Package ‘iSEE’

October 17, 2020

Title Interactive SummarizedExperiment Explorer

Version 2.0.0

Date 2020-04-25

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, colourpicker, igraph, vipor, mgcv, graphics, grDevices, viridisLite, shinyWidgets, ComplexHeatmap, circlize

Suggests testthat, BiocStyle, knitr, rmarkdown, scRNAseq, scater, DelayedArray, RColorBrewer, viridis, htmltools

URL `https://github.com/iSEE/iSEE`

BugReports `https://github.com/iSEE/iSEE/issues`

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

License MIT + file LICENSE

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.1.0

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

git_branch RELEASE_3_11

git_last_commit 0d82c2a

git_last_commit_date 2020-04-27

Date/Publication 2020-10-16

Author Kevin Rue-Albrecht [aut] (<https://orcid.org/0000-0003-3899-3872>),
Federico Marini [aut] (<https://orcid.org/0000-0003-3252-7758>),
Charlotte Soneson [aut, cre] (<https://orcid.org/0000-0003-3833-2169>),
Aaron Lun [aut] (<https://orcid.org/0000-0002-3564-4813>)

Maintainer Charlotte Soneson <charlottesoneson@gmail.com>

R topics documented:

`.addMultiSelectionPlotCommands`
Add multiple selection plotting commands

Description

Add ggplot instructions to create brushes and lassos for both saved and active mutliple selections in a DotPlot panel.

Usage

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

Arguments

`x`	An instance of a DotPlot class.
`envir`	The environment in which the ggplot commands are to be evaluated.
`commands`	A character vector representing the sequence of commands to create the ggplot object.
`flip`	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

`.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"))
```

`.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

`panel_name`	String containing the name of the panel.
`fields`	Character vector of names of parameters for which to set up observers.
`input`	The Shiny input 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.
`ignoreInit, ignoreNULL`	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.

`.fullName` *Get panel names*

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

.panelColor *Get panel colors*

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. and determines the default color for all instances of that class. It is highly recommended to define 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
iSEEOptions$set(panel.color=c(ReducedDimensionPlot="#1e90ff"))
.getPanelColor(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

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

`.retrieveOutput` *Reactive manipulations for Panel output*

Description

Respond to or request a re-rendering of the Panel output via reactive variables.

Usage

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

.requestUpdate(panel_name, rObjects)

.requestCleanUpdate(panel_name, pObjects, rObjects)

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

Arguments

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

Details

`.retrieveOutput` should be used in the expression for rendering output, e.g., in `.renderOutput`. This takes care of a number of house-keeping tasks required to satisfy `.renderOutput`'s requirements with respect to updating various fields in `pObjects`. It also improves efficiency by retrieving cached outputs that were used elsewhere in the app.

`.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.

Value

`.retrieveOutput` will return the output of running `.generateOutput` for the current panel.

`.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 call request a re-rendering of the panel.

All `.request*` functions will invisibly return `NULL`.

Author(s)

Aaron Lun

See Also

`.createProtectedParameterObservers`, for examples where the update-requesting functions are used.

`.setCachedCommonInfo` *Set and get cached commons*

Description

Get and set common cached information for each class. The setter is usually called in `.cacheCommonInfo` while the getter is usually called in `.defineInterface`.

Usage

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

.getCachedCommonInfo(se, cls)
```

Arguments

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

Details

This function is intended for use by developers of Panel classes. If you're an end-user and you're reading this, you probably took a wrong turn somewhere.

Value

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

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

Author(s)

Aaron Lun

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

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

Value

Aesthetic instructions for `ggplot` as a character value.

Author(s)

Kevin Rue-Albrecht

Examples

```
.buildAes()
```

---

checkColormapCompatibility

Check compatibility between ExperimentColorMap and SummarizedExperiment 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 incompability 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 `initialize` method for a given class.
`field`	String specifying the field to set.
`default`	The default value of the slot in `field`.

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)
```

`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

`id`	String specifying the identifier for this object, to use as a field of the Shiny input.
`title`	String specifying the title of the box for use in the UI.
`...`	Additional UI elements to show inside the box.
`open`	Logical scalar indicating whether this box should be open upon initialization.
`style`	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 column metadata variable to use on the y-axis and what to plot on the x-axis.

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"` or `"Column data"`. 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.
- `.allowableYAxisChoices(x,se)` returns a character vector specifying the acceptable variables in `colData(se)` that can be used as choices for the y-axis.

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 `"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.

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.

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:

- `.hideInterface(x,field)` returns `TRUE` if `field="DataBoxOpen"`, otherwise it calls `.hideInterface,Table-me`

- `.fullName(x)` will return the full name of the panel class.
- `.panelColor(x)` will return the specified default color for this panel class.

For defining the panel name:

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

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.

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 control various aesthetics of the dots and to store the brush or lasso selection.

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 `.refineParameters` below).

- `ColorByFeatureNameAssay`, a string specifying the assay of the SummarizedExperiment object containing values to use for coloring, if `ColorBy="Feature name"`. Defaults to the name of the first valid assay (see `?".refineParameters,DotPlot-method"` for details).
- `ColorBySampleNameColor`, a string specifying the color to use for coloring an individual sample on the plot, if `ColorBy="Sample name"`. Defaults to `"red"`.

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.

In addition, this class inherits all slots from its 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:

- `.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.
- `.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"`, `"RowSelectionType"` and `"RowSelectionSaved"`), 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:

- `.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.

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:

- `.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.

`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. It provides observers for monitoring table selection, global search and column-specific search.

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"`, `"RowSelectionType"` and `"RowSelectionSaved"`), 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.

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 which assay to display in the main heatmap, and which metadata variables to display as row and column heatmap annotations.

Slot overview

The following slots control the assay that is used:

- `Assay`, string specifying the name of the assay to use for obtaining expression values. Defaults to the first valid assay name (see ?"`.refineParameters,ComplexHeatmapPlot-method`" for details).
- `CustomRows`, a logical scalar indicating whether the custom list of features should be used. If `FALSE`, the incoming selection is used instead. Defaults to `TRUE`.
- `CustomRowsText`, string specifying a custom list of features to use, as newline-separated row names. If `NA`, defaults to the first row name of the SummarizedExperiment object.

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 `columnAnnotation`. Defaults to `character(0)`.

The following slots control the clustering of features:

- `ClusterRows`, a logical scalar indicating whether features should be clustered by assay data. 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 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"`.
- `LegendPosition`, string specifying the position of the legend on the plot. Defaults to `"Bottom"` but can also be `"Right"`.
- `LegendDirection`, string specifying the orientation of the legend on the plot for continuous covariates. Defaults to `"Horizontal"` but can also be `"Vertical"`.

The following slots control the effect of the transmitted selection from another panel:

- `SelectionEffect`, a string specifying the selection effect. This should be one of `"Color"` (the default), where all selected points change to the specified color; `"Restrict"`, where all non-selected points are not plotted.
- `SelectionColor`, a string specifying the color to use for selected points when `SelectionEffect="Color"`. Defaults to `"red"`.

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:

- `.refineParameters(x,se)` returns `x` after replacing `NA` values in `"Assay"` with the first valid assay name; and `NA` values 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:

- `.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 Panel method.

For defining the panel name:

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

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)
```

createCustomPanels *Create custom panels*

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",
  where = topenv(parent.frame())
)

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

Arguments

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

createLandingPage *Create a landing page*

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. Note that only Panel classes that were in the original `iSEE()` call (as the original `initial` or in the `extra`) will be used for security and other reasons.

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; and `INITIAL`, a list of Panel objects describing the initial application state. If `INITIAL=NULL`, the initial state from `initial` is used instead.
- `input`, the Shiny input list.
- `output`, the Shiny output list.
- `session`, the Shiny session object.

The 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[grep("Data$", all.data)]

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

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

`dataframe-utils` *Class utilities*

Description

`.findAtomicFields`: A utility function to find columns in a data.frame or DataFrame that are atomic R types, as most of the app does not know how to handle more complex types being stored as columns. An obvious example is in data.frames expected by `ggplot` or `datatable`.

Usage

```
.findAtomicFields(x)

.whichGroupable(x)

.whichNumeric(x)
```

Arguments

x	A data.frame or DataFrame.

Details

`.whichNumeric`: Identify continuous columns that can be used as options in various interface elements, e.g., for sizing. This is typically called in `.cacheCommonInfo` for later use by methods of `.defineInterface`.

`.whichGroupable`: Identify categorical columns that can be used as options in various interface elements, e.g., for faceting or shaping. This is typically called in `.cacheCommonInfo` for later use by methods of `.defineInterface`.

Value

`.findAtomicFields`: A character vector of names of atomic fields in `x`.

`.whichGroupable`: An integer vector containing the indices of the categorical columns.

`.whichNumeric`: An integer vector containing the indices of the numeric columns.

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)
```

`defaults` *Deprecated default functions*

Description

These default-creating functions are deprecated and should be replaced by constructor calls.

Usage

```
redDimPlotDefaults(se, number)

featAssayPlotDefaults(se, number)

colDataPlotDefaults(se, number)

rowStatTableDefaults(se, number)

colStatTableDefaults(se, number)

rowDataPlotDefaults(se, number)

sampAssayPlotDefaults(se, number)

heatMapPlotDefaults(se, number)
```

Arguments

se	A SummarizedExperiment object.
number	Integer scalar specfiying the number of panels to use.

Details

The iSEE function will attempt to translate the default parameters here into a the new S4 framework based on the Panel and its subclasses.

Value

A DataFrame containing default settings for parameters of each of `number` feature assay panels.

Author(s)

Aaron Lun

Examples

```
example(SingleCellExperiment, echo=FALSE) # mock up 'sce'.
redDimPlotDefaults(sce, n=1)
featAssayPlotDefaults(sce, n=1)
colDataPlotDefaults(sce, n=1)

rowStatTableDefaults(sce, n=1)
colStatTableDefaults(sce, n=1)

rowDataPlotDefaults(sce, n=1)
sampAssayPlotDefaults(sce, n=1)
heatMapPlotDefaults(sce, n=1)
```

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 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"`.
- `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 `"---"`.

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:

- `FacetByRow`, a string specifying the metadata field to use for creating row facets. For RowDotPlots, this should be a field in the `rowData`, while for ColumnDotPlots, this should be a field in the `colData`. Defaults to `"---"`, i.e., no row faceting.
- `FacetByColumn`, a string specifying the metadata field to use for creating column facets. For RowDotPlots, this should be a field in the `rowData`, while for ColumnDotPlots, this should be a field in the `colData`. Defaults to `"---"`, i.e., no column faceting.

The following slots control the effect of the transmitted selection from another panel:

- `SelectionEffect`, a string specifying the selection effect. This should be one of `"Transparent"` (the default), where all non-selected points become transparent; `"Color"`, where all selected points change to the specified color; `"Restrict"`, where all non-selected points are not plotted.
- `SelectionAlpha`, a numeric scalar in [0, 1] specifying the transparency to use for non-selected points when `SelectionEffect="Transparent"`. Defaults to 0.1.
- `SelectionColor`, a string specifying the color to use for selected points when `SelectionEffect="Color"`. Defaults to `"red"`.

The following slots control the behavior of brushes:

- `ZoomData`, a named numeric vector of plot coordinates with `"xmin"`, `"xmax"`, `"ymin"` and `"ymax"` elements parametrizing 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.

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 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.
- 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.

The following slots refer to general plotting parameters:

- FontSize, positive numeric scalar specifying the relative font size. Defaults to 1.
- LegendPosition, string specifying the position of the legend on the plot. Defaults to "Right" but can also be "Bottom".

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:

- .defineOutput(x,id) returns a UI element for a brushable plot.

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; and commands, a list of character vector containing the R commands required to generate contents and plot.

- `.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 testboxes.

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, i.e., is `SelectionEffect="Restrict"`?
- `.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)` 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`.

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

`assays`	List of colormaps for `assays`.
`colData`	List of colormaps for `colData`.
`rowData`	List of colormaps for `rowData`.
`all_discrete`	Colormaps applied to all undefined categorical `assays`, `colData`, and `rowData`, respectively.
`all_continuous`	Colormaps applied to all undefined continuous `assays`, `colData`, and `rowData`, respectively.
`global_discrete`	Colormap applied to all undefined categorical covariates.
`global_continuous`	Colormap applied to all undefined continuous covariates.
`...`	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. If the colormap can not immediately be found in the appropriate slot, `discrete` is a `logical(1)` that indicates whether the default colormap returned should be categorical `TRUE` or continuous (`FALSE`, default).

`assayColorMap(x, i, ..., discrete=FALSE)`: Get an `assays` colormap.

`colDataColorMap(x, i, ..., discrete=FALSE)`: Get a `colData` colormap.

`rowDataColorMap(x, i, ..., discrete=FALSE)`: Get a `rowData` colormap.

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_fpkm=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_fpkm")
assay(ecm, "cufflinks_fpkm") # 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 for specifying which feature to use and what to plot on the x-axis.

Slot overview

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

- `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 the first valid assay name (see `?".refineParameters,DotPlot-method"` for details).
- `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.
- `XAxis`, string specifying what should be plotting on the x-axis. This can be any one of `"None"`, `"Feature name"` or `"Column data"`. 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,ColumnDotPlo` for details).
- `XAaxisFeatureName`, 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.

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 `callNextMethod`.

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")
```

`interface-generics` *Generics for the panel interface*

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

In `.defineInterface(x,se,select_info)`, 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. Refer to the contract for each Panel class to determine what is already provided by each parent.

Defining the data parameter interface

In `.defineDataInterface(x,se,select_info)`, 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 method aims to provide a simpler alternative to specializing `.defineInterface` for the most common use case, where new panels wish to add their own interface elements for altering the contents of the panel. In fact, `.defineInterface,Panel-method` will simply call `.defineDataInterface` to populate the data parameter box.

It is the developer's responsibility to call `callNextMethod` to obtain interface elements for parent classes. Refer to the contract for each Panel class to determine what is already provided by each parent.

Hiding interface elements

In `.hideInterface(x,field)`, 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.

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. Refer to the contract for each Panel class to determine what is already provided by each parent.

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,
  redDimArgs = NULL,
  colDataArgs = NULL,
  featAssayArgs = NULL,
  rowStatArgs = NULL,
  rowDataArgs = NULL,
  sampAssayArgs = NULL,
  colStatArgs = NULL,
  customDataArgs = NULL,
  customStatArgs = NULL,
  heatMapArgs = NULL,
  redDimMax = 5,
  colDataMax = 5,
  featAssayMax = 5,
  rowStatMax = 5,
  rowDataMax = 5,
  sampAssayMax = 5,
  colStatMax = 5,
  customDataMax = 5,
  customStatMax = 5,
  heatMapMax = 5,
  initialPanels = NULL,
  annotFun = NULL,
  customDataFun = NULL,
  customStatFun = NULL,
  customSendAll = FALSE,
  colormap = ExperimentColorMap(),
  landingPage = createLandingPage(),
  tour = NULL,
  appTitle = NULL,
  runLocal = TRUE,
  voice = FALSE,
  bugs = FALSE,
  ...
)
```

Arguments

`se`	An object that is coercible to SingleCellExperiment. If missing, an app is launched with a landing page generated by the `landingPage` 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**.
`redDimArgs`	Deprecated, use `initial` instead.
`colDataArgs`	Deprecated, use `initial` instead.
`featAssayArgs`	Deprecated, use `initial` instead.
`rowStatArgs`	Deprecated, use `initial` instead.
`rowDataArgs`	Deprecated, use `initial` instead.
`sampAssayArgs`	Deprecated, use `initial` instead.
`colStatArgs`	Deprecated, use `initial` instead.
`customDataArgs`	Deprecated, use `initial` instead.
`customStatArgs`	Deprecated, use `initial` instead.
`heatMapArgs`	Deprecated, use `initial` instead.

`redDimMax`	Deprecated and ignored.
`colDataMax`	Deprecated and ignored.
`featAssayMax`	Deprecated and ignored.
`rowStatMax`	Deprecated and ignored.
`rowDataMax`	Deprecated and ignored.
`sampAssayMax`	Deprecated and ignored.
`colStatMax`	Deprecated and ignored.
`customDataMax`	Deprecated and ignored.
`customStatMax`	Deprecated and ignored.
`heatMapMax`	Deprecated and ignored.
`initialPanels`	Deprecated, use `initial` instead.
`annotFun`	Deprecated and ignored. Read the book for the new, more extensible approach for creating custom panels.
`customDataFun`	Deprecated and ignored. Read the book for the new, more extensible approach for creating custom panels.
`customStatFun`	Deprecated and ignored. Read the book for the new, more extensible approach for creating custom panels.
`customSendAll`	Deprecated and ignored.
`colormap`	An ExperimentColorMap object that defines custom colormaps to apply to individual `assays`, `colData` and `rowData` covariates.
`landingPage`	A function that renders a landing page when `se` is started without any specified `se`. See `createLandingPage` for more details.
`tour`	A data.frame with the content of the interactive tour to be displayed after starting up the app.
`appTitle`	A string indicating the title to be displayed in the app. If not provided, the app displays the version info of `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., `c(bugs=3L,spiders=1L)`).
`...`	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 typicall 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).

The tour argument needs to be provided in a form compatible with the format expected by the rintrojs package. There should be two columns, element and intro, with the former describing the element to highlight and the latter providing some descriptive text. See https://github.com/carlganz/rintrojs#usage for more information.

By default, categorical data types such as factor and character are limited to 24 levels, beyond which they are coerced to numeric variables for faster plotting. This limit may be set to a different value as a global option, e.g. options(iSEE.maxlevels=30).

The default landing page allows users to upload their own RDS files 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.maxRequestSiz

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.

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>

---

iSEEOptions *Default panel options*

Description

Set global options using `iSEEOptions$set()`, so that all relevant panels will use these default values during initialization.

Usage

```
iSEEOptions
```

Format

An object of class `list` of length 4.

Details

See `str(iSEEOptions$get())` for a list of default panel options.

Note that `iSEEOptions$restore()` can be used to 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"`).

`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.

`assay` Character vector of assay names to use if available, in order of preference.

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

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

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` *Find rows of data within a closed lasso*

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 **mgcv** 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

`cmd`	A character vector containing commands to be executed.
`envir`	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)`.

In practice, 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. This can be intersected with additional requirements in this function.

Allowable x-axis choices

`.allowableXAxisChoices(x,se)` is the same as above but for the variables to show on the x-axis. This need not return the same subset of variables as `.allowableYAxisChoices`.

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.

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`. 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.

As an example, in DotPlots, the method for this generic would return the contents of the `BrushData` slot.

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`. This defaults to `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 ReducedDimensionPlots 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 pertain to how `x` responds to a transmitted 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, in DotPlots, the method for this generic would return `TRUE` if `SelectionEffect="Restrict"`. 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.

Author(s)

Aaron Lun

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

Description

An overview of the generic for defining the panel observers, 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 and `SLOT` is the name of the slot modified by the interface element; so observers should respond to those names 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.

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`, 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

In `.defineOutput(x,...)`, the following arguments are required:

- `x`, an instance of a Panel subclass.
- `...`, further arguments that are not currently used.

Methods for this generic are expected to return an output element for inclusion into the **iSEE** interface, such as the output of `plotOutput`. Multiple elements can be provided 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

In `.renderOutput(x,se,...,output,pObjects,rObjects)`, 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 a reactive expression to `output` to render the output elements created by `.defineOutput`. The return value of this generic is not used; only the side-effect of the output set-up is relevant.

The rendering expression defined in `.renderOutput` is also 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. Fill `pObjects$contents[[PANEL]]` with some content related to the displayed output that allows cross-referencing with single/multiple selection structures. This will be used in other generics like `.multiSelectionCommands` and `.singleSelectionValue` to determine the identity of the selected point(s). As a result, it is only strictly necessary if the panel is a potential transmitter, as determined by the return value of `.multiSelectionDimension`.
3. Fill `pObjects$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.

4. 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 tasks 1-3 above, rather than calling `.generateOutput` manually. This means that the only extra work required in the implementation of `.renderOutput` is to perform task 4 and to choose an appropriate rendering function.

Generating content

In `.generateOutput(x,se,all_memory,all_contents)`, 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). This is used during app initialization to ensure that `pObjects$contents` of transmitter panels is filled before rendering their children. It should be of a structure that allows it to be cross-referenced against the output of `.multiSelectionActive` to determine the selected rows/columns.
- `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`.

The output list may contain any number of other fields that will be 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. Developers should consider using the `.processMultiSelections` function for easily processing the multiple selection parameters.

Exporting content

In `.exportOutput(x,se,all_memory,all_contents)`, 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 handling the contents appropriately.

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 to control the choice of transmitting panels from which to receive a multiple selection (i.e., a selection of multiple rows or columns from a SingleCellExperiment).

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.
- `PanelHeight`, an integer scalar specifying the height of the panel in pixels. This is expected to lie between 400 and 1000; defaults to 500.

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 `"---"`.
- `RowSelectionType`, a string specifying which of the multiple row selections from the transmitting panel should be used. Takes one of `"Active"`, only the active selection; `"Union"`, the union of active and saved selections; and `"Saved"`, one of the saved selections. Defaults to `"Active"`.
- `RowSelectionSaved`, an integer scalar specifying the index of the saved multiple row selection to use when `RowSelectionType="Saved"`. Defaults to 0.
- `DynamicRowSelectionSource`, a logical scalar indicating whether `x` should dynamically change its selection source for multiple row selections.

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 `"---"`.
- `ColumnSelectionType`, a string specifying which of the column-based selections from the transmitting panel should be used. Takes one of `"Active"`, only the active selection; `"Union"`, the union of active and saved selections; and `"Saved"`, one of the saved selections. Defaults to `"Active"`.
- `ColumnSelectionSaved`, an integer scalar specifying the index of the saved multiple column selection to use when `ColumnSelectionType="Saved"`. Defaults to 0.
- `DynamicColumnSelectionSource`, a logical scalar indicating whether `x` should dynamically change its selection source for multiple column selections.

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.

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`.

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)` is a no-op, returning `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 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)`.
- `.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.

`plot-generics` *Plotting generics*

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

In `.generateDotPlotData(x,envir)`, 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`.

The method 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 `NA`s 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 should be prefixed with `.` to avoid potential clashes with reserved variable names in the rest of the application.

Generating the ggplot object

In `.generateDotPlot(x,labels,envir)`, 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 ue 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 `SelectionEffect="Restrict"`, `plot.data` will already have been subsetted 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 `SelectionEffect="Restrict"`. 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 uninuitive 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.

Prioritizing points

In `.prioritizeDotPlotData(x,envir)`, 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 peform more aggressive downsampling for unimportant points.

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

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**.

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(FacetByRow = "Covariate_1", FacetByColumn = "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 for specifying which dimensionality reduction result to use.

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.

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)
```

`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 row metadata variable to use and what to plot on the x-axis.

Slot overview

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

- `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"` or `"Row data"`. 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.
- `.allowableYAxisChoices(x, se)` returns a character vector specifying the acceptable variables in `rowData(se)` that can be used as choices for the y-axis.

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 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.

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 *The RowDataTable panel*

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.

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:

- `.hideInterface(x,field)` returns TRUE if `field="DataBoxOpen"`, otherwise it calls `.hideInterface,Table-me`
- `.panelColor(x)` will return the specified default color for this panel class.

For defining the panel name:

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

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.

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 control various aesthetics of the dots and to store the brush or lasso selection.

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 field.

- `ColorBySampleNameAssay`, a string specifying the assay of the SummarizedExperiment object containing values to use for coloring, if `ColorBy="Sample name"`. Defaults to the name of the first assay.
- `ColorByFeatureNameColor`, a string specifying the color to use for coloring an individual sample on the plot, if `ColorBy="Feature name"`. Defaults to `"red"`.

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.

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:

- `.defineInterface(x,se,select_info)` defines the user interface for manipulating all slots in the RowDotPlot. It will also create a data parameter box that can respond to specialized `.defineDataInterface`. This will *override* the Panel method.
- `.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"`, `"RowSelectionType"` and `"RowSelectionSaved"`), 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:

- `.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.

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:

- `.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. It provides observers for monitoring table selection, global search and column-specific search.

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"`, `"ColumnSelectionType"` and `"ColumnSelectionSaved"`), 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.

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 ColumnDotPlot 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 which sample to use and what to plot on the x-axis.

Slot overview

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

- `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 the first valid assay name (see `?".refineParameters,DotPlot-method"` for details).
- `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.
- `XAxis`, string specifying what should be plotting on the x-axis. This can be any one of `"None"`, `"Sample name"` or `"Column data"`. 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,ColumnDotPlo` for details).
- `XAaxisSampleName`, 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.

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 `callNextMethod`.

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)
```

`setup-generics` *Generics for setting up parameters*

Description

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

Caching common information

In `.cacheCommonInfo(x,se)`, 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 `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.

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`.

Remember, the cache is strictly for use in defining interface elements and in observers. Developers should not expect to be able to retrieve cached values when rendering the output for a panel, as the code tracker does not capture the code used to construct the cache.

Refining parameters

In `.refineParameters(x,se)`, 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` does not know about `se`; thus, certain slots (e.g., for the row/column names) cannot be set to a reasonable default or checked at that point.

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 previously cached valid values. Of course, any slots that are not `se`-dependent should 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 selection contains a single "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. If no row or column is selected, it should return NULL.

contents is any arbitrary structure set by the rendering expression in .renderOutput for x. 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) should return a list with one internal list for each slot 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.

The paradigm here is that the interface will contain two selectInput elements, one for each of the param and source slots. It is possible for users to manually alter the choice in the param's selectInput; it is also possible for users to specify a transmitting panel via the source's selectInput, which will then trigger automatic updates to 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. This 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, 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 this is NA, the usage mode is assumed to be such that the panel should always respond to transmissions.
- 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.
- 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

`subsetPointsByGrid` *Subset points for faster plotting*

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 *Synchronize assay colormaps to match those in a SummarizedExperiment*

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` *The Table class*

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 monitoring table selection, global search and column-specific search.

Slot overview

The following slots control aspects of the `DT::datatable` interface:

- `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 character vector where each entry contains the search string for each column. Defaults to an empty character vector, i.e., no search.

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,id)` returns a UI element for a `dataTableOutput` widget.

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 testboxes.
- `.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; and `commands`, a list of character vector containing the R commands required to generate `contents` and `plot`.
- `.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,pObjects)` returns the name of the row that was last selected in the `datatable` widget.

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.

`table-generics` *Generics for table construction*

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

In `.generateTable(x,envir)`, 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.

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

`panel_name`	String containing the panel name.
`rObjects`	A reactive list of values generated in the `iSEE` app.

Details

`.trackUpdate` will track whether an update has been requested to the current panel (see also `.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.

Value

All functions will cause the current reactive context to respond to the designated event. `NULL` is returned invisibly.

Author(s)

Aaron Lun

visual-parameters-generics

Generics for visual parameters

Description

These generics allow subclasses to override the user inputs controlling visual parameters of DotPlot panels.

Details

In practice, 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.

It is possible to hide individual sections of visual parameters by returning `NULL`.

Color parameters

`.defineVisualColorInterface(x,se,select_info)` takes `x`, a Panel instance, `se`, the SummarizedExperiment object, and `select_info` a list of character vectors named `row` and `column` which specifies the names of panels available for transmitting single selections on the rows/columns. It is expected to return an HTML tag definition that contains user inputs controlling the `color` aesthetic of `ggplot` objects.

Shape parameters

`.defineVisualShapeInterface` takes `x`, a Panel instance, and `se`, the SummarizedExperiment object. It is expected to return an HTML tag definition that contains user inputs controlling the `shape` aesthetic of `ggplot` objects.

Size parameters

`.defineVisualSizeInterface` takes `x`, a Panel instance, and `se`, the SummarizedExperiment object. It is expected to return an HTML tag definition that contains user inputs controlling the `size` aesthetic of `ggplot` objects.

Point parameters

`.defineVisualPointInterface` takes `x`, a Panel instance, and `se`, the SummarizedExperiment object. It is expected to return an HTML tag definition that contains user inputs to controlling other aesthetics of `ggplot` objects (e.g. `alpha`).

Facet parameters

`.defineVisualFacetInterface` takes `x`, a Panel instance, and `se`, the SummarizedExperiment object. It is expected to return an HTML tag definition that contains user inputs controlling the `facet_grid` applied to `ggplot` objects.

Text parameters

`.defineVisualTextInterface` takes `x`, a Panel instance, and `se`, the SummarizedExperiment object. It is expected to return an HTML tag definition that contains user inputs controlling the appearance of non-data text elements of `ggplot` objects (e.g., font size, legend position).

Other parameters

`.defineVisualOtherInterface` takes `x`, a Panel instance. It is expected to return an HTML tag definition that contains user inputs to display in the `"Other"` section of the visual parameters.

Author(s)

Kevin Rue-Albrecht

See Also

`tagList`

Index