrid	<i>Subset points for faster plotting</i>
--------------------	--

Description

Subset points using a grid-based system, to avoid unnecessary rendering when plotting.

Usage

```
subsetPointsByGrid(X, Y, resolution = 200, grouping = NULL)
```

Arguments

X	A numeric vector of x-coordinates for all points.
Y	A numeric vector of y-coordinates for all points, of the same length as X.
resolution	A positive integer specifying the number of bins on each axis of the grid. Alternatively, if grouping is specified, this may be a named integer vector containing the number of bins to be used for each level.
grouping	A character vector of length equal to X specifying the group to which each point is assigned. By default, all points belong to the same group.

Details

This function will define a grid of the specified resolution across the plot. Each point is allocated to a grid location (i.e., pair of bins on the x- and y-axes). If multiple points are allocated to a given location, only the last/right-most point is retained. This mimics the fact that plotting will overwrite earlier points with later points. In this manner, we can avoid unnecessary rendering of earlier points that would not show up anyway.

If grouping is specified, redundant points are only identified within each unique level. The resolution of downsampling within each level can be varied by passing an integer vector to resolution. This can be useful for tuning the downsampling when points differ in importance, e.g., in a MA plot, points corresponding to non-DE genes can be aggressively downsampled while points corresponding to DE genes should generally be retained.

For plots where X and Y are originally categorical, use the jittered versions as input to this function.

Value

A logical vector indicating which points should be retained.

Author(s)

Aaron Lun

Examples

```
X <- rnorm(100000)
Y <- X + rnorm(100000)

summary(subsetPointsByGrid(X, Y, resolution=100))

summary(subsetPointsByGrid(X, Y, resolution=200))

summary(subsetPointsByGrid(X, Y, resolution=1000))
```

synchronizeAssays	<i>Synchronize assay colormaps to match those in a SummarizedExperiment</i>
-------------------	---

Description

This function returns an updated [ExperimentColorMap](#) in which colormaps in the assays slot are ordered to match the position of their corresponding assay in the [SingleCellExperiment](#) object. Assays in the [SingleCellExperiment](#) that do not have a match in the [ExperimentColorMap](#) are assigned the appropriate default colormap.

Usage

```
synchronizeAssays(ecm, se)
```

Arguments

ecm	An ExperimentColorMap .
se	A SingleCellExperiment .

Details

It is highly recommended to name *all* assays in both [ExperimentColorMap](#) and [SummarizedExperiment](#) prior to calling this function, as this will facilitate the identification of matching assays between the two objects. In most cases, unnamed colormaps will be dropped from the new [ExperimentColorMap](#) object.

The function supports three main situations:

- If *all* assays in the [SingleCellExperiment](#) are named, this function will populate the assays slot of the new [ExperimentColorMap](#) with the name-matched colormap from the input [ExperimentColorMap](#), if available. Assays in the [SingleCellExperiment](#) that do not have a colormap defined in the [ExperimentColorMap](#) are assigned the appropriate default colormap.
- If *all* assays in the [SingleCellExperiment](#) are unnamed, this function requires that the [ExperimentColorMap](#) supplies a number of assay colormaps *identical* to the number of assays in the [SingleCellExperiment](#) object. In that case, the [ExperimentColorMap](#) object will be returned *as is*.

- If only a subset of assays in the SingleCellExperiment are named, this function will ignore unnamed colormaps in the ExperimentColorMap; It will populate the assays slot of the new ExperimentColorMap with the name-matched colormap from the input ExperimentColorMap, if available. Assays in the SingleCellExperiment that are unnamed, or that do not have a colormap defined in the ExperimentColorMap are assigned the appropriate default colormap.

Value

An [ExperimentColorMap](#) with colormaps in the assay slot synchronized to match the position of the corresponding assay in the SingleCellExperiment.

Author(s)

Kevin Rue-Albrecht

Examples

```
# Example ExperimentColorMap ----

count_colors <- function(n){
  c("black", "brown", "red", "orange", "yellow")
}
fpkm_colors <- viridis::inferno

ecm <- ExperimentColorMap(
  assays = list(
    counts = count_colors,
    tophat_counts = count_colors,
    cufflinks_fpkm = fpkm_colors,
    rsem_counts = count_colors,
    orphan = count_colors,
    orphan2 = count_colors,
    count_colors,
    fpkm_colors
  )
)

# Example SingleCellExperiment ----

library(scRNAseq)
sce <- ReprocessedAllenData(assays="tophat_counts")
library(scater)
sce <- logNormCounts(sce, exprs_values="tophat_counts")
sce <- runPCA(sce)
sce <- runTSNE(sce)

# Example ----

ecm_sync <- synchronizeAssays(ecm, sce)
```

Table-class	<i>The Table class</i>
-------------	------------------------

Description

The Table is a virtual class for all panels containing a [datatable](#) widget from the **DT** package, where each row *usually* corresponds to a row or column of the [SummarizedExperiment](#) object. It provides observers for rendering the table widget, monitoring single selections, and applying global and column-specific searches (which serve as multiple selections).

Slot overview

The following slots control aspects of the `DT::datatable` selection:

- `Selected`, a string containing the name of the currently selected row of the data.frame. Defaults to NA, in which case the value should be chosen by the subclass' `.refineParameters` method.
- `Search`, a string containing the regular expression for the global search. Defaults to "", i.e., no search.
- `SearchColumns`, a unnamed character vector of length equal to the number of columns of the data.frame, where each entry contains the search string for its corresponding column. Alternatively, a character vector of variable length, containing search strings for one or more columns. Defaults to an character vector of length zero, which is internally expanded to an vector of zero-length strings, i.e., no search.

The following slots control the appearance of the table:

- `HiddenColumns`, a character vector containing names of columns to hide. Defaults to an empty vector.

In addition, this class inherits all slots from its parent [Panel](#) class.

Supported methods

In the following code snippets, `x` is an instance of a [Table](#) class. Refer to the documentation for each method for more details on the remaining arguments.

For defining the interface:

- `.defineOutput(x)` returns a UI element for a [dataTableOutput](#) widget.
- `.defineDataInterface(x)` will create interface elements for modifying the table, namely to choose which columns to hide. Note that this is populated by `.generateOutput` upon table rendering, as we do not know the available columns before that point.

For defining reactive expressions:

- `.createObservers(x, se, input, session, pObjects, rObjects)` sets up observers for all of the slots. This will also call the equivalent [Panel](#) method.

- `.renderOutput(x, se, output, pObjects, rObjects)` will add a rendered `datatable` object to output. This will also call the equivalent `Panel` method to render the panel information text boxes.
- `.generateOutput(x, se, all_memory, all_contents)` returns a list containing `contents`, a `data.frame` with one row per point currently present in the table; `commands`, a list of character vector containing the R commands required to generate contents and plot; and `varname`, a string specifying the name of the variable in commands used to generate contents.
- `.exportOutput(x, se, all_memory, all_contents)` will create a CSV file containing the current table, and return a string containing the path to that file. This assumes that the `contents` field returned by `.generateOutput` is a `data.frame` or can be coerced into one.

For controlling selections:

- `.multiSelectionRestricted(x)` returns `TRUE`. Transmission of a selection to a `Table` will manifest as a subsetting of the rows.
- `.multiSelectionActive(x)` returns a list containing the contents of `x[["Search"]]` and `x[["ColumnSearch"]]`. If both contain only empty strings, a `NULL` is returned instead.
- `.multiSelectionCommands(x, index)` returns a character vector of R expressions that - when evaluated - return a character vector of the row names of the table after applying all search filters. The value of `index` is ignored.
- `.singleSelectionValue(x, contents)` returns the name of the row that was last selected in the `datatable` widget.

For documentation:

- `.definePanelTour(x)` returns an `data.frame` containing the steps of a tour relevant to subclasses, mostly describing the effect of selection from other panels and the use of row filters to transmit selections.

Unless explicitly specialized above, all methods from the parent class `Panel` are also available.

Subclass expectations

The `Table` is a rather vaguely defined class for which the only purpose is to avoid duplicating code for `ColumnDotPlots` and `RowDotPlots`. We recommend extending those subclasses instead.

Author(s)

Aaron Lun

See Also

`Panel`, for the immediate parent class.

Description

Generic to control the creation of a data.frame to show in the `datatable` widget of a `Table` panel. `Table` subclasses can specialize methods to modify the behavior of `.generateOutput`.

Constructing the table

`.generateTable(x, envir)` generates the data.frame to use in the `datatable` widget. The following arguments are required:

- `x`, an instance of a `Table` subclass.
- `envir`, the evaluation environment in which the data.frame is to be constructed. This can be assumed to have `se`, the `SummarizedExperiment` object containing the current dataset; possibly `col_selected`, if a multiple column selection is being transmitted to `x`; and possibly `row_selected`, if a multiple row selection is being transmitted to `x`.

In return, the method should add a `tab` variable in `envir` containing the relevant data.frame. This will automatically be passed to the `datatable` widget as well as being stored in `pObjects$contents`. The return value should be a character vector of commands that produces `tab` when evaluated in `envir`.

Each row of the `tab` data.frame should correspond to one row or column in the `SummarizedExperiment` `envir$se` for `RowTables` and `ColumnTables` respectively. Unlike `.generateDotPlotData`, it is not necessary for all rows or columns to be represented in this data.frame.

Ideally, the number and names of the columns of the data.frame should be fixed for all calls to `.generateTable`. Violating this principle may result in unpredictable interactions with existing values in the `SearchColumns` slot. Nonetheless, the app will be robust to mismatches, see `filterDT` for more details.

Any internal variables that are generated by the commands should be prefixed with `.` to avoid potential clashes with reserved variable names in the rest of the application.

This generic is called by `.generateOutput` for `Table` subclasses. Thus, developers of such subclasses only need to specialize `.generateTable` to change the table contents, without needing to reimplement the entirety of `.generateOutput`.

Adding details on the selection

`.showSelectionDetails(x)` should return a HTML element containing details on the currently selected row, given an instance of a `Table` subclass `x`. The identity of the selected row should be extracted from `x[["Selected"]]`. The element will only be rerendered upon a single selection in the `Table`. Alternatively, it may return `NULL` in which case no selection details are shown in the interface.

Author(s)

Aaron Lun

track-utils*Track internal events*

Description

Utility functions to track internal events for a panel by monitoring the status of reactive variables in `rObjects`.

Usage

```
.trackUpdate(panel_name, rObjects)

.trackSingleSelection(panel_name, rObjects)

.trackMultiSelection(panel_name, rObjects)

.trackRelinkedSelection(panel_name, rObjects)
```

Arguments

<code>panel_name</code>	String containing the panel name.
<code>rObjects</code>	A reactive list of values generated in the iSEE app.

Details

`.trackUpdate` will track whether an update has been requested to the current panel via [.requestUpdate](#).

`.trackSingleSelection` will track whether the single selection in the current panel has changed. Note that this will not cause a reaction if the change involves cancelling a single selection.

`.trackMultiSelection` will track whether the multiple selections in the current panel have changed. This will respond for both active and saved selections.

`.trackRelinkedSelection` will track whether the single or multiple selection sources have changed.

These functions should be called within observer or rendering expressions to trigger their evaluation upon panel updates. It is only safe to call these functions within expressions for the same panel, e.g., to synchronize multiple output elements. Calling them with another `panel_name` would be unusual, not least because communication between panels is managed by the [iSEE](#) framework and is outside of the scope of the per-panel observers.

Value

All functions will cause the current reactive context to respond to the designated event. `NULL` is returned invisibly.

Author(s)

Aaron Lun

Description

Helper functions to implement `setValidity` methods for `Panel` subclasses.

Usage

```
.singleStringError(msg, x, fields)

.validLogicalError(msg, x, fields)

.validStringError(msg, x, fields)

.allowableChoiceError(msg, x, field, allowable)

.multipleChoiceError(msg, x, field, allowable)

.validNumberError(msg, x, field, lower, upper)
```

Arguments

<code>msg</code>	Character vector containing the current error messages.
<code>x</code>	An instance of a <code>Panel</code> subclass.
<code>fields</code>	Character vector containing the names of the relevant slots.
<code>field</code>	String containing the name of the relevant slot under investigation.
<code>allowable</code>	Character vector of allowable choices for a multiple-choice selection.
<code>lower</code>	Numeric scalar specifying the lower bound of possible values.
<code>upper</code>	Numeric scalar specifying the upper bound of possible values.

Details

`.singleStringError` adds an error message if any of the slots named in `fields` does not contain a single string.

`.validStringError` adds an error message if any of the slots named in `fields` does not contain a single non-NA string.

`.validLogicalError` adds an error message if any of the slots named in `fields` does not contain a non-NA logical scalar.

`.allowableChoiceError` adds an error message if the slot named `field` does not have a value in `allowable`, assuming it contains a single string.

`.multipleChoiceError` adds an error message if the slot named `field` does not have all of its values in `allowable`, assuming it contains a character vector of any length.

`.validNumberError` adds an error message if the slot named `field` is not a non-NA number within `[lower, upper]`.

Value

All functions return a character vector containing `msg`, possibly appended with additional error messages.

Author(s)

Aaron Lun

visual-parameters-generics

Generics for visual DotPlot parameters

Description

These generics allow subclasses to override the user interface elements controlling visual parameters of [DotPlot](#) panels.

Interface definition

In all of the code snippets below, `x` is a [Panel](#) instance and `se` is the [SummarizedExperiment](#) object.

- `.defineVisualColorInterface(x, se, select_info)` should return a HTML tag definition that contains UI input elements controlling the color aesthetic of ggplot objects. Here, `select_info` is a list of two character vectors named `row` and `column`, which specifies the names of panels available for transmitting single selections on the rows/columns respectively. A common use case would involve adding elements to change the default color of the points or to color by a chosen metadata field/assay values.
- `.defineVisualShapeInterface(x, se)` should return a HTML tag definition that contains UI input elements controlling the shape aesthetic of ggplot objects. A common use case would involve adding elements to change the shape of each point according to a chosen metadata field.
- `.defineVisualSizeInterface(x, se)` should return a HTML tag definition that contains UI input elements controlling the size aesthetic of ggplot objects. A common use case would involve adding elements to change the size of each point according to a chosen metadata field or assay values.
- `.defineVisualPointInterface(x, se)` should return a HTML tag definition that contains UI input elements controlling other aesthetics of ggplot objects. This might include controlling the transparency or downsampling.
- `.defineVisualFacetInterface(x, se)` should return a HTML tag definition that contains UI input elements controlling the `facet_grid` applied to ggplot objects. This typically involves providing UI elements to choose the metadata variables to use for faceting.
- `.defineVisualTextInterface(x, se)` should return a HTML tag definition that contains UI input elements controlling the appearance of non-data text elements of ggplot objects. This typically involves matters such as the font size and legend position.

- `.defineVisualOtherInterface(x)` should a HTML tag definition that contains UI inputs elements to display in the "Other" section of the visual parameters. This is a grab-bag of other parameters that don't fit into the more defined categories above.

A method for any of these generics may also return NULL, in which case the corresponding section of the visual parameter box is completely hidden.

All of these generics are called by `.defineInterface` for DotPlot subclasses. Developers of subclasses can simply specialize these generics to change the UI instead of reimplementing `.defineInterface` itself.

When implementing methods for these generics, it is a good idea to make use of information pre-computed by `.cacheCommonInfo`. For example, `.cacheCommonInfo, ColumnDotPlot-method` will add vectors specifying whether a variable in the `colData` is valid and discrete or continuous.

Controlling ColorBy*Data choices

`.allowableColorByDataChoices(x, se)` should return a character vector of the allowable row/column data variables to use when ColorBy is set to "Row data" or "Column data" for `RowDotPlots` and `ColumnDotPlots`, respectively. The default method will use all available (atomic) variables, but subclasses can specialize this to only allow, e.g., continuous or discrete variables.

Controlling hover choices

`.getTooltipUI(x, se, name)` should return an HTML tag definition representing information to display in the tooltip that is displayed in DotPlot panels when hovering over a data point. The data point is identified by name, its rownames or colnames value in se.

Author(s)

Kevin Rue-Albrecht

Index

- * **datasets**
 - iSEEOptions, [66](#)
- .activateAppOptionRegistry, [91](#)
- .activateAppOptionRegistry (registerAppOptions), [89](#)
- .addCustomLabelsCommands, [4](#)
- .addFacets (plot-utils), [87](#)
- .addLabelCentersCommands, [4](#)
- .addMultiSelectionPlotCommands, [5](#), [85](#)
- .addSpecificTour, [46](#)
- .addSpecificTour (specific-tours), [104](#)
- .addTourStep, [6](#)
- .allowableChoiceError (validate-utils), [113](#)
- .allowableColorByDataChoices, [50](#)
- .allowableColorByDataChoices (visual-parameters-generics), [114](#)
- .allowableColorByDataChoices, DotPlot-method (DotPlot-class), [46](#)
- .allowableXAxisChoices, [27](#), [92](#)
- .allowableXAxisChoices (metadata-plot-generics), [71](#)
- .allowableXAxisChoices, ColumnDataPlot-method (ColumnDataPlot-class), [27](#)
- .allowableXAxisChoices, RowDataPlot-method (RowDataPlot-class), [92](#)
- .allowableYAxisChoices, [28](#), [93](#)
- .allowableYAxisChoices (metadata-plot-generics), [71](#)
- .allowableYAxisChoices, ColumnDataPlot-method (ColumnDataPlot-class), [27](#)
- .allowableYAxisChoices, RowDataPlot-method (RowDataPlot-class), [92](#)
- .buildAes (aes-utils), [20](#)
- .buildLabs, [7](#)
- .cacheCommonInfo, [9](#), [19](#), [21](#), [29](#), [31](#), [37](#), [49](#), [71](#), [80](#), [88](#), [94](#), [96](#), [115](#)
- .cacheCommonInfo (setup-generics), [102](#)
- .cacheCommonInfo, ColumnDataTable-method (ColumnDataTable-class), [29](#)
- .cacheCommonInfo, ColumnDotPlot-method (ColumnDotPlot-class), [31](#)
- .cacheCommonInfo, ComplexHeatmapPlot-method (ComplexHeatmapPlot-class), [35](#)
- .cacheCommonInfo, DotPlot-method (DotPlot-class), [46](#)
- .cacheCommonInfo, Panel-method (Panel-class), [78](#)
- .cacheCommonInfo, ReducedDimensionPlot-method (ReducedDimensionPlot-class), [87](#)
- .cacheCommonInfo, RowDataTable-method (RowDataTable-class), [94](#)
- .cacheCommonInfo, RowDotPlot-method (RowDotPlot-class), [95](#)
- .checkboxGroupInput.iSEE (interface-wrappers), [61](#)
- .checkboxInput.iSEE (interface-wrappers), [61](#)
- .clearSpecificTours (specific-tours), [104](#)
- .colorByNoneDotPlotField, [50](#)
- .colorByNoneDotPlotField (plot-generics), [83](#)
- .colorByNoneDotPlotField, DotPlot-method (DotPlot-class), [46](#)
- .colorByNoneDotPlotScale, [50](#)
- .colorByNoneDotPlotScale (plot-generics), [83](#)
- .colorByNoneDotPlotScale, DotPlot-method (DotPlot-class), [46](#)
- .conditionalOnCheckGroup (.conditionalOnRadio), [8](#)
- .conditionalOnCheckSolo (.conditionalOnRadio), [8](#)
- .conditionalOnRadio, [8](#)
- .createCustomDimnamesModalObservers, [9](#)

- [11, 12, 75](#)
- .createObservers, [9, 17, 28, 32, 34, 38, 46, 50, 56, 60, 77, 80, 88, 93, 96, 98, 100, 109](#)
- .createObservers (observer-generics), [74](#)
- .createObservers, ColumnDataPlot-method (ColumnDataPlot-class), [27](#)
- .createObservers, ColumnDotPlot-method (ColumnDotPlot-class), [31](#)
- .createObservers, ColumnTable-method (ColumnTable-class), [34](#)
- .createObservers, ComplexHeatmapPlot-method (ComplexHeatmapPlot-class), [35](#)
- .createObservers, DotPlot-method (DotPlot-class), [46](#)
- .createObservers, FeatureAssayPlot-method (FeatureAssayPlot-class), [54](#)
- .createObservers, Panel-method (Panel-class), [78](#)
- .createObservers, ReducedDimensionPlot-method (ReducedDimensionPlot-class), [87](#)
- .createObservers, RowDataPlot-method (RowDataPlot-class), [92](#)
- .createObservers, RowDotPlot-method (RowDotPlot-class), [95](#)
- .createObservers, RowTable-method (RowTable-class), [98](#)
- .createObservers, SampleAssayPlot-method (SampleAssayPlot-class), [99](#)
- .createObservers, Table-method (Table-class), [109](#)
- .createProtectedParameterObservers, [18, 75](#)
- .createProtectedParameterObservers (.createUnprotectedParameterObservers), [10](#)
- .createUnprotectedParameterObservers, [10, 75](#)
- .dataParamBoxOpen (constants), [39](#)
- .deactivateAppOptionRegistry, [91](#)
- .deactivateAppOptionRegistry (registerAppOptions), [89](#)
- .defineDataInterface, [27, 38, 49, 56, 71, 80, 88, 92, 100, 109](#)
- .defineDataInterface (interface-generics), [60](#)
- .defineDataInterface, ColumnDataPlot-method (ColumnDataPlot-class), [27](#)
- .defineDataInterface, ComplexHeatmapPlot-method (ComplexHeatmapPlot-class), [35](#)
- .defineDataInterface, FeatureAssayPlot-method (FeatureAssayPlot-class), [54](#)
- .defineDataInterface, Panel-method (Panel-class), [78](#)
- .defineDataInterface, ReducedDimensionPlot-method (ReducedDimensionPlot-class), [87](#)
- .defineDataInterface, RowDataPlot-method (RowDataPlot-class), [92](#)
- .defineDataInterface, SampleAssayPlot-method (SampleAssayPlot-class), [99](#)
- .defineDataInterface, Table-method (Table-class), [109](#)
- .defineInterface, [19, 21, 26, 38, 46, 49, 75, 80, 115](#)
- .defineInterface (interface-generics), [60](#)
- .defineInterface, ColumnDotPlot-method (ColumnDotPlot-class), [31](#)
- .defineInterface, ColumnTable-method (ColumnTable-class), [34](#)
- .defineInterface, ComplexHeatmapPlot-method (ComplexHeatmapPlot-class), [35](#)
- .defineInterface, DotPlot-method (DotPlot-class), [46](#)
- .defineInterface, Panel-method (Panel-class), [78](#)
- .defineInterface, RowDotPlot-method (RowDotPlot-class), [95](#)
- .defineInterface, RowTable-method (RowTable-class), [98](#)
- .defineOutput, [38, 50, 81, 109](#)
- .defineOutput (output-generics), [76](#)
- .defineOutput, ComplexHeatmapPlot-method (ComplexHeatmapPlot-class), [35](#)
- .defineOutput, DotPlot-method (DotPlot-class), [46](#)
- .defineOutput, Table-method (Table-class), [109](#)
- .definePanelTour, [28, 30, 32, 38, 51, 56, 80, 89, 93, 95, 97, 101, 110](#)
- .definePanelTour (documentation-generics), [45](#)
- .definePanelTour, ColumnDataPlot-method (ColumnDataPlot-class), [27](#)

- .definePanelTour, ColumnDataTable-method
(ColumnDataTable-class), [29](#)
- .definePanelTour, ColumnDotPlot-method
(ColumnDotPlot-class), [31](#)
- .definePanelTour, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), [35](#)
- .definePanelTour, DotPlot-method
(DotPlot-class), [46](#)
- .definePanelTour, FeatureAssayPlot-method
(FeatureAssayPlot-class), [54](#)
- .definePanelTour, Panel-method
(Panel-class), [78](#)
- .definePanelTour, ReducedDimensionPlot-method
(ReducedDimensionPlot-class),
[87](#)
- .definePanelTour, RowDataPlot-method
(RowDataPlot-class), [92](#)
- .definePanelTour, RowDataTable-method
(RowDataTable-class), [94](#)
- .definePanelTour, RowDotPlot-method
(RowDotPlot-class), [95](#)
- .definePanelTour, SampleAssayPlot-method
(SampleAssayPlot-class), [99](#)
- .definePanelTour, Table-method
(Table-class), [109](#)
- .defineVisualColorInterface, [49](#)
- .defineVisualColorInterface
(visual-parameters-generics),
[114](#)
- .defineVisualColorInterface, DotPlot-method
(DotPlot-class), [46](#)
- .defineVisualFacetInterface, [50](#)
- .defineVisualFacetInterface
(visual-parameters-generics),
[114](#)
- .defineVisualFacetInterface, DotPlot-method
(DotPlot-class), [46](#)
- .defineVisualOtherInterface, [50](#)
- .defineVisualOtherInterface
(visual-parameters-generics),
[114](#)
- .defineVisualOtherInterface, DotPlot-method
(DotPlot-class), [46](#)
- .defineVisualPointInterface, [50](#)
- .defineVisualPointInterface
(visual-parameters-generics),
[114](#)
- .defineVisualPointInterface, DotPlot-method
(DotPlot-class), [46](#)
- .defineVisualShapeInterface, [49](#)
- .defineVisualShapeInterface
(visual-parameters-generics),
[114](#)
- .defineVisualShapeInterface, DotPlot-method
(DotPlot-class), [46](#)
- .defineVisualSizeInterface, [49](#)
- .defineVisualSizeInterface
(visual-parameters-generics),
[114](#)
- .defineVisualSizeInterface, DotPlot-method
(DotPlot-class), [46](#)
- .defineVisualTextInterface, [50](#)
- .defineVisualTextInterface
(visual-parameters-generics),
[114](#)
- .defineVisualTextInterface, DotPlot-method
(DotPlot-class), [46](#)
- .emptyDefault (class-utils), [23](#)
- .exportOutput, [38](#), [50](#), [80](#), [110](#)
- .exportOutput (output-generics), [76](#)
- .exportOutput, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), [35](#)
- .exportOutput, DotPlot-method
(DotPlot-class), [46](#)
- .exportOutput, Panel-method
(Panel-class), [78](#)
- .exportOutput, Table-method
(Table-class), [109](#)
- .extractAssaySubmatrix, [11](#)
- .findAtomicFields (cache-utils), [21](#)
- .fullName, [13](#), [28](#), [30](#), [32](#), [35](#), [38](#), [56](#), [81](#), [88](#),
[93](#), [95](#), [97](#), [99](#), [100](#)
- .fullName, ColumnDataPlot-method
(ColumnDataPlot-class), [27](#)
- .fullName, ColumnDataTable-method
(ColumnDataTable-class), [29](#)
- .fullName, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), [35](#)
- .fullName, FeatureAssayPlot-method
(FeatureAssayPlot-class), [54](#)
- .fullName, ReducedDimensionPlot-method
(ReducedDimensionPlot-class),
[87](#)
- .fullName, RowDataPlot-method
(RowDataPlot-class), [92](#)
- .fullName, RowDataTable-method

- (RowDataTable-class), 94
- .fullName, SampleAssayPlot-method
(SampleAssayPlot-class), 99
- .generateDotPlot, 4, 5, 50
- .generateDotPlot (plot-generics), 83
- .generateDotPlot, DotPlot-method
(DotPlot-class), 46
- .generateDotPlotData, 28, 32, 56, 84–86,
88, 93, 97, 100, 111
- .generateDotPlotData (plot-generics), 83
- .generateDotPlotData, ColumnDataPlot-method
(ColumnDataPlot-class), 27
- .generateDotPlotData, FeatureAssayPlot-method
(FeatureAssayPlot-class), 54
- .generateDotPlotData, ReducedDimensionPlot-method
(ReducedDimensionPlot-class),
87
- .generateDotPlotData, RowDataPlot-method
(RowDataPlot-class), 92
- .generateDotPlotData, SampleAssayPlot-method
(SampleAssayPlot-class), 99
- .generateOutput, 15, 18, 19, 38, 50, 73, 77,
78, 81, 83, 84, 86, 102, 103, 109–111
- .generateOutput (output-generics), 76
- .generateOutput, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), 35
- .generateOutput, DotPlot-method
(DotPlot-class), 46
- .generateOutput, Table-method
(Table-class), 109
- .generateTable, 30, 35, 95, 99
- .generateTable (table-generics), 111
- .generateTable, ColumnDataTable-method
(ColumnDataTable-class), 29
- .generateTable, RowDataTable-method
(RowDataTable-class), 94
- .getCachedCommonInfo, 102
- .getCachedCommonInfo
(.setCachedCommonInfo), 19
- .getDotPlotColorHelp, 32, 97
- .getDotPlotColorHelp
(documentation-generics), 45
- .getDotPlotColorHelp, ColumnDotPlot-method
(ColumnDotPlot-class), 31
- .getDotPlotColorHelp, RowDotPlot-method
(RowDotPlot-class), 95
- .getEncodedName, 59, 60, 62, 75–78
- .getEncodedName (.fullName), 13
- .getFullName (.fullName), 13
- .getPanelColor (.panelColor), 13
- .getSpecificTours (specific-tours), 104
- .getTooltipUI
(visual-parameters-generics),
114
- .getTooltipUI, ColumnDotPlot-method
(ColumnDotPlot-class), 31
- .getTooltipUI, RowDotPlot-method
(RowDotPlot-class), 95
- .hideInterface, 6, 32, 34, 38, 59–61, 80, 96,
98
- .hideInterface (interface-generics), 60
- .hideInterface, ColumnDotPlot-method
(ColumnDotPlot-class), 31
- .hideInterface, ColumnTable-method
(ColumnTable-class), 34
- .hideInterface, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), 35
- .hideInterface, DotPlot-method
(DotPlot-class), 46
- .hideInterface, Panel-method
(Panel-class), 78
- .hideInterface, RowDotPlot-method
(RowDotPlot-class), 95
- .hideInterface, RowTable-method
(RowTable-class), 98
- .hideInterface, Table-method
(Table-class), 109
- .isAssayNumeric (cache-utils), 21
- .isBrushable (multi-select-generics), 71
- .isBrushable, DotPlot-method
(DotPlot-class), 46
- .isBrushable, Panel-method
(Panel-class), 78
- .multiSelectHistory (constants), 39
- .multiSelectionActive, 50, 72, 80, 110
- .multiSelectionActive
(multi-select-generics), 71
- .multiSelectionActive, DotPlot-method
(DotPlot-class), 46
- .multiSelectionActive, Panel-method
(Panel-class), 78
- .multiSelectionActive, Table-method
(Table-class), 109
- .multiSelectionAvailable, 80
- .multiSelectionAvailable
(multi-select-generics), 71

- .multiSelectionAvailable, Panel-method
(Panel-class), 78
- .multiSelectionClear, 51, 80
- .multiSelectionClear
(multi-select-generics), 71
- .multiSelectionClear, DotPlot-method
(DotPlot-class), 46
- .multiSelectionClear, Panel-method
(Panel-class), 78
- .multiSelectionCommands, 50, 77, 110
- .multiSelectionCommands
(multi-select-generics), 71
- .multiSelectionCommands, DotPlot-method
(DotPlot-class), 46
- .multiSelectionCommands, Table-method
(Table-class), 109
- .multiSelectionDimension, 32, 34, 60, 77,
80, 97, 98
- .multiSelectionDimension
(multi-select-generics), 71
- .multiSelectionDimension, ColumnDotPlot-method
(ColumnDotPlot-class), 31
- .multiSelectionDimension, ColumnTable-method
(ColumnTable-class), 34
- .multiSelectionDimension, DotPlot-method
(DotPlot-class), 46
- .multiSelectionDimension, Panel-method
(Panel-class), 78
- .multiSelectionDimension, RowDotPlot-method
(RowDotPlot-class), 95
- .multiSelectionDimension, RowTable-method
(RowTable-class), 98
- .multiSelectionInvalidated, 28, 32, 56,
80, 93, 97, 101
- .multiSelectionInvalidated
(multi-select-generics), 71
- .multiSelectionInvalidated, ColumnDataPlot-method
(ColumnDataPlot-class), 27
- .multiSelectionInvalidated, ColumnDotPlot-method
(ColumnDotPlot-class), 31
- .multiSelectionInvalidated, FeatureAssayPlot-method
(FeatureAssayPlot-class), 54
- .multiSelectionInvalidated, Panel-method
(Panel-class), 78
- .multiSelectionInvalidated, RowDataPlot-method
(RowDataPlot-class), 92
- .multiSelectionInvalidated, RowDotPlot-method
(RowDotPlot-class), 95
- .multiSelectionInvalidated, SampleAssayPlot-method
(SampleAssayPlot-class), 99
- .multiSelectionResponsive, 80
- .multiSelectionResponsive
(multi-select-generics), 71
- .multiSelectionResponsive, ColumnDotPlot-method
(ColumnDotPlot-class), 31
- .multiSelectionResponsive, ColumnTable-method
(ColumnTable-class), 34
- .multiSelectionResponsive, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), 35
- .multiSelectionResponsive, Panel-method
(Panel-class), 78
- .multiSelectionResponsive, RowDotPlot-method
(RowDotPlot-class), 95
- .multiSelectionResponsive, RowTable-method
(RowTable-class), 98
- .multiSelectionRestricted, 32, 80, 97,
110
- .multiSelectionRestricted
(multi-select-generics), 71
- .multiSelectionRestricted, ColumnDotPlot-method
(ColumnDotPlot-class), 31
- .multiSelectionRestricted, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), 35
- .multiSelectionRestricted, Panel-method
(Panel-class), 78
- .multiSelectionRestricted, RowDotPlot-method
(RowDotPlot-class), 95
- .multiSelectionRestricted, Table-method
(Table-class), 109
- .multipleChoiceError (validate-utils),
113
- .noSelection (constants), 39
- .numericInput.iSEE
(interface-wrappers), 61
- .organizationHeight (constants), 39
- .organizationWidth (constants), 39
- .panelColor, 13, 27, 30, 32, 35, 38, 56, 81,
88, 92, 95, 97, 99, 100
- .panelColor, ColumnDataPlot-method
(ColumnDataPlot-class), 27
- .panelColor, ColumnDataTable-method
(ColumnDataTable-class), 29
- .panelColor, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), 35
- .panelColor, FeatureAssayPlot-method
(FeatureAssayPlot-class), 54

- .panelColor, ReducedDimensionPlot-method
(ReducedDimensionPlot-class), [87](#)
- .panelColor, RowDataPlot-method
(RowDataPlot-class), [92](#)
- .panelColor, RowDataTable-method
(RowDataTable-class), [94](#)
- .panelColor, SampleAssayPlot-method
(SampleAssayPlot-class), [99](#)
- .prioritizeDotPlotData, [50](#)
- .prioritizeDotPlotData (plot-generics), [83](#)
- .prioritizeDotPlotData, DotPlot-method
(DotPlot-class), [46](#)
- .processMultiSelections, [12](#), [15](#), [78](#), [85](#)
- .radioButtons.iSEE
(interface-wrappers), [61](#)
- .refineParameters, [16](#), [27](#), [30](#), [32](#), [34](#), [38](#),
[49](#), [56](#), [60](#), [75](#), [80](#), [88](#), [92](#), [94](#), [96](#), [98](#),
[100](#), [102](#), [109](#)
- .refineParameters (setup-generics), [102](#)
- .refineParameters, ColumnDataPlot-method
(ColumnDataPlot-class), [27](#)
- .refineParameters, ColumnDataTable-method
(ColumnDataTable-class), [29](#)
- .refineParameters, ColumnDotPlot-method
(ColumnDotPlot-class), [31](#)
- .refineParameters, ColumnTable-method
(ColumnTable-class), [34](#)
- .refineParameters, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), [35](#)
- .refineParameters, DotPlot-method
(DotPlot-class), [46](#)
- .refineParameters, FeatureAssayPlot-method
(FeatureAssayPlot-class), [54](#)
- .refineParameters, Panel-method
(Panel-class), [78](#)
- .refineParameters, ReducedDimensionPlot-method
(ReducedDimensionPlot-class),
[87](#)
- .refineParameters, RowDataPlot-method
(RowDataPlot-class), [92](#)
- .refineParameters, RowDataTable-method
(RowDataTable-class), [94](#)
- .refineParameters, RowDotPlot-method
(RowDotPlot-class), [95](#)
- .refineParameters, RowTable-method
(RowTable-class), [98](#)
- .refineParameters, SampleAssayPlot-method
(SampleAssayPlot-class), [99](#)
- .refineParameters, Table-method
(Table-class), [109](#)
- .removeInvalidChoices, [16](#)
- .renderOutput, [15](#), [18](#), [19](#), [38](#), [50](#), [72](#), [75](#), [78](#),
[80](#), [81](#), [84](#), [110](#)
- .renderOutput (output-generics), [76](#)
- .renderOutput, ComplexHeatmapPlot-method
(ComplexHeatmapPlot-class), [35](#)
- .renderOutput, DotPlot-method
(DotPlot-class), [46](#)
- .renderOutput, Panel-method
(Panel-class), [78](#)
- .renderOutput, Table-method
(Table-class), [109](#)
- .replaceMissingWithFirst, [16](#)
- .requestActiveSelectionUpdate
(.requestUpdate), [17](#)
- .requestCleanUpdate, [11](#), [76](#)
- .requestCleanUpdate (.requestUpdate), [17](#)
- .requestUpdate, [11](#), [17](#), [18](#), [75](#), [77](#), [112](#)
- .retrieveOutput, [18](#), [77](#)
- .selectInput.iSEE, [46](#), [105](#)
- .selectInput.iSEE (interface-wrappers),
[61](#)
- .selectInputHidden (hidden-inputs), [59](#)
- .selectizeInput.iSEE
(interface-wrappers), [61](#)
- .setCachedCommonInfo, [19](#), [102](#)
- .showSelectionDetails, [35](#), [98](#)
- .showSelectionDetails (table-generics),
[111](#)
- .showSelectionDetails, ColumnTable-method
(ColumnTable-class), [34](#)
- .showSelectionDetails, RowTable-method
(RowTable-class), [98](#)
- .singleSelectionDimension, [32](#), [34](#), [60](#), [80](#),
[97](#), [98](#)
- .singleSelectionDimension
(single-select-generics), [103](#)
- .singleSelectionDimension, ColumnDotPlot-method
(ColumnDotPlot-class), [31](#)
- .singleSelectionDimension, ColumnTable-method
(ColumnTable-class), [34](#)
- .singleSelectionDimension, Panel-method
(Panel-class), [78](#)
- .singleSelectionDimension, RowDotPlot-method

- assayNames<-, ExperimentColorMap, ANY-method
(ExperimentColorMap-class), 51
- assays, ExperimentColorMap-method
(ExperimentColorMap-class), 51
- assays<-, ExperimentColorMap, list-method
(ExperimentColorMap-class), 51
- brushedPoints, 48, 69
- cache-utils, 21
- callNextMethod, 46, 60, 61, 75, 85, 102
- checkboxInput, 41, 104
- checkColormapCompatibility, 22
- class-utils, 23
- class:ExperimentColorMap
(ExperimentColorMap-class), 51
- cleanDataset, 24
- cleanDataset, SingleCellExperiment-method
(cleanDataset), 24
- cleanDataset, SummarizedExperiment-method
(cleanDataset), 24
- colData, 21, 25, 27–31, 36, 48, 55, 71, 100, 115
- colData, ExperimentColorMap-method
(ExperimentColorMap-class), 51
- colData<-, ExperimentColorMap, ANY-method
(ExperimentColorMap-class), 51
- colDataColorMap, 33
- colDataColorMap
(ExperimentColorMap-class), 51
- colDataColorMap, ExperimentColorMap, character-method
(ExperimentColorMap-class), 51
- colDataColorMap, ExperimentColorMap, missing-method
(ExperimentColorMap-class), 51
- colDataColorMap<-
(ExperimentColorMap-class), 51
- colDataColorMap<-, ExperimentColorMap, character-method
(ExperimentColorMap-class), 51
- collapseBox, 25, 60
- columnAnnotation, 36
- ColumnDataPlot, 27, 71
- ColumnDataPlot (ColumnDataPlot-class), 27
- ColumnDataPlot-class, 27
- ColumnDataTable, 29
- ColumnDataTable
(ColumnDataTable-class), 29
- ColumnDataTable-class, 29
- ColumnDotPlot, 27, 28, 31, 47, 48, 51, 54–56, 73, 80, 82, 84, 85, 87–89, 100, 101, 110, 115
- ColumnDotPlot-class, 31
- columnSelectionColorMap, 33
- ColumnTable, 29, 30, 34, 35, 67, 90, 94, 111
- ColumnTable-class, 34
- ComplexHeatmapPlot, 12, 37, 73, 82
- ComplexHeatmapPlot
(ComplexHeatmapPlot-class), 35
- ComplexHeatmapPlot-class, 35
- conditionalPanel, 8
- constants, 39
- createCustomPanels, 40
- createCustomPlot (createCustomPanels), 40
- createCustomTable (createCustomPanels), 40
- createLandingPage, 42, 64
- DataFrame, 21
- datatable, 21, 34, 57, 58, 98, 109–111
- dataTableOutput, 76, 109
- defaultTour, 44, 45, 64
- deparse, 102
- documentation-generics, 45
- DotPlot, 4, 5, 13, 17, 27, 31–33, 46, 49, 50, 55, 71–73, 76, 81–86, 88, 92, 96–98, 100, 103, 104, 114
- DotPlot-class, 46
- eval, 78
- ExperimentColorMap, 22, 33, 44, 63, 64, 107, 108
- ExperimentColorMap
(ExperimentColorMap-class), 51
- ExperimentColorMap-class, 51
- FeatureAssayPlot, 55, 82, 104
- FeatureAssayPlot
(FeatureAssayPlot-class), 54
- FeatureAssayPlot-class, 54
- filterDT, 58, 111
- filterDT (filterDTColumn), 57
- filterDTColumn, 57
- getAllAppOptions (registerAppOptions), 89
- getAppOption, 91

- getAppOption (registerAppOptions), 89
- getPanelDefault, 31, 36, 37, 47, 49, 55, 79, 96, 99, 100
- getPanelDefault (panelDefaults), 81
- ggplot, 4, 5, 7, 21, 31, 40, 46, 50, 77, 84–86, 95
- Heatmap, 35, 38
- hidden, 62
- hidden-inputs, 59
- in.out, 69
- initialize, 23, 102
- initialize, ColumnDataPlot-method (ColumnDataPlot-class), 27
- initialize, ColumnDataTable-method (ColumnDataTable-class), 29
- initialize, ColumnDotPlot-method (ColumnDotPlot-class), 31
- initialize, ColumnTable-method (ColumnTable-class), 34
- initialize, ComplexHeatmapPlot-method (ComplexHeatmapPlot-class), 35
- initialize, DotPlot-method (DotPlot-class), 46
- initialize, FeatureAssayPlot-method (FeatureAssayPlot-class), 54
- initialize, Panel-method (Panel-class), 78
- initialize, ReducedDimensionPlot-method (ReducedDimensionPlot-class), 87
- initialize, RowDataPlot-method (RowDataPlot-class), 92
- initialize, RowDataTable-method (RowDataTable-class), 94
- initialize, RowDotPlot-method (RowDotPlot-class), 95
- initialize, RowTable-method (RowTable-class), 98
- initialize, SampleAssayPlot-method (SampleAssayPlot-class), 99
- initialize, Table-method (Table-class), 109
- int_metadata, 19, 102
- interface-generics, 60
- interface-wrappers, 61
- iSEE, 9, 11, 14, 17, 18, 21, 43–46, 62, 63, 64, 75, 76, 89–91, 112
- iSEE-package (iSEE-pkg), 65
- iSEE-pkg, 65
- iSEEOptions, 66
- jitterSquarePoints, 67
- jitterViolinPoints (jitterSquarePoints), 67
- lassoPoints, 48, 69
- make.unique, 25
- manage_commands, 70
- metadata-plot-generics, 71
- multi-select-generics, 71
- multiSelectionToFactor, 33, 74
- numericInput, 41
- observeEvent, 11
- observer-generics, 74
- offsetX, 68
- output-generics, 76
- package_version, 79
- Panel, 6, 12–17, 25–27, 30–32, 34, 35, 37–41, 43, 44, 46, 49–51, 55, 59–64, 66, 71, 73, 75–79, 81–83, 88–90, 92, 95–98, 100, 102, 103, 105, 109, 110, 113, 114
- Panel-class, 78
- panelDefaults, 66, 81, 90
- parse, 78
- plot-generics, 83
- plot-utils, 87
- plotOutput, 76
- readRDS, 43
- ReducedDimensionPlot, 63, 73, 88, 103
- ReducedDimensionPlot (ReducedDimensionPlot-class), 87
- ReducedDimensionPlot-class, 87
- reducedDims, 25, 87, 88
- registerAppOptions, 35, 66, 83, 89, 91, 98
- renderPlot, 76
- renderUI, 44
- rowAnnotation, 36
- rowData, 21, 25, 36, 48, 71, 92–96
- rowData, ExperimentColorMap-method (ExperimentColorMap-class), 51

rowData<- ,ExperimentColorMap,ANY-method
 (ExperimentColorMap-class), 51
 rowDataColorMap, 33
 rowDataColorMap
 (ExperimentColorMap-class), 51
 rowDataColorMap,ExperimentColorMap,character-method
 (ExperimentColorMap-class), 51
 rowDataColorMap,ExperimentColorMap,missing-method
 (ExperimentColorMap-class), 51
 rowDataColorMap<-
 (ExperimentColorMap-class), 51
 rowDataColorMap<- ,ExperimentColorMap,character-method
 (ExperimentColorMap-class), 51
 RowDataPlot, 71, 92
 RowDataPlot (RowDataPlot-class), 92
 RowDataPlot-class, 92
 RowDataTable, 94
 RowDataTable (RowDataTable-class), 94
 RowDataTable-class, 94
 RowDotPlot, 47, 48, 51, 73, 82, 84, 85, 92, 93,
 96, 99, 101, 110, 115
 RowDotPlot-class, 95
 rowSelectionColorMap
 (columnSelectionColorMap), 33
 RowTable, 67, 90, 98, 103, 111
 RowTable-class, 98
 runApp, 63

 SampleAssayPlot, 82, 100
 SampleAssayPlot
 (SampleAssayPlot-class), 99
 SampleAssayPlot-class, 99
 scale_color_manual, 86
 selectInput, 25, 41, 59, 62, 104
 setup-generics, 102
 setValidity, 113
 shinyApp, 63
 show,ExperimentColorMap-method
 (ExperimentColorMap-class), 51
 show,Panel-method (Panel-class), 78
 single-select-generics, 103
 SingleCellExperiment, 22, 25, 87, 103, 107
 specific-tours, 104
 subsetPointsByGrid, 49, 106
 SummarizedExperiment, 9, 12, 15, 18, 19, 21,
 24, 27, 29, 31, 32, 34–36, 41, 43, 46,
 54, 60, 62–64, 71, 72, 75–78, 84, 90,
 92, 94, 95, 97–99, 102, 109, 111, 114
 synchronizeAssays, 107

 Table, 29, 34, 35, 81, 94, 98, 99, 109, 111
 Table-class, 109
 table-generics, 111
 tagList, 76
 textInput, 41
 tree-utils, 112
 updateObject, 79, 80
 updateObject,ColumnDotPlot-method
 (ColumnDotPlot-class), 31
 updateObject,ComplexHeatmapPlot-method
 (ComplexHeatmapPlot-class), 35
 updateObject,DotPlot-method
 (DotPlot-class), 46
 updateObject,Panel-method
 (Panel-class), 78
 updateObject,RowDotPlot-method
 (RowDotPlot-class), 95
 updateObject,Table-method
 (Table-class), 109

 validate-utils, 113
 visual-parameters-generics, 114