guietta

Release 0.3.8

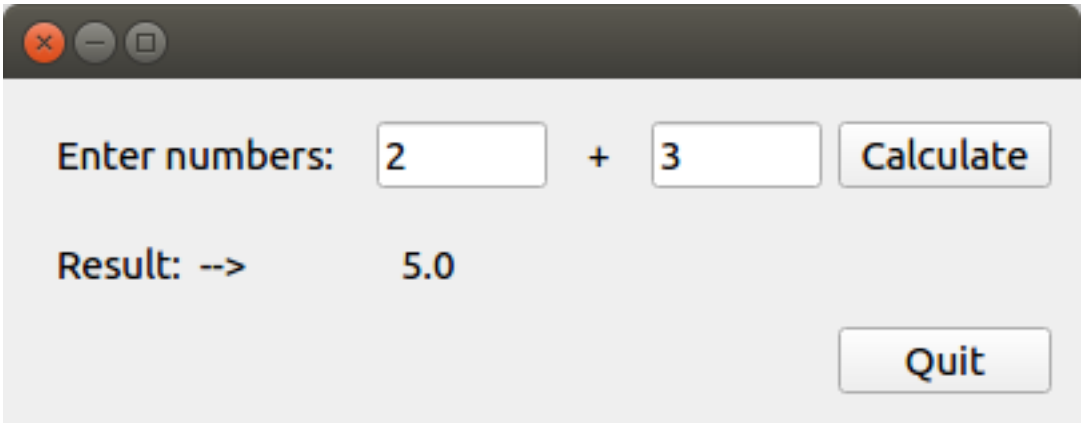
July 20, 2020

Contents

Guietta is a tool that makes simple GUIs *simple*:

```
from guietta import _, Gui, Quit

gui = Gui(

  [  'Enter numbers:', '__a__'  , '+' , '__b__',  ['Calculate'] ],
  [  'Result:  -->'  , 'result' ,  _  ,    _   ,       _        ],
  [  _               ,    _     ,  _  ,    _   ,      Quit      ]
)

with gui.Calculate:
    gui.result = float(gui.a) + float(gui.b)

gui.run()
```

And here it is:



Also featuring:

- matplotlib and pyqtgraph integration, for easy event-driven plots
- easily display columns of data in labels using lists and dicts
- multiple windows
- customizable behaviour in case of exceptions
- queue-like mode (a la PySimpleGUI)
- integrate any QT widget seamlessly, even your custom ones (as long as it derives from QWidget, it is OK)
- easy background processing for long-running operations
- ordinary QT signals/slots, accepting any Python callable, if you really want to use them

CHAPTER 1

Installation

pip install guietta

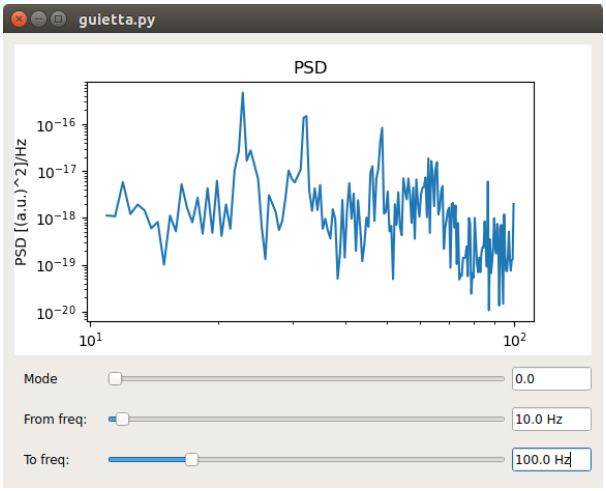
1.1 Install on older platforms

Guietta uses the PySide2 QT5 binding by default, and some systems (older Macs, Raspberry PI) do not have it available. Guietta can fallback to the PyQt5 binding if available, but does not specify it as an automatic dependency. If you get an installation error about PySide2, try to use PyQt5 instead using the following:

```
pip install guietta --no-deps
pip install pyqt5
```

CHAPTER 2

Screenshots



Guietta at work in a scientific lab, showing an interactive plot.

CHAPTER 3

Tutorial

Direct link to the tutorial: tutorial.html.

See also the rest of the documentation below.

CHAPTER 4

Source code

https://github.com/alfiopuglisi/guietta

CHAPTER 5

Troubleshooting

Guietta uses Qt5, and some Linux distributions, like Ubuntu 16.04, appear to have an incomplete default installation. If you encounter trouble running guietta, please read the troubleshooting guide.

If you use conda, please read our page on QT incompatibilities with conda.

CHAPTER 6

Documentation

6.1 Introduction

6.1.1 The problem with GUIs

If you have worked with any GUI framework, like the excellent QT library, you will have noticed a very common problem: in order to make difficult things possible, simple things become difficult.

Guietta makes simple GUIs *simple*. For example, suppose that you want to make a GUI where the user enters two numbers, and when a button is clicked some complicated operation is performed, say, the numbers are added up, and the result displayed back to the user. With plain QT, you have two choices:

1. Start Qt Designer up and build your GUI using drag and drop. Designer saves an .ui file, which can be converted using pyuic to a python module, which can be imported into your program. . . .
2. Skip Designer and do things programmatically: create a layout, then create your three or four widgets, then add them to the layout. Ah wait, should we build a row with HBoxLayout and join several of them into a VBoxLayout, or is it the other way around? How things are going to line up?

It's not over. In both cases, now define a function that does the calculation, connect the right signal to the right slot (what was the signal name again?), etc etc. . . .

I've given up already.

Using Guietta's compact syntax, here is how the layout looks like:

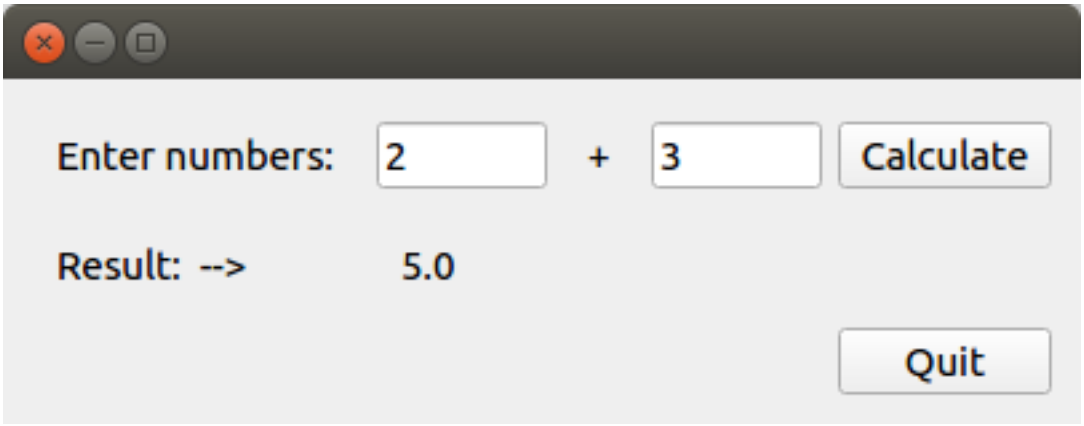
```
from guietta import _, Gui, Quit

gui = Gui(

  [  'Enter numbers:', '__a__'  , '+' , '__b__',  ['Calculate'] ],
  [  'Result:  -->'  , 'result' ,  _  ,    _   ,       _        ],
  [  _               ,    _     ,  _  ,    _   ,      Quit      ] )
```

can you *see* the GUI? Right in the code? We can add then the behaviour with a few more lines:

```
with gui.Calculate:
   gui.result = float(gui.a) + float(gui.b)

gui.run()
```

That's enough to get it working! That was 11 lines in total, including a few blank ones for clarity.

And here is the result on my computer:



It's *that* simple.

Note: for IPython users: if you type the previous example on the command prompt, it may or may not work (it appears to work on recent ipython versions). IPython does strange things with the command history. If it does not work, put the example into a file and run it.

6.1.2 What Guietta is

Guietta is actually a thin wrapper over QT. It allows one to quickly create QT widgets, assemble them into a layout, and make it responsive to user input. All standard QT features are available, including signals/slots and the event loop, so if you know enough QT, you can do some pretty amazing things.

And if you have a big graphical program with multiple windows and pop-ups, you can use a subset of Guietta to simplify the window creation where it makes sense.

It also works in the other direction: if you have a big complicated custom QT widget, and you need to insert it into a dialog together with some more buttons, Guietta can do that too.

6.1.3 What Guietta is not

Guietta is a tool for making *simple* GUIs. You will not able to use it to make the next version of Photoshop, not even the next version of MS Paint. It does not do super-fancy layouts, or cinema-like animations.

6.1.4 Aren't you just copying from PySimpleGUI?

PySimpleGUI is a much bigger project with a bigger goal: produce a GUI framework that can work with multiple interfaces (QT, TKinter, even web-based) and is easy to use for the beginner. PySimpleGUI's simplified syntax was a great idea and it was the inspiration for Guietta, but it stopped to soon. Guietta goes much further in simplifying things, and as a result it has less features than PySimpleGUI.

6.1.5 The layout doesn't respect PEP8!

Alas, no. Laying out GUIs with code was not foreseen when PEP8 was written.

Next topic: the tutorial.

6.2 Tutorial

6.2.1 What you need

Python 3.5 or newer

6.2.2 Install Guietta

```
pip install guietta
```

This should also automatically install PySide2, if you don't already have it. If you plan to use Matplotlib together with guietta, you should install that too. It is not done automatically.

6.2.3 Should I learn QT before starting?

No. Knowing QT will make it easier to digest the most advanced topics, but there is no need for it.

6.2.4 Quickstart

We will work with a simple assignment: make a GUI application that, given a number, doubles it. Here is how we begin:

```
from guietta import Gui, _

gui = Gui( [ 'Enter number' , '__num__' , ['Go'] ],
           [ 'Result ---> ' , 'result'  ,   _    ] )
gui.run()
```

This code is enough to display a GUI with three widgets. Let's see what each line does.

```
from Guietta import Gui, _
```

Every GUI made with Guietta is an instance of the `Gui` class, so we need to import it. We also use the special underscore _, explained later, so we import it too.

```
gui = Gui( [ 'Enter number' , '__num__' , ['Go'] ],
           [ 'Result ---> ' , 'result'  ,   _    ] )
```

Arguments to the `Gui` constructor define the GUI layout. The layout is specified with a series of Python lists, one for each widget row. Our example has two rows, so there are two lists. Each list element must be a valid widget. Here we see four different ones:

- `'Enter number'`: a string will be converted to a simple text display (in GUI parlance it is called a *label*). It is possible to change the text later on.
- `'__num__'`: a string starting and ending with double underscores will be converted to an edit box (imagine a form to be filled out). The edit box is initially empty.

- `['Go']`: a string inside square brackets will be converted to a button that the user can click. The button's text can also be changed later if wanted.
- _: an underscore means no widget.

Notice how we formatted the lists to keep things aligned. You are encouraged to use spaces to make the GUI layout visible right in the code.

The constructor will create all these widgets and arrange them in a regular grid. At this point, the GUI is ready to be displayed.

```
gui.run()
```

This line displays the GUI and starts the QT event loop. This function will not return until the GUI is closed (there are ways around this, and we will see them later). If you try the GUI, you will notice that the `Go` button does nothing, since we did not assign it any function. We will see how to do that in the next chapters.

6.2.5 Guietta's magic properties

Each GUI widget is assigned a name, which is usually automatically derived from the constructor: any text-based widget has a name that corresponds to the initial widget text. Special characters in the text are removed in order to be sure that the resulting name is a valid Python identifier. In practice, this means that letters a-Z, A-Z, numbers 0-9 and underscores are kept, and everything else is removed.

The GUI shown in the previous chapter will have five widgets: "Enternumber", "num", "Go", "Result", and "result". If a name is a duplicate, it is auto-numbered starting with the number 2.

All widgets are available in the `widgets` dictionary, so it is possible to use all ordinary QT methods:

```
gui.widgets['result'].setText('foo')
```

but Guietta makes it much easier, by automatically creating *magic properties* for the widgets:

```
gui.result = 'foo'
```

Magic properties are not real Python properties, because they are not defined in the class but instead are emulated using getattr/setattr, but behave in the same way. Properties can be read too:

```
labeltext = gui.Enternumber
```

Labels can be assigned anything:

```
gui.result = 3.1415
gui.result = ['a', 'b', 'c']
gui.result = dict(a=1, b='bar', c=None)
```

A list will result in a multi-line label, one element per line. A dictionary will be displayed using two columns, one item per line, with keys on the left column and values on the right one. Anything else is converted to a string using str() on the value being assigned.

Magic properties work for buttons and other widgets too, but with different object types. For example, a button will accept a callable that will be called when the button is clicked. A complete list of property types is available in the reference guide.

A special case where magic properties are also used are dynamical layouts, described in more detail *here*.

6.2.6 GUI actions

There are several ways to assign actions. Most rely on QT concept of *event*: an event is something that happens to a widget (for example, a button is clicked), and this event causes some piece of code to be executed. Events only work if the QT *event loop* has been started, which is done automatically by *gui.run()*. The "GUI queues" chapter later on describes how one can work without the event loop, if desired.

The events layer

The canonical Guietta way to specify events is to add a *layer* to the gui, using the *events()* method:

```
gui = Gui( [ 'Enter number' , '__num__' , ['Go'] ],
           [ 'Result ---> ' , 'result'  ,   _    ] )

gui.events([       _        ,     _     ,  recalc ],
           [       _        ,     _     ,   _     ] )
```

Notice how we have kept exactly the same layout for the Gui constructor and the events() method. This makes immediately visible that the *recalc* event has been assigned to the *Go* button, while other widgets are ignored.

An event assigned this way can be any Python callable, thus we need to define a *recalc* function before the gui is constructed:

```
def recalc(gui, *args):
    gui.result = float(gui.num)*2
```

The first argument to an event function is always the *gui* instance that generated the event. Other arguments may be added depending on the QT signal that generated the event. Since we are not interested in them, we put a generic *args there.

The recalc function is updating the Gui using the magic properties described in the previous chapter. Since the properties always return strings, it uses float() to convert the string to a number.

Custom events

In QT, a single widget can have several different events. For example, an edit box can trigger an event every single time the text is changed, or just when Return is pressed. Guietta assigns to each widget a *default event*, which is the one that makes sense most of the time (the list of default events for each widget is listed in the reference guide).

It is possible to specify a custom event using the tuple syntax:

```
gui = Gui( [ 'Enter number' , '__num__' , ['Go'] ],
           [ 'Result ---> ' , 'result'  ,   _    ] )

gui.events([       _  ,  ('textEdited', recalc) ,  recalc ],
           [       _  ,           _             ,   _     ] )
```

The tuple must be (*'event_name'*, *callable*). The event name must be a valid one for the widget, and a list can be found in the QT documentation, where it is called a *signal*. The QT documentation lists the possible signals for each widget, for example for edit boxes, in the "Signals" chapter.

Here we are assigning the *recalc* function to the *textEdited* event, which is fired every time the text in the editbox is updated by the user. Try it and you should see the value in the result label updating at every keystroke.

Assign a callable

Buttons (ordinary buttons, checkboxes and radio buttons, that in QT are all derived from QAbstractButton) can be assigned any python callable using Guietta's magic properties:

```
def handler(gui, *args):
    print('handler!')

gui.mybutton = handler
```

Due to how the QT signal/slots mechanism works, it is not possible to read the same property to get the button handler.

Automatic events

If your GUI events are relatively simple, you might be able to do away with their definitions entirely, using the *gui.auto* decorator:

```
@gui.auto
def recalc(gui, *args):
        gui.result = float(gui.num)*2

gui.run()
```

When the "auto" decorator is used, Guietta will inspect the function code, detect any property read like the *gui.num* above (but not the *gui.result*, which is a property store), and automatically connect the decorated function to the default event of that widget. Since the default event for editboxes is *returnPressed*, the above code will run every time the user presses *Return* on the editbox. The *Go* button at this point could be removed.

Notice that the *auto* decorator is a member of a *Gui* instance, and not a standalone one. Thus any decorated function must be declared after the gui is constructed.

Note: due to how the code introspection features work (from the standard library inspect module) the @auto decorator will not work on the Python command prompt.

The *with* statement

We saved the best for last. Enter the *with* statement:

```
with gui.Go:
   gui.result = float(gui.num)*2

gui.run()
```

The "with *magic property*" statement will save the code block and execute it when the corresponding widget, in this case the *Go* button, fires its default event.

Multiple *with* blocks can be defined, and multiple properties can be listed in a single with block, without limits.

While extremely simple and intuitive, this style has a number of caveats:

- signal arguments are not supported. The example above was a mouse click, but for example a *valueChanged()* signal from a slider would not have transferred the new slider value.
- the *as* clause in the *with* statement cannot be used.
- nested *with* statements will not work

- like the @auto decorator above, it is not guaranteed to work on a Python prompt. It works on the standard Python one, but for example will not work with some versions of IPython.
- the code inside *with* block is also executed once when it is encoureted for the first time, before *gui.run()* is called. This is unavoidable due to how code is parsed by Python. Most probably, it will generate an exception (in this case, because the gui.num content cannot be converted to a float object), and the guietta's *with* code block will discard all such exceptions.

It is possible to protect such a code block using Guietta's is_running attribute:

```
with gui.Go:
   if gui.is_running:
       gui.result = float(gui.num)*2
```

this way, one is sure that the code will be executed only under *gui.run()*, but most of the time there is no need.

Exception handling

You may have noticed that, in our events example above, there was no exception catching in the event functions. Guietta by default catches all exceptions and pops a warning up to the user if one happens. This behavior can be modified with the `guietta.Exceptions` enum, which has four values:

- Exceptions.POPUP: the default one, a warning popup is shown
- Exceptions.PRINT: the exception is printed on standard output
- Exceptions.SILENT: all exceptions are silently ignored
- Exceptions.OFF: no exception is caught, you have to do all the work.

The value must be given to the Gui constructor using the `exceptions` keyword argument:

```
from guietta import Gui, _, Exceptions

gui = Gui( [ 'Enter number' , '__num__' , ['Go'] ],
           [ 'Result ---> ' , 'result'  ,   _    ],
           exceptions = Exceptions.SILENT )    # Ignore exceptions
```

The `exceptions` keyword can also accept any Python callable. In this case, when an exception occurs the callable will be called with the exception as an argument.

GUI queues

A completely different way of getting events out of the guis is to use Guietta's *get()* method instead of *run()*.

With *get()*, the GUI behaves like a queue of *events*. These events are exactly the same as the ones we have seen before, but instead of triggering a function or a with block, they are put into an internal queue.

get() blocks until an event happens. It returns the name of the widget that generated the event, plus an *Event* object with additional information about the event:

```
name, event = gui.get()
```

By the way, *get()* automatically shows the GUI if had not been shown before.

If you try to call `gui.get()` and click on the *Go* button, you should see something like this:

```
>>> gui.get()
('Go', Event(signal=<bound PYQT_SIGNAL clicked of QPushButton object at␣
↪0x7fef88dc9708>, args=[False]))
```

(continues on next page)

(continued from previous page)

here we see that the event name was Go, as expected, and the Event object tells us some details about the QT signal. Most of the time, we do not need to even look at the detailed information.

If instead you call `gui.get()` and click the X to close the window, the result will be:

```
>>> gui.get()
(None, None)
```

This is how we discover that the user has closed the window.

> **Note:** if you have clicked multiple times on the *Go* button in between the *get()* calls, you will have to call `gui.get()` the same number of times before getting `(None, None)`, because you have to empty out the event queue.

The usual way of using *get()* is to put it into an infinite loop, breaking out of it when we get None:

```
while True:
    name, event = gui.get()

    if name == 'Go':
        print('You clicked Go!')
        gui.result = float(gui.num)*2

    elif name == None:
        break
```

It is important to keep whatever is done in the loop very short, because for the whole time we are outside *get()*, the GUI is not responsive to user clicks and will not be redrawn if dragged, etc.

A word on exceptions

If you have tried the previous code clicking *Go* without entering a number before, or entering something else like a letter, the loop will have exited with an exception caused by the failed *float()* call.

This happens because, when using *get()*, the exception catching described above is not used. Instead, we should be prepared to catch any exception generated by the code. Rather than using a big try/except for the whole loop, it is best to put the the exception handling right where it is needed, in order to be able to display a meaningful error message to the user. Something like this:

```
if name == 'Go':
    try:
        gui.result = float(gui.num)*2
    except ValueError as e:
        gui.result = e
```

Notice how we are displaying the error message right in the GUI.

Non-blocking *get*

The *get()* call shown before blocks forever, until an event arrives. However the call syntax is identical to the standard library queue.get call:

```
Gui.get(self, block=True, timeout=None)
```

If we pass a *timeout* argument (in seconds), the call will raise a `guietta.Empty` exception if *timeout* seconds have passed without a event. This feature is useful to "wake up" the event loop and perform some tasks regularly. Just for demonstration purposes, this loop re-uses the *Enter number* label to show a counter going up an 10 Hz. while still being responsive to the *Go* button:

```
from guietta import Empty

counter = 0
while True:
    try:
        name, event = gui.get(timeout=0.1)
    except Empty:
        counter += 1
        gui.Enternumber = counter
        continue

    if name == 'Go':
       try:
           gui.result = float(gui.num)*2
       except ValueError as e:
           gui.result = e

    elif name is None:
        break
```

Notice the `continue` statement in the except clause. If it was not there, execution would have progressed to the `if` statement below, and the handler for the *Go* button might be executed multiple times.

6.2.7 Using images

Labels and buttons can display images instead of text: just write the image filename as the label or button text, and if the file is found, it will be used as an an image. By default, images are searched in the current directory, but the *images_dir* keyword argument can be supplied to the `Gui` constructor to change it. So for example:

```
import os.path
from guietta import Gui, _

gui = Gui(

  [  _             , ['up.png']    , _              ],
  [  ['left.png'] ,     _          , ['right.png']  ],
  [  _             , ['down.png'] , _              ,  ],

  images_dir = os.path.dirname(__file__))
```

This code will display four image buttons arranged in the four directions, provided that you have four PNG images with the correct filename in the same directory as the python script. Notice how we use `os.path` to get the directory where our script resides.

6.2.8 Radio buttons

Radio buttons can be created using the R() widget, which stands for a QRadioButton() instance. By default, all radio buttons in a single Gui instance are exclusive. If multiple radio buttons are desired, Guietta makes available ten pre-

defined widgets classes called R0, R1, R2 ... R9, which will create radio buttons belonging to one of the 10 groups. For example:

```
from guietta import Gui, R1, R2, R3

gui = Gui(
    [ R1('rad1')  ,  R2('rad3')  , R3('rad5')   ],
    [ R1('rad2')  ,  R2('rad4')  , R3('rad6')   ],
)

gui.run()
```

That code creates six radio buttons, belonging to three different exclusive groups arranged vertically.

The radio buttons example shows how to connect radio buttons to events and how to check if a radio button is checked or not.

Pre-defined radio buttons were introduced in version 0.3.4.

6.2.9 Special layouts

Sometimes we would like for a widget to be bigger than the others, spanning multiple rows or columns. For example a label with a long text, or a horizontal or vertical slider, or again a plot made with Matplotlib should occupy most of the window. The following example introduces two new Guietta symbols, ___ (three underscores) and **III** (three capital letter i) which are used for horizontal and vertical expansion:

```
from guietta import Gui, _, ___, III, HS, VS

gui = Gui(

  [ 'Big label' ,    ___    ,       ___       ,  VS('slider1') ],
  [     III     ,    III    ,       III       ,       III      ],
  [     III     ,    III    ,       III       ,       III      ],
  [      _      , 'a label' , 'another label' ,        _       ],
  [HS('slider2'),    ___    ,       ___       ,        _       ]
)
```

We also introduce the new widgets `HS` (horizontal slider) and `VS` (vertical slider). The rules for expansion are:

- a widget can be continued horizontally to the right with ___ (the HS widget shown above)
- a widget can be continued vertically below with **III** (the VS widget shown above)
- the two continuations can be combined as shown for 'Big label' to obtain a big rectangular widget (here 'Big label' is a 3x3 widget). The widget must be in the top-left corner in the layout, while in the GUI it will appear centered.

The additional labels have been inserted to expand the layout. Without them, QT would have compressed the empty rows and columns to nothing.

6.2.10 Matplotlib

Matplotlib provides a QT-compatible widget. Guietta wraps it into its M() widget:

```
from guietta import Gui, M, ___, III, VS

gui = Gui(
```

(continues on next page)

(continued from previous page)

```
  [  M('plot') , ___ ,  ___ , VS('slider') ],
  [     III    , III ,  III ,      III      ],
  [     III    , III ,  III ,      III      ],
  [     III    , III ,  III ,  '^^^ Move the slider'  ],
 )
```

Here we define a big M widget, giving it the name *plot*. If a static plot was wanted, we could now directly draw into it. But since we like flashy things, we will make a plot that updates based on the slider position.

The M() widget has a magic property to refresh the plot: if you assign it a numpy array, it will just replot itself. If the array is 2d, it will use imshow(). Here is a simple callback to redraw the plot:

```
import numpy as np

def replot(gui, value):
    t = np.linspace(0, 1+value/10, 500)
    gui.plot = np.tan(t)
```

The callback, as usual, has the gui as its first argument. Since we intend to connect it to the slider, it also has a *value* argument, that will be the slider position. Guietta's sliders are basic QT sliders with a value that can go from 0 to 99 included.

If more control is needed, one could manually draw onto the widget:

```
import numpy as np

def replot(gui, value):

    ax = gui.plot.ax
    ax.clear()
    ax.set_title('y=tan(x)')
    t = np.linspace(0, 1+value/10, 500)
    ax.plot(t, np.tan(t), ".-")
    ax.figure.canvas.draw()
```

The callback can find the axis to draw on using "gui.<widgetname>.ax". It then proceeds to clear the axis and use normal Matplotlib commands. At the end, the canvas is redrawn.

Note: it is important to clear the axis before starting, otherwise the old plots will still be there and, in addition to confuse the drawing, things will slow down a lot very quickly because Matplotlib will be still redrawing all of them.

To simplify these requirements, Guietta provides a context manager that handles the clearing and redrawing. Thus the above callback can be simplified to this:

```
from guietta import Ax

def replot(gui, value):

    with Ax(gui.plot) as ax:
        ax.set_title('y=tan(x)')
        t = np.linspace(0, 1+value/10, 500)
        ax.plot(t, np.tan(t), ".-")
```

We now need to connect this callback to our slider:

```
gui.events(

    [  _            ,  _ , _ ,   replot     ],
    [  _            ,  _ , _ ,   _          ],
    [  _            ,  _ , _ ,   _          ], )
```

and run the GUI:

```
replot(gui, 1)
gui.run()
```

Notice how we first call the callback ourselves, giving it a default value, in order to have a plot ready when the GUI is displayed.

Multiple plots

The M() widget has an optional *subplots* keyword:

```
M('myplot', subplots=(2,3))
```

this will create a widget with 6 subplots organized in two rows. The *guietta.myplot.ax* member will be the return value from the subplots call, so it will be two lists with three ax objects each.

Note: if the subplots keyword is set to any value different from its default, the Ax context manager cannot be used.

6.2.11 Pyqtgraph

pyqtgraph is a plotting module with less features than matplotlib, but much faster. It is ideal if the graph must be updated frequently. Guietta wraps it into its *PG()* widget:

```
import numpy as np
from guietta import Gui, PG, ___, III, _, VS

gui = Gui(
  [  PG('plot'),  ___, ___, VS('slider') ],
  [     III     , III, III,     III      ],
  [     III     , III, III,     III      ],
  [     III     , III, III,  '^^^ Move the slider'  ],
 )
```

Replotting in pyqtgraph is much simpler than in matplotlib: we just have to call the widget's *plot()* method, setting *clear=True* to ensure that the previous plot is erased:

```
with gui.slider:
    t = np.linspace(0, 1+gui.slider/10, 500)
    gui.plot.plot(t, np.tan(t), clear=True)
```

Now we initialize the plot with a default one, and run the gui:

```
gui.slider = 1
gui.run()
```

If something more complex is needed, remember that pyqtgraph are full-featured QT widgets, so they can be instantiated and dropped into Guietta without the need to use the PG() wrapper.

Support for pyqtgraph was introducted in version 0.3.5.

6.2.12 Splash screens

Guietta supports extremely basic splash screens, with the *guietta.splash* function:

guietta.**splash**(*text*, *textalign=<PySide2.QtCore.Qt.Alignment object>*, *width=None*, *height=None*, *color=PySide2.QtCore.Qt.GlobalColor.lightGray*, *image=None*)

Display and return a splash screen.

This function displays a splashscreen and returns a QSplashScreen instance.

The splashscreen must be closed with close() or finish(gui.window()). Alternatively, it will close when the user clicks on it.

The splash function was introducted in version 0.3.1.

6.2.13 Background processing

Sometimes a handler function needs to run for a long time, and during that time the GUI would be frozen. In order to avoid this, the Gui class allows to span a function into a background thread. Once the function is done, an optional callback in the main thread will be triggered.

guietta.Gui.**execute_in_background**(*self*, *func*, *args=()*, *callback=None*)

Executes `func` in a background thread and updates GUI with a callback.

When func is done, the callback is called in the GUI thread. The callback receives a reference to this Gui instance as the first argument, plus whatever was returned by `func` as additional arguments.

6.2.14 Using other QT classes

Guietta makes available several other useful QT classes (like *QFileDialog*) without the need to import directly from PySide2:

```
from guietta import QFileDialog
```

a complete list is available in the reference guide.

6.2.15 Hierarchical GUIs

Sometimes you want to insert a Gui inside another one, for example in a program that builds its interface dynamically from basic GUI building blocks. Guietta supports it as a special case of its magic properties:

```
gui.label = another_gui
```

if `gui` and `another_gui` are two guietta.Gui instances, the *gui.label* widget (which could be any kind of widget) will be removed, and in its place the entire *another_gui* layout will appear, complete with all its widgets

The sub_layout example shows this trick in action.

Groupboxes

Sub-layouts are commonly associated with group boxes, that draw a rectangle with a title around a window section. Guietta supports the QGroupBox widget with its *G* widget, which can be assigned directly another Gui instance:

```
gui = Gui( [ G('my group') ] )

gui.mygroup = another_gui
```

See also the groupbox example

6.2.16 Packaging your application

Guietta runs perfectly fine if you have a Python interpreter, but sometimes you want to package a self-contained program which is able to run without dependencies. Guietta does not provide anything specific, but there are several programs that can do this for generic Python applications. One such program is PyInstaller.

Please note that PyInstaller is an independent project and we cannot give any guarantee that it will work, nor provide any support.

Next topic: the reference guide.

6.3 Reference

6.3.1 Layout

Guietta uses a grid layout (*QGridLayout*). Number of rows and columns is automatically calculated from the input. There is typically one widget per grid cell (_ will result in an empty cell), but widgets may span multiple rows and/or columns as described below.

Syntax

To create a layout, instantiate a *guietta.Gui* object and pass it a series of lists. Each list corresponds to a row of widgets. All lists must have the same length.

As a special case, if a list contains a single widget, the widget will be expanded to fill the whole row. This is useful for titles and horizontal separators.

6.3.2 Widgets

Here is the complete widget set:

```
from guietta import Gui, B, E, L, HS, VS, HSeparator, VSeparator
from guietta import Yes, No, Ok, Cancel, Quit, _, ___, III

gui = Gui(

   [ '<center>A big GUI with all of Guietta''s widgets</center>'],
   [ HSeparator ],

  [ 'Label'    , 'imagelabel.jpeg' , L('another label')  , VS('slider1')],
  [  _         ,     ['button']    , B('another button') ,     III      ],
```

(continues on next page)

(continued from previous page)

```
  [ '__edit__' ,  E('an edit box') , _                 ,   VSeparator ],
  [   Quit     ,        Ok         , Cancel             ,     III      ],
  [    Yes     ,        No         , _                  ,     III      ],
  [  HS('slider2'),    ___         , ___                ,      _       ] )

gui.show()
```

Syntax	Equivalent Qt widget	Event name
_	nothing (empty layout cell)	none
'text' L('text')	QLabel('text')	'text'
'image.jpg' L('image.jpg')	QLabel with QPixmap('image.jpg')	'image'
['text'] B('text')	QPushButton('text')	'text'
['image.jpg'] B('image.jpg')	QPushButton(QIcon('image.jpg'), '')	'image'
'__name__'	QLineEdit(''), name set to 'name'	'name'
E('text')	QLineEdit('text')	'text'
C('text')	QCheckBox('text')	'text'
R('text')	QRadioButton('text')	'text'
P('name')	QProgressBar()	'name'
G('title')	QGroupBox('title')	'title'
HS('name')	QSlider(Qt::Horizontal)	'name'
VS('name')	QSlider(Qt::Horizontal)	'name'
HSeparator	Horizontal separator	
VSeparator	Vertical separator	
M('name')	Matplotlib FigureCanvas*	
PG('name')	pyqtgraph PlotWidget*	
widget	any valid QT widget	none
(widget, 'name')	any valid QT widget	'name'

- Matplotlib or pyqtraph will only be imported if the M() or PG() widgets are used. Matplotlib and pyqtgraph are not is not installed automatically together with guietta. If the M() widget is used, the user must install matplotlib manually, same for PG() and pyqtgraph.

Buttons support both images and texts at the same time:

Syntax	Equivalent Qt widget	Event name
['image.jpg', 'text'] B('image.jpg', 'text')	QPushButton() with image and text	'text'

6.3.3 Continuations

How to extend widgets over multiple rows and/or columns:

```
from guietta import Gui, HS, VS, _, ___, III

gui = Gui(

   [ 'Big label', ___ , ___ , 'xxx' , VS('s1') ],
   [    III     , III , III , 'xxx' ,   III    ],
   [    III     , III , III , 'xxx' ,   III    ],
   [     _      ,  _  ,  _  , 'xxx' ,   III    ],
   [  HS('s2')  , ___ , ___ ,  ___  ,    _     ])
```

Syntax	Meaning
_	nothing (empty layout cell)
___	(three underscores) Horizontal widget span
III	(three capital letters i) vertical widget span

Rules:

- all grid cells must contain either a widget, one of ___ or **III**, or _ if the cell is empty. Other values will cause a ValueError exception. Empty elements are not allowed by the Python list syntax and will cause a SyntaxError.
- ___ can only be used to the right of a widget to extend it
- **III** can only be used below a widget to extend it
- ___ and **III** can be combined to form big rectangular widgets, with the widget to be extended in the top-left corner.

6.3.4 Signals

Signals can be connected with gui.events() where each widget has:

Syntax	Meaning
_	no connection
slot	reference to Python callable, using the default widget signal (if pre-defined, otherwise ValueError)
('textEdited', slot)	tuple(signal name, Python callable)

Table of default signals:

Widget	Signal
QPushButton	clicked(bool)
QLineEdit	returnPressed()
QCheckBox	stateChanged(int)
QRadioButton	toggled())
QAbstractSlider (QSlider, QDial, QScrollBar) QProgressBar	valueChanged(int)
QListWidget	currentTextChanged
QComboBox	textActivated

Widgets not listed in this table must be connected using the tuple syntax.

6.3.5 Properties

Table of properties created for each widget type:

Widget	Read property type	Write property type
QLabel, QLineEdit QGroupBox	str	str
QAbstractButton (QPushButton, QCheckBox, QRadioButton)	widget instance	callable
QAbstractSlider (QSlider, QDial, QScrollBar) QProgressBar	int	int
QAbstractItemView (QListWidget)	list of str	list of str
QComboBox	dict{str: any}	dict{str: any}
Matplotlib widgets	widget instance	1d and 2d array-like
Everything else	widget instance	raises an exception

All write properties accept a *guietta.Gui* instance. For all widgets except QGroupBox, such a write will cause the widget to be replaced by the new Gui's main QWidget, while the original widget will be hidden using *hide()*. For QGroupBox, its *setLayout()* method will be called using the new Gui layout as the argument.

6.3.6 Exception catching in slots

When a slot is called, they will be enclosed in a "try - except Exception" block. What happens in the except clause depends on the "exceptions" keyword parameter of the GUI constructor, which accepts the following enums:

Enum	Exception handling
Exceptions.OFF	nothing, exception is re-raised
Exceptions.POPUP (*default*)	popup a QMessageBox.warning with the exception string
Exceptions.PRINT	exception string printed on stdout
Exceptions.SILENT	nothing, exception is "swallowed"

6.3.7 QT symbols in Guietta

List of QT symbols defined in guietta. These symbols can be imported like `from guietta import x` instead of importing from PySde2:

```
from PySide2.QtWidgets import QApplication, QLabel, QWidget, QAbstractSlider
from PySide2.QtWidgets import QPushButton, QRadioButton, QCheckBox, QFrame
from PySide2.QtWidgets import QLineEdit, QGridLayout, QSlider, QAbstractButton
from PySide2.QtWidgets import QMessageBox, QListWidget, QAbstractItemView
```

(continues on next page)

(continued from previous page)

```
from PySide2.QtWidgets import QPlainTextEdit, QHBoxLayout, QComboBox
from PySide2.QtWidgets import QSplashScreen, QFileDialog, QButtonGroup
from PySide2.QtWidgets import QProgressBar, QGroupBox
from PySide2.QtGui import QPixmap, QIcon, QFont
from PySide2.QtCore import Qt, QTimer, Signal, QEvent
```

6.3.8 Gui class reference

class guietta.**Gui**(**lists*, *images_dir='.'*, *create_properties=True*, *exceptions=<Exceptions.POPUP: 3>*, *persistence=1*, *title=''*, *font=None*)

Main GUI class.

The GUI is defined passing to the initializer a set of QT widgets organized in rows of equal length. All other methods that expect lists (like events() or names()) will expect a series of list with the same length.

Every widget will be added as an attribute to this instance, using the widget text as the attribute name (removing all special characters and only keeping letters, numbers and underscores.)

align_fake_properties()
Make sure that any and all widgets have a property

auto(*func*)
Auto-connection decorator.

Analyzes a function and auto-connects the function as a slot for all widgets that are accessed in the function itself.

close(*dummy=None*)
Closes the window

column_stretch(**lists*)
Defines the column stretches

Arguments are lists as in the initializer. Since typically all rows have the same stretch, it is allowed to define just one or only a few rows in this method.

Every element must be a number, that will be passed to the setColumnStretch() QT function, or _ if no particular stretch is desired.

enable_drag_and_drop(*from_*, *to*)
Enable drag and drop between the two widgets

events(**lists*)
Defines the GUI events.

Arguments are lists as in the initializer. It is allowed to define just one or only a few rows in this method, if for example the last rows do not contain widgets with associated events.

Every element is a tuple with:

('signal_name', slot)

where 'signal_name' is the name of the QT signal to be connected, and slot is any Python callable. Use _ for widgets that do not need to be connected to a slot.

If just the default signal is wanted, 'signal_name' can be omitted and just the callable slot is required (without using a tuple).

Bound methods are called without arguments. Functions and unbound methods will get a single argument with a reference to this Gui instance.

execute_in_background(*func*, *args=()*, *callback=None*)
Executes `func` in a background thread and updates GUI with a callback.

When func is done, the callback is called in the GUI thread. The callback receives a reference to this Gui instance as the first argument, plus whatever was returned by `func` as additional arguments.

fonts(**lists*)
Defines the fonts used for each GUI widget.

Arguments are lists as in the initializer. It is allowed to define just one or only a few rows in this method, if for example the last rows do not contain widgets whose fonts need not to be modified.

Every element is either a QFont instance, a string with a font family name (.e.g 'Helvetica'), or a tuple with the QFont constructor elements: family string, point size, weight, and italic, int that order. All except the family string are optional, point size and weight ar integers, and italic is a boolean True/False. All these specifications are valid:

- QFont('helvetica', pointSize=12)
- 'helvetica'
- ('helvetica', 12, 1, True)

Use _ for widgets that do not need their fonts to be changed.

get(*block=True*, *timeout=None*)
Runs the GUI in queue mode

In queue mode, no callbacks are used. Instead, the user should call gui.get() in a loop to get the events and process them. The QT event loop will stop in between calls to gui.get(), so event processing should be quick.

Every time an event happens, get() will return a tuple:

    name, event = gui.get()

where `name` is widget name that generated the event, and event is a `namedtuple` with members `signal` (the PyQT signal) and `args` which is a list of signal arguments, which may be empty for signals without arguments.

get() will return (None, None) after the gui is closed.

get_selections(*name*)
Returns the selected items in widget *name*. Raises TypeError if the widget does not support selection.

import_into(*obj*)
Add all widgets to `obj`.

Adds all this Gui's widget to `obj` as new attributes. Typically used in classes as an alternative from deriving from Gui. Duplicate attributes will raise an AttributeError.

layout()
Returns the GUI layout, containing all the widgets

rename(**lists*)
Overrides the default widget names.

Arguments are lists as in the initializer. It is allowed to define just one or only a few rows in this method, if for example the last rows do not contain widgets that must be renamed.

Every element is a string with the new name for the widget in that position. Use _ for widgets that do not need to be renamed.

row_stretch(**lists*)
Defines the row stretches

Arguments are lists as in the initializer. Since typically all rows have the same stretch, it is allowed to define just one or only a few rows in this method.

Every element in the lists must be a number, that will be passed to the setRowStretch() QT function, or _ if no particular stretch is desired.

run()
Display the Gui and start the event loop.

This call is blocking and will return when the window is closed. Any user interaction must be done with callbacks.

show()
Shows the GUI. This call is non-blocking

title(*title*)
Sets the window title

widgets
Read-only property with the widgets dictionary

window()
Returns the window containing the GUI (an instance of QWidget). If the window had not been built before, it will be now.

6.3.9 Module-level functions reference

guietta.**splash**(*text*, *textalign=<PySide2.QtCore.Qt.Alignment object>*, *width=None*, *height=None*, *color=PySide2.QtCore.Qt.GlobalColor.lightGray*, *image=None*)
Display and return a splash screen.

This function displays a splashscreen and returns a QSplashScreen instance.

The splashscreen must be closed with close() or finish(gui.window()). Alternatively, it will close when the user clicks on it.

guietta.**Ax**(*widget*)
Context manager to help drawing on Matplotlib widgets.

Takes care of clearing and redrawing the canvas before and after the inner code block is executed:

```
with Ax(gui.plot) as ax:
    ax.plot(...)
```

guietta.**M**(*name*, *width=5*, *height=3*, *dpi=100*, *subplots=(1, 1)*, ***kwargs*)
A Matplotlib Canvas widget

The **kwargs accepts additional keywords that will become function calls every time the Ax decorator is used. For example, adding the argument `set_ylabel='foo'`, will result in this function call: `ax.set_ylabel('foo')`

6.4 How Guietta works

For the constructor aruguments:

- **argument checks (*layer_check()*)**

    1. Check that all elements are iterables, raise ValueError if not.

2. Take the longest
3. Expand single-elements ones to the longest using ___
4. Check that all rows have the same length, raise ValueError if not.

- **Compact syntax is expanded (*convert_compacts()*)**
  1. 'xxx' is converted to L('xxx')
  2. '__xxx__' is converted to QLineEdit('xxx')
  3. ['xxx', 'yyy'] is converted to B('xxx', 'yyy'), with 'yyy' optional. Lists with 0 or >2 elements raise ValueError
  4. 2-tuples are recursed into in order to expand the first element if needed
- Labels and buttons are created (*create_deferred()*)
  - **Labels**
    1. L('xxx') becomes (QLabel('xxx'), 'xxx')
    2. L('xxx.png') becomes (QLabel(QPixmap('xxx.png')), 'xxx')
  - **Buttons**
    1. B('xxx') becomes (QPushButton('xxx'), 'xxx')
    2. B('xxx.png', 'yyy' becomes (QPushButton(QIcon('xxx.png')), 'yyy')
  - Automatic buttons (Quit, Yes, No.. etc) are created and connected
  - Separators are created
  - 2-tuples are rercursed into in order to expand the first element if needed.
- **Multiple names are collapsed (*collapse_names()*)**
  - Things like (((widget, 'name1'), 'name2'), 'name3') become (widget, 'name3'). Nesting is flattened for an arbitrary depth.
- **Type check (*check_widget()*). All resulting widgets must be one of two types:**
  1. A QWidget instance, or
  2. a 2-tuple (QWidget, 'name')

6.4.1 Instance properties

Widgets values can be get/set like a property:

```
gui.label = 'text'
value = gui.slider
```

These property-like attributes are created on the fly when the GUI is built. We cannot use real properties, because these are class attributes and they would be shared between instances. Instead, there is a dictionary *self._fake_properties* which contains a mapping from property name to a pair of get/set functions (a namedtuple is used for the pair in order to have nice methods names). These methods are set to the ones appropriate for the widget type at construction time.

The _fake_properties dict is used by __getattr__ and __setattr__ to emulate the property behaviour. Since these methods would be used to lookup _fake_properties itself, this mapping must be created in the __init__ method as its first instruction, and using self.__dict__ instead of direct attribute access.

6.5 Changelog

6.5.1 [0.3.8] - 2020-07-20

Changed

- Workaround for bugs for some version of the "inspect" module.

6.5.2 [0.3.7] - 2020-07-20

Changed

- Compatibility with old Matplotlib versions (<2.1)

6.5.3 [0.3.6] - 2020-07-20

Added

- support for group boxes
- hierarchical layouts ("child" Gui instances) using property assignments
- font() method and construction keyword argument
- matplotlib widgets: added magic properties, subplots, arbitrary calls upon redrawing.

Changed

- callback in background processing is now optional
- documentation on readthedocs is finally properly versioned

6.5.4 [0.3.5] - 2020-07-10

Added

- title() method and construction keyword argument
- pyqtgraph integration

Changed

- QComboBox default signal is now 'currentTextChanged' for better backward compatibility with older QT versions.
- Fallback to import from PyQt5 instead of PySide2 if the latter fails.

6.5.5 [0.3.4] - 2020-07-06

Added

- Pre-defined radio button groups
- Progress bar widget
- Added default signal 'valueChanged' for QDial and QScrollBar

Changed

- Fixed bug for images when using the full file path
- Fixed small bugs in the examples
- "with" context manager now can reference imports and functions defined outside it.

6.5.6 [0.3.3] - 2020-06-18

Changed

- Support for older PySide versions (v5.9+)
- Fixed bug in ValueSlider layout
- Internal refactor adding the new Rows class.

Added

- 'clicked' signal for Matplotlib widgets

6.5.7 [0.3.2] - 2020-05-26

Changed

- Fixing incompatibilites between GitHub's and PyPI's README format.

6.5.8 [0.3.1] - 2020-05-26

Added

- Support for ComboBoxes (using QComboBox)
- Splash Screen (using QSplashScreen)
- @auto decorator syntax
- "with" context manager syntax
- removed all widget-generating functions, all widgets are now classes
- widgets can be specified with just the class, a widget with a default name will be allocated.

Changed

- “dropped” signal for list boxes (QListBox) renamed to “drop”

6.5.9 [0.3.0] - 2020-05-18

Changed

- Using PySide2 bindings instead of PyQt5

Index