Output Widgets

In a GUI, the fillings are known as **widgets**. There are lots of different widgets to choose from, each suited to a specific task - we've grouped them into output widgets & input widgets.

Output widgets are used for displaying information to a user.

They usually provide three functions:

- **ADD** - this creates the widget
- **GET** - this gets the contents of the widget
- **SET** - this changes the contents of the widget

As well as options to change the way they look/act.

For each of the above to work, we need to know which widget you are referring to - so every widget gets a **unique title**.

Label

Labels are used for displaying text in the GUI.

- They are great for titles, at the top of the GUI, usually spanning multiple columns.
- They are really useful before *Entries* and *Drop-downs* to explain their purpose.
- And, they're very helpful at the bottom of the GUI, to show the results of an action.

Add Labels

- `.addLabel(title, text=None)`
  This will create a label widget to display text in the GUI.
  The `title` is used to uniquely identify the label, in case you want to change it later, and the `text` is what gets displayed.
  If `text` is set to None, or no `text` is provided, the `title` will be displayed in the label.
- `.addEmptyLabel(title)`
  Does the same as add a *label*, except there's no parameter to set any text.
- `.addSelectableLabel(title, text=None)`
  This adds a label whose text can be selected with the mouse.
  This is really just a *read-only* Entry, disguised to look like a label.
  But it seems to do the trick...
- `.addFlashLabel(title, text=None)`
  This adds a flashing *label*, that will alternate between the foreground and background colours.

Set Labels

- `.setLabel(title, text)`
  Change the contents of the *label*.
- `.clearLabel(title)`
  Clear the contents of the *label*.
- `.clearAllLabels()`
  Clears the contents of all *labels*.

Get Labels

- `.getLabel(title)`
  Get the contents of the *label*.

Auto-Labelled Widgets

It's possible to automatically include a *label* alongside a lot of the widgets.
Both the label and widget will be placed in the same grid space.
Simply add the word `Label` to the command when adding the widget:

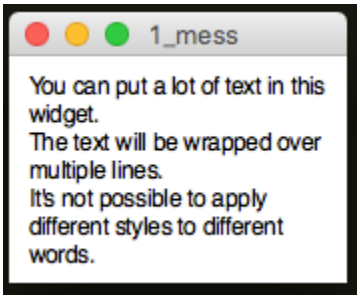
- `.addLabelEntry(title)`
- `.addLabelNumericEntry(title)`
- `.addLabelSecretEntry(title)`
- `.addLabelAutoEntry(title, words)`
- `.addLabelScale(title)`
- `.addLabelOptionBox(title, values)`
- `.addLabelTickOptionBox(title, values)`

- `.addLabelSpinBox(title, values)`
- `.addLabelSpinBoxRange(title, from, to)`

See the relevant section for a description of what the widget does.

Message

Very similar to a Label, except it will wrap the text over multiple lines.
By default the text is laid out 50% wider than it is high.
This can be changed by setting a specific `width` or by changing the `aspect` ratio.



Add Messages

- `.addMessage(title, text)`
  Adds a Message widget, with the specified text.
  If not text is provided, the title will be used for the text.
- `.addEmptyMessage(title)`
  Adds an empty Message widget.

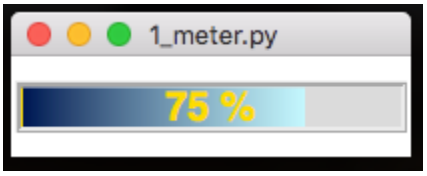
Set Messages

- `.clearMessage(title)`
  Clears the specified Message widget.
- `.setMessage(title, text)`
  Sets the contents of the specified Message widget, to the specified text.
- `.setMessageAspect(title, aspect)`
  Sets the aspect ratio at which text is wrapped.
  The default is 150, which means the text will be 50% wider than it is high.
  Ignored if a `width` has been set.
- `.setMessageWidth(title, width)`
  Sets the number of characters per line for the widget.
  If not set, width is calculated using the default aspect ratio.

Meter

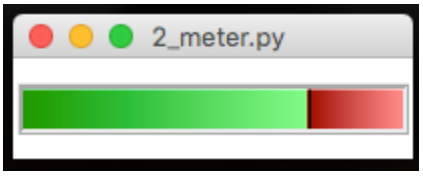
Various styles of progress meter:

- Meter

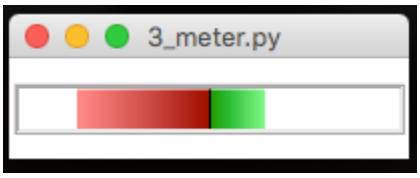
  A simple meter for showing progress from 0% to 100%.

- SplitMeter

  A possession style meter, showing percentages on either side.

- DualMeter

  Two separate meters, expanding out from the middle.

Add Meters

- `.addMeter(name)` & `.addSplitMeter(name)` & `.addDualMeter(name)`
  Adds a meter with the specified name, of the specified type.

Set Meters

- `.setMeter(name, value, text=None)`
  Changes the specified meter to the specified value.
  For `Meter` & `SplitMeter` should be a value between 0 and 100.
  For `DualMeter` should be a list of two values, each between 0 and 100.

- `.setMeterFill(name, colour)`
  Changes the fill colour of the specified meter.
  For `SplitMeter` & `DualMeter` should be a list of two colours.

Get Meters

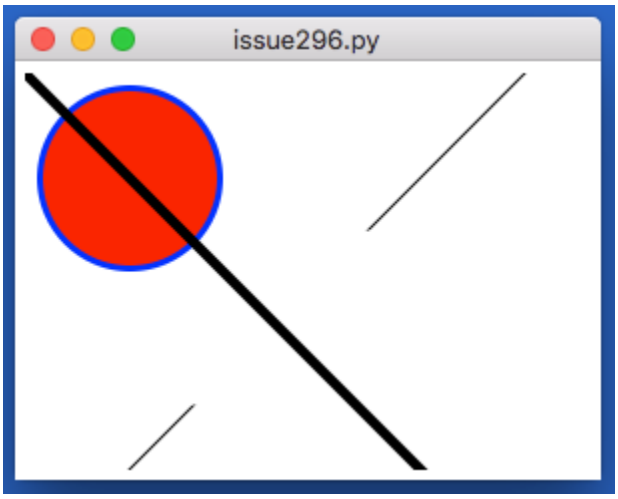
- `.getMeter(name)`
  Gets the value of the specified meter.
  As meters convert their data to a value between 0 and 1, this will return a list of two values: `(0.45, '45 %')`
- `.getAllMeters()`
  This will return the contents of all Meters in the app, as a dictionary.

Background Processing

Meters are designed to show progress over time.
One common solution is to register a function that is constantly updating a meter.
This should then be monitoring/updating a global variable:

Canvas

This lets you embed a canvas in appJar.
Canvases are very powerful, appJar will never provide wrappers for all their functions.
So, if you're looking to truly harness a canvas, add it and save the widget as a variable: `canvas = app.addCanvas("c1")`. Then, you can call all the canvas functions as you would a tKinter canvas.



Or, as mentioned above, you can work directly with the canvas object:

- `.addCanvas(title)`
  Creates a canvas widget.
- `.getCanvas(title)`
  Gets the specified canvas widget.

Setting a Canvas

- `.setCanvasMap(title, func, coords)`
  It is possible to set up a simple CanvasMap - a clickable canvas, with names linked to different areas. When one of those areas is clicked, a function will be called, passing the name of the area as a parameter. `coords` must contain a dictionary of areas on the map. When a position on the canavs is clicked, in one of the areas, the named function will be called, passing in the area's name. When an unknown position on the canavs is clicked, UNKNOWN will be passed to the function, along with the coordinates.

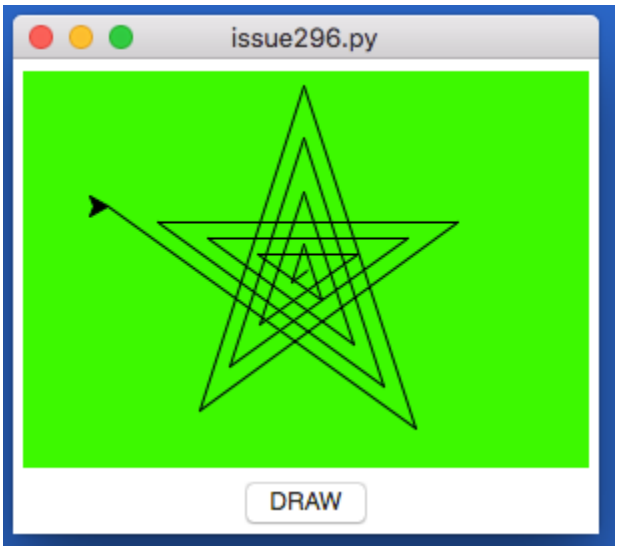
Drawing on a Canvas

NB. each of these functions returns the object being created, so you can later change it:

- `.addCanvasCircle(title, x, y, diameter, **kwargs)`
  Draws a circle on the canvas.
- `.addCanvasOval(title, x, y, xDiam, yDiam, **kwargs)`
  Draws an oval on the canvas.
- `.addCanvasRectangle(title, x, y, w, h, **kwargs)`
  Draws a rectangle on the canvas.
- `.addCanvasLine(title, x, y, x2, y2, **kwargs)`
  Draws a line on the canvas.
- `.addCanvasText(title, x, y, text, **kwargs)`
  Draws text on the canvas.
- `.addCanvasImage(title, x, y, image, **kwargs)`
  Draws the specified image on the canvas.
- `.clearCanvas(title)`
  Removes all items from the canvas.

Turtle

This lets you embed a turtle widget in appJar.



- `.addTurtle(title)`
  Creates a turtle widget.
- `.getTurtle(title)`
  Gets the specified turtle widget.
- `.getTurtleScreen(title)`
  Gets the screen behind the turtle widget.