# Fiatlight: Brighten the Journey from Idea to Creation

## Contents



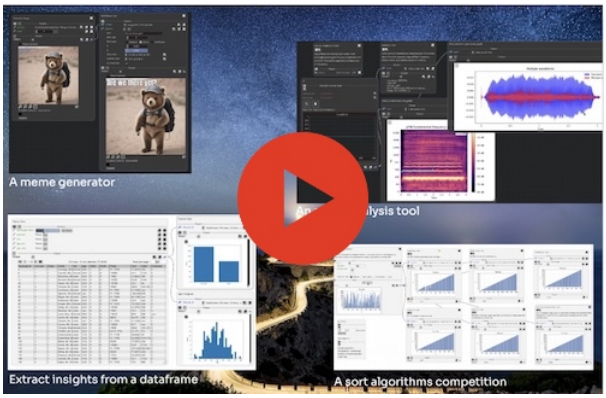
*Expressive Code, Instant Applications*

Fiatlight **bridges the gap between code and UI**, allowing you to turn ideas into **fully functional applications in minutes**. It **automates UI generation** for functions and structured data, making prototyping and fine-tuning faster and easier.

> *For technical readers:*
>
> FiatLight provides automatic UI generation for functions and structured data, making it a powerful tool for rapid prototyping and application development.
>
> - Instant Widgets: Edit and visualize any Python object with fine-grained control.
> - Function Pipelines: Chain functions into visual and interactive workflows.
> - Built-in Validation & Debugging: Enforce constraints, inspect data, and replay errors.
> - State Persistence: Save and restore application state seamlessly.

*The name “Fiatlight” is inspired by “Fiat Lux”, i.e. “Let there be light”.*

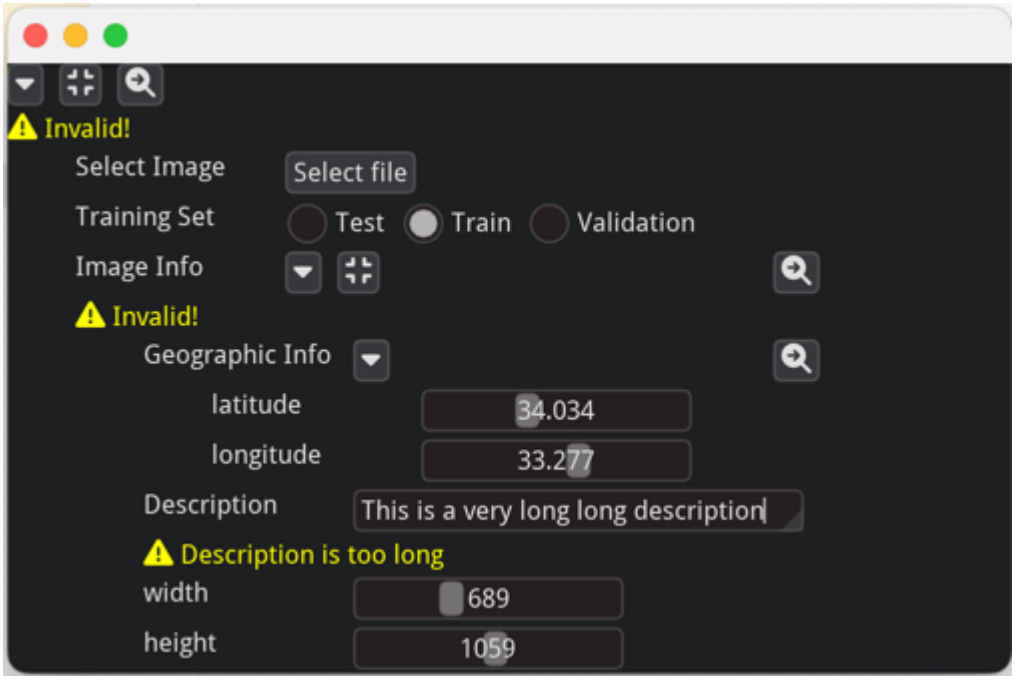
*Fiatlight is designed for rapid prototyping, experimentation, and fine-tuning applications. It does not provide full design control over GUI.*
*Notes: this page intends to provide a high-level overview of Fiatlight’s capabilities. For detailed tutorials, please refer to the video tutorials and the manual. Also, the demos presented in this page are also available in the video below*




# Create a GUI for structured data

In the example below, the GUI definition was created automatically, from the data structure definition of a nested pydantic BaseModel (including the validation rules, in yellow).

```
from fiatlight.demos.tutorials.pydantic_gui import demo_basemodel_app
# demo_basemodel_app.main()
```

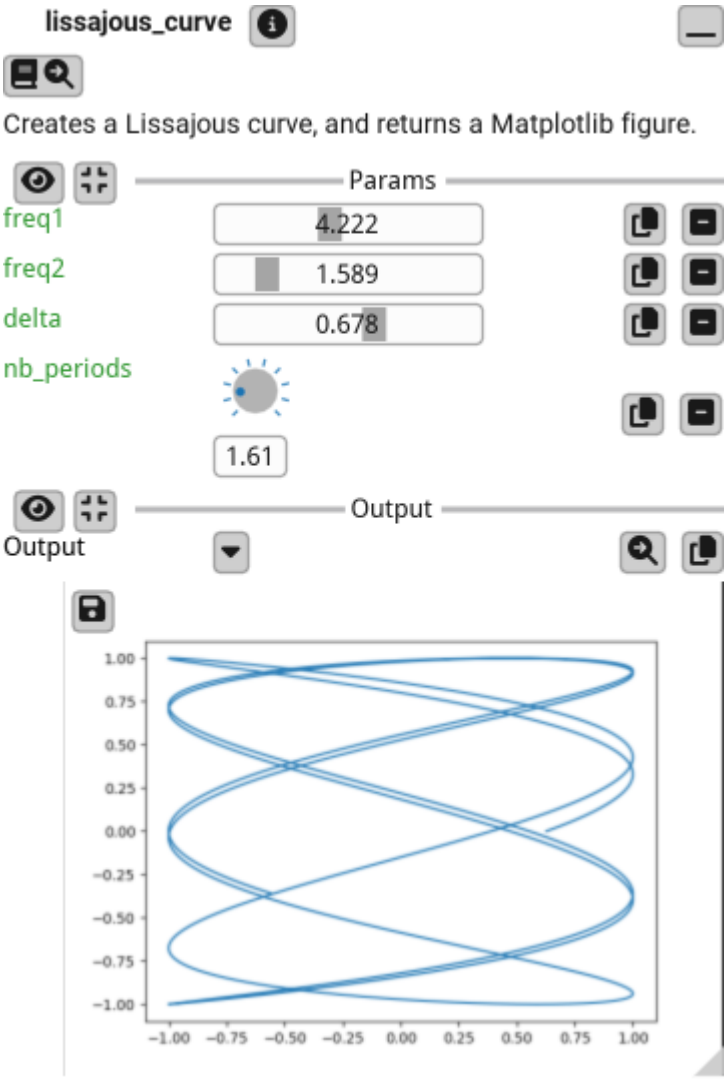



> *For technical readers: See the source code for demo_basemodel_app.py*. The GUI was created automatically, from a nested Pydantic model, with custom validator.

# Create a GUI for any function

Simply call `fl.run` with a function or a list of functions, and Fiatlight will automatically generate a GUI for them.

```python
# Part 1: Standard Python code (no user interface)
# ------------------------------------------------
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt


def lissajous_curve(freq1: float = 5.0, freq2: float = 4.0, delta: float = np.pi /
    """Creates a Lissajous curve, and returns a Matplotlib figure."""
    t = np.linspace(0, 2 * np.pi * nb_periods, 10_000)
    x = np.sin(freq1 * t + delta)
    y = np.sin(freq2 * t)
    fig, ax = plt.subplots()
    ax.plot(x, y)
    return fig


# Part 2: Add a GUI to the code in a few seconds
# ----------------------------------------------
import fiatlight as fl

# Options for widgets
fl.add_fiat_attributes(
    lissajous_curve,
    freq1__range=(0, 10), freq2__range=(0, 10), delta__range=(-np.pi, np.pi),
    nb_periods__range=(0.1, 10), nb_periods__edit_type="knob",
)
# Run the function interactively
fl.run(lissajous_curve, app_name="Interactive Lissajou Curve")
```



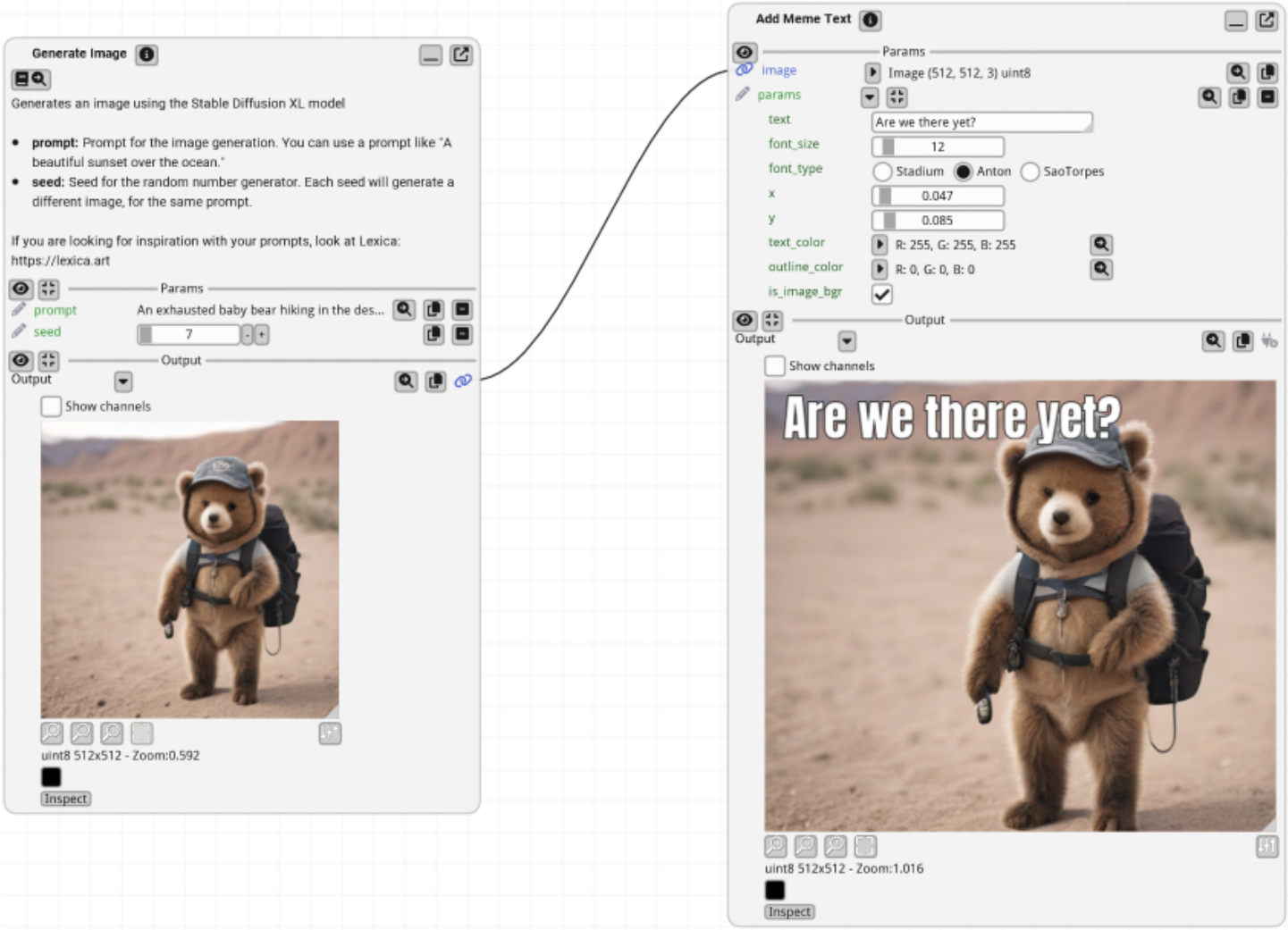

*See the application in action in the video below*

# From Idea to App in 3 minutes

Create a full application in just 4 lines of Python by chaining pure Python functions into an interactive graph. This graph visually displays each function's inputs and outputs, allowing for manual input adjustments.

*Example: The application below is a meme generator. It is a simple composition of an AI image generator, and a function that adds text onto an image*

```python
import fiatlight as fl
from fiatlight.fiat_kits.fiat_ai import invoke_sdxl_turbo
from fiatlight.fiat_kits.fiat_image.add_meme_text import add_meme_text

# Run the composition to create a simple app
fl.run([invoke_sdxl_turbo, add_meme_text], app_name="Old school meme generator")
```




This can be used as a full application:

- All inputs are saved: prompt, and meme text, font, color, position of the text
- All preferences are saved: window size, position, and layout of the nodes
- The user can save and load different state of the application (i.e. different memes)

> *For technical readers:* ***`invoke_sdxl_turbo`*** *provides a simple wrapper to SDXL, and* ***`add_meme_text`*** *is a Python function that adds colored text onto an image.*

# Domain-specific Kits:

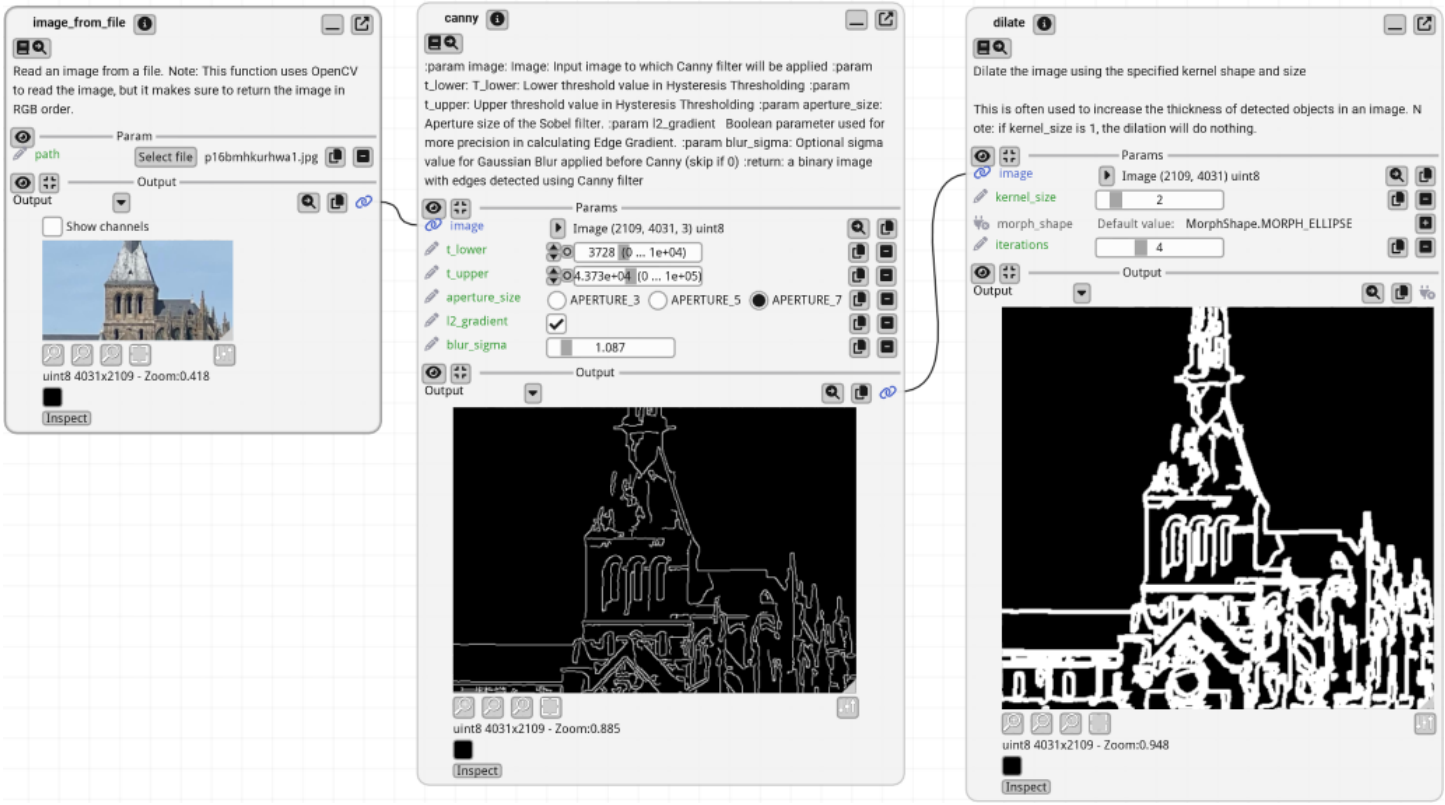
`fiatlight.fiats_kits` is intended to provide a set of pre-built functions and widgets for various domains, such as:

- **Image analysis**: Sophisticated image analysis and manipulation widgets. See fiat_image.
- **Data Visualization**: Display interactive data plots and charts for real-time data analysis, using MatPlotlib or ImPlot. See fiat_matplotlib, and fiat_implot (for ImPlot)
- **Data Exploration**: Provide widgets for exploring dataframes. See fiat_dataframe.
- **AI**: (Draft) Provide a widget for Prompt entry, and an interface to Stable Diffusion. See fiat_ai.

## Image analysis

*The example below shows an image which undergoes a pipeline for a dilated edge extraction. The image viewer can pan & zoom the images in sync, and display the pixel values*

```
import fiatlight as fl
from fiatlight.fiat_kits.fiat_image import image_from_file
from fiatlight.demos.images.demo_canny import canny, dilate

fl.run([image_from_file, canny, dilate], app_name="demo_computer_vision")
```

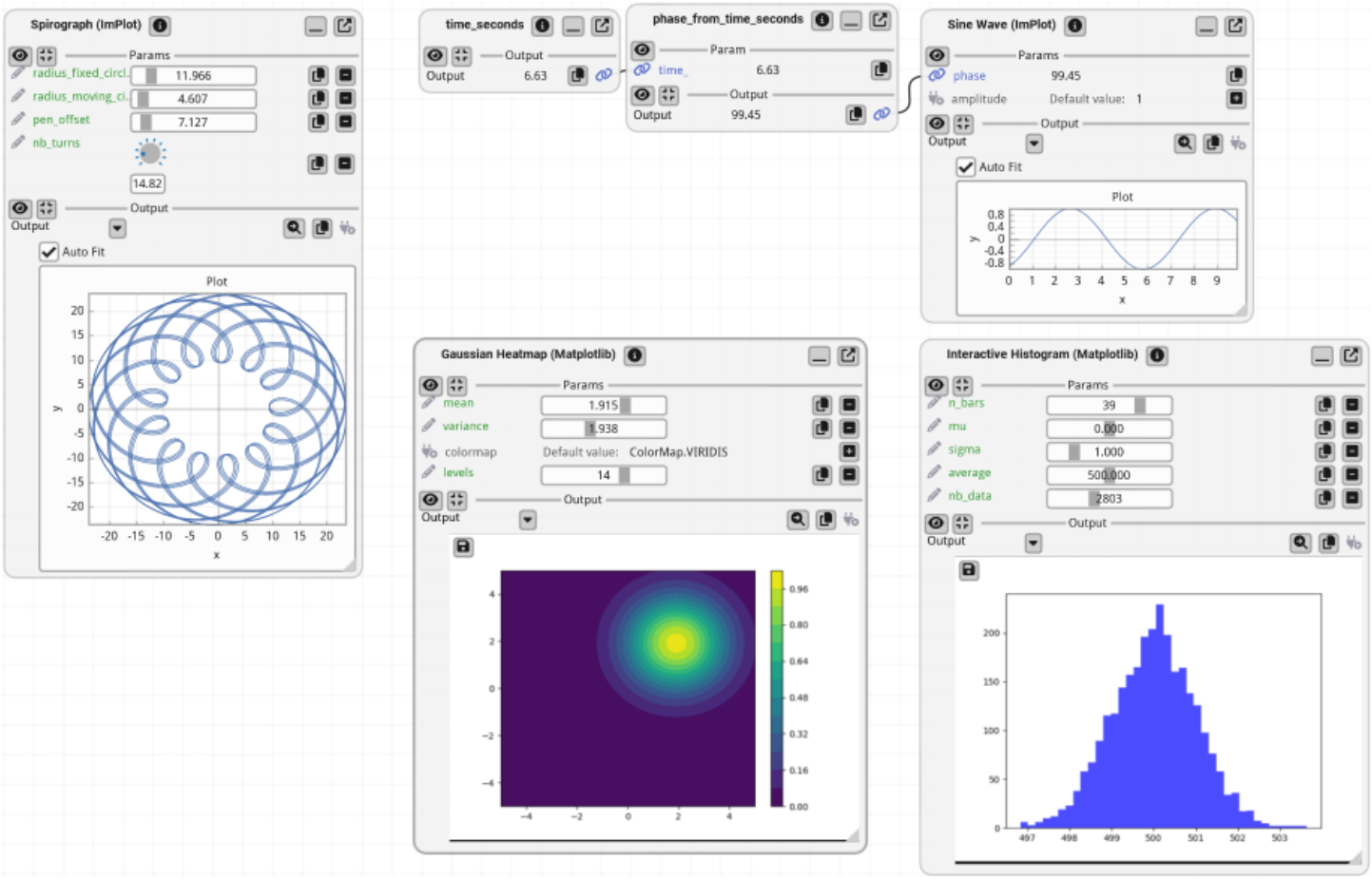



> *For technical readers:* `image_from_file` *is a function that reads an image from a file,* `canny` *applies the Canny edge detection algorithm, and* `dilate` *dilates the edges.*

## Data visualization with Matplotlib and ImPlot

*In the example below, we display figures using ImPlot (left) and Matplotlib (right). Each figure provides user-settable parameters (in a given range, with customizable widgets). The sine wave function is updated in real time.*

```
from fiatlight.demos.plots import demo_mix_implot_matplotib

demo_mix_implot_matplotib.main()
```

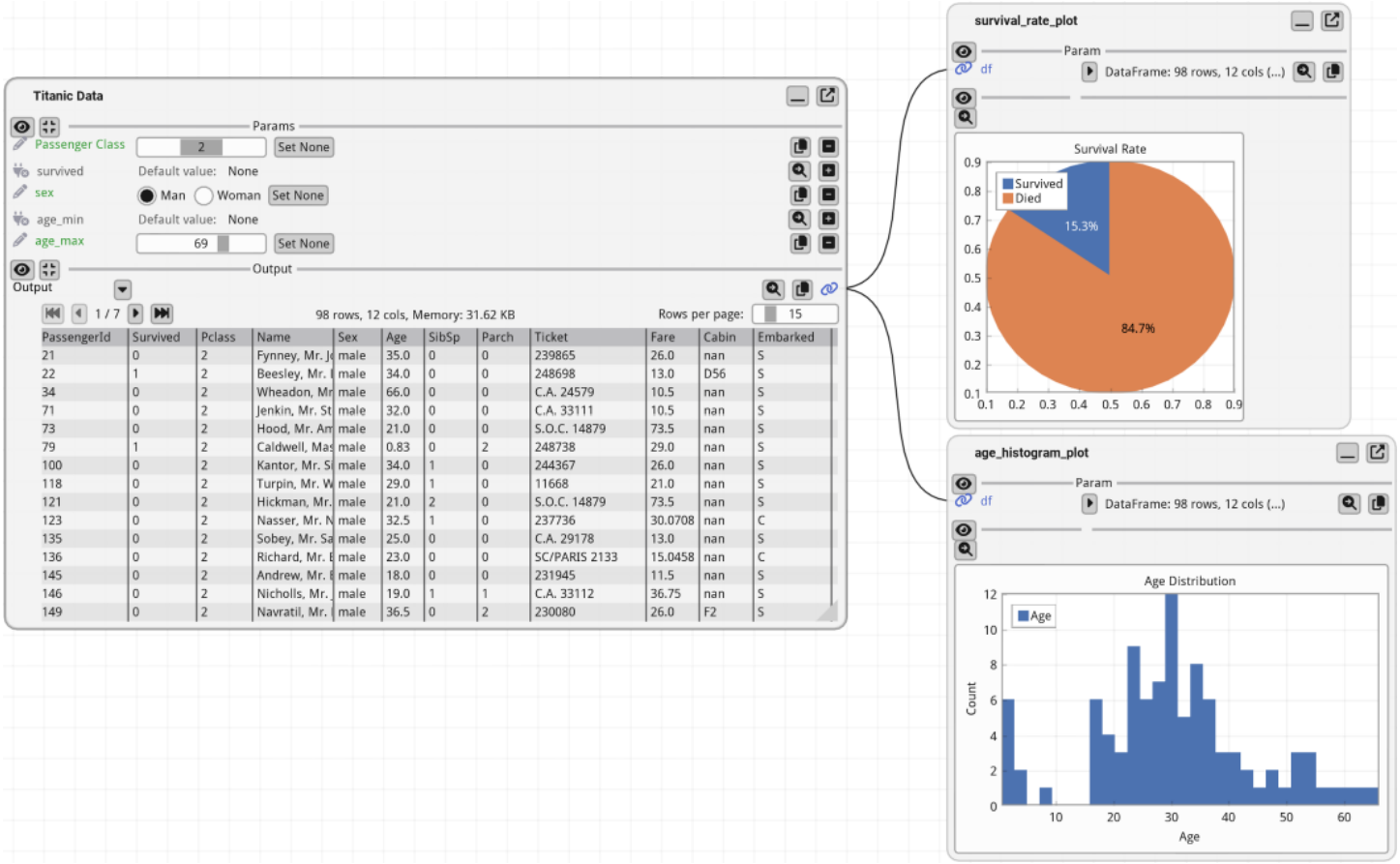



> *For technical readers:*
>
> - when a function returns a `matplotlib.figure.Figure`, its output will be displayed as a plot. See demo_matplotlib.py source code.*
> - when a function returns a `fiat_implot.FloatMatrix_Dim1` or `fiat_implot.FloatMatrix_Dim2` (which are aliases for np.ndarray), its output will be displayed as a plot, using ImPlot. See demo_implot source code.
> - ImPlot is a plotting library for Dear ImGui. It is often faster than Matplotlib, and can be used in real-time applications. For a complete demo of ImPlot, click here: ImPlot complete demo*

# Data Exploration

*In the example below, we display a data frame from the famous titanic example with filtering.*

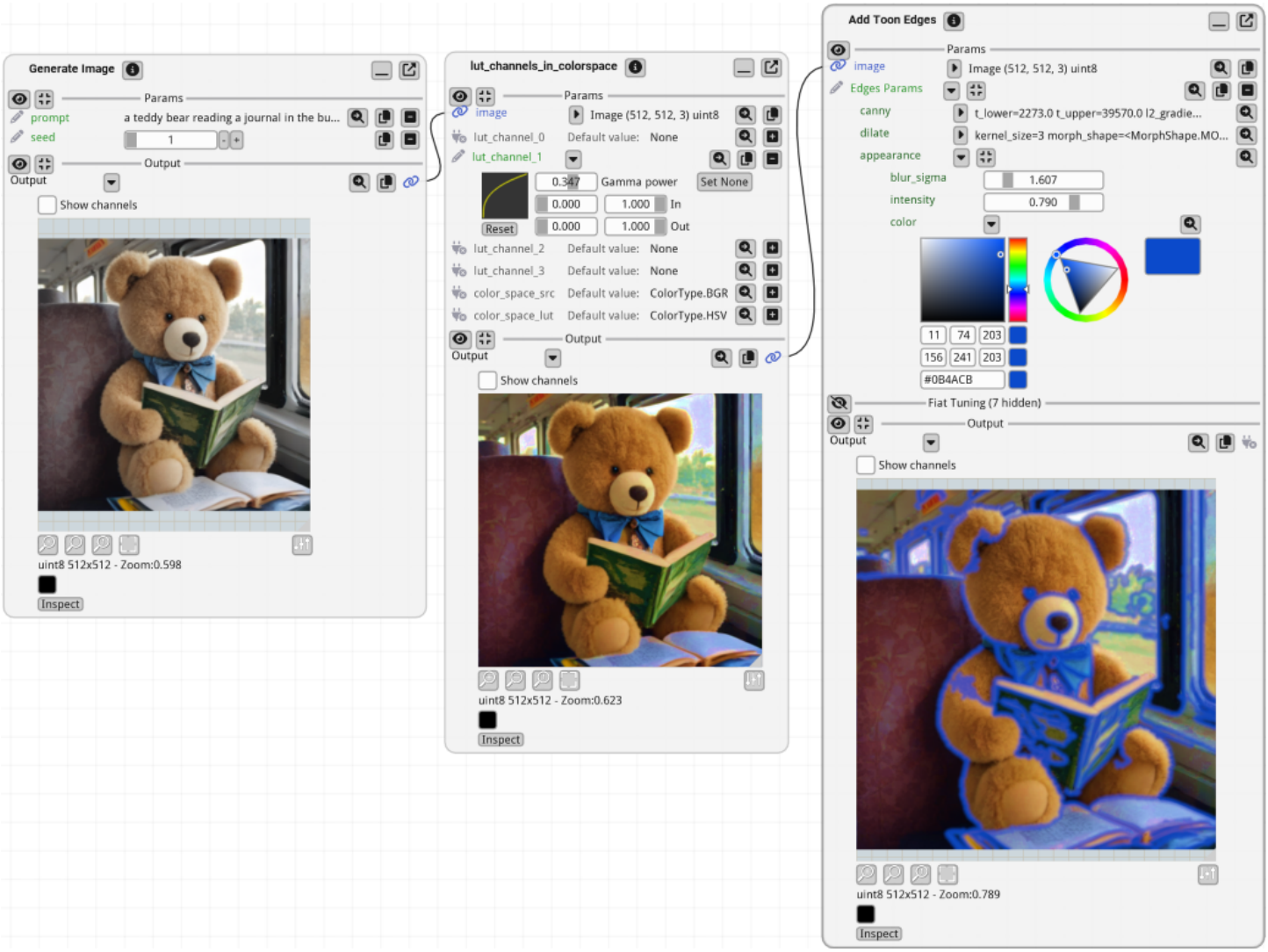
```
from fiatlight.fiat_kits.fiat_dataframe import dataframe_with_gui_demo_titanic
dataframe_with_gui_demo_titanic.main()
```




# AI - Image generation

*Example: the application below generates images using a stable diffusion model, and enables to add effects to it (color transformation, add colored edges).*

```
import fiatlight as fl
from fiatlight.fiat_kits.fiat_ai import invoke_sdxl_turbo
from fiatlight.fiat_kits.fiat_image import lut_channels_in_colorspace
from fiatlight.demos.images.toon_edges import add_toon_edges

fl.run([invoke_sdxl_turbo, lut_channels_in_colorspace, add_toon_edges], app_name="S
```




> *For technical readers:* `invoke_sdxl_turbo` *uses HuggingFace's diffuser library to invoke stable diffusion. See its source code*

# Visualize, Understand, Innovate

## Visualize the Pipeline flow

*Example: the application below looks for the most frequent words in a given text file (here with the text from "Hamlet"), by applying a pipeline of transformations. It is possible to inspect the input and outputs of each function.*

```
from fiatlight.demos.string import demo_word_count
demo_word_count.main()
```

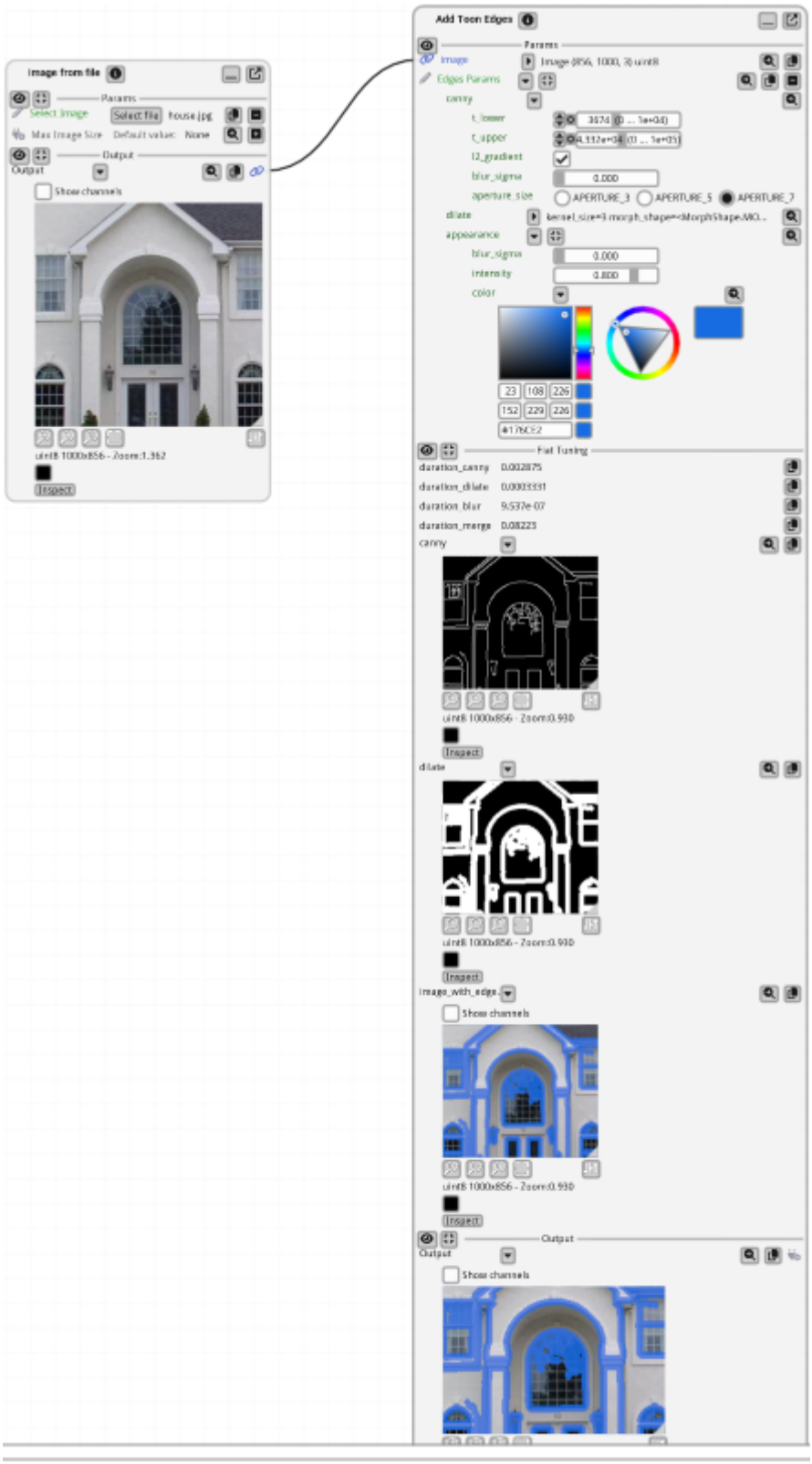

> *For technical readers: demo_word_count will simply chain the following string functions:* `text_from_file, str_lower, split_words, filter_out_short_words, sort_words, run_length_encode, sort_word_with_counts`. *See its source*

# Examine and understand function internals

fiatlight provides you with powerful tools to visually debug the intermediate states of your function.

*Example: the function add_toon_edges below is a complex function that adds a toon effect to an image. We can visualize the intermediate internal variables of the function (edges, dilated edges), even if they are not returned by the function.*

```
import fiatlight as fl
from fiatlight.fiat_kits.fiat_image import ImageU8_GRAY, ImageU8_3, image_source
from fiatlight.demos.images.toon_edges import add_toon_edges

fl.run([image_source, add_toon_edges], app_name="Toon Edges")
```

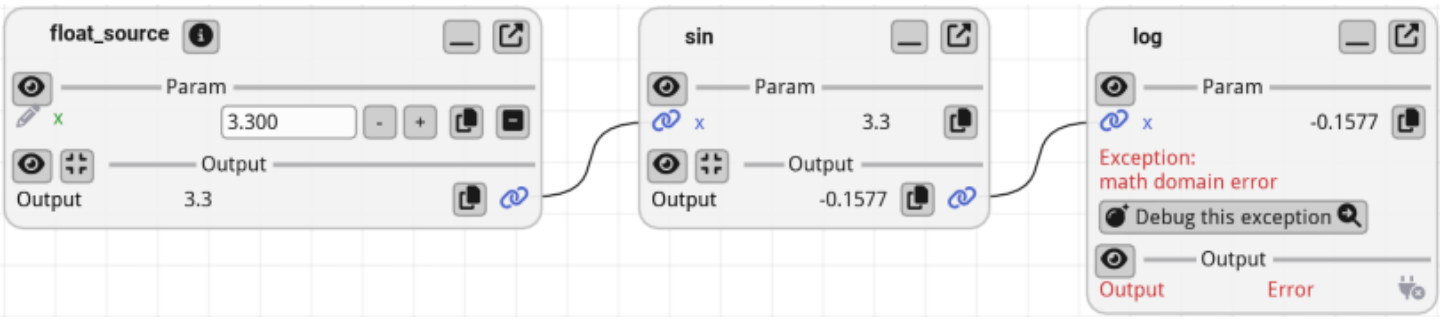



> *For technical readers: the function `add_toon_edges` has an attribute `fiat_tuning` that contains the internal variables that will be displayed. See demos/images/toon_edges.py.*

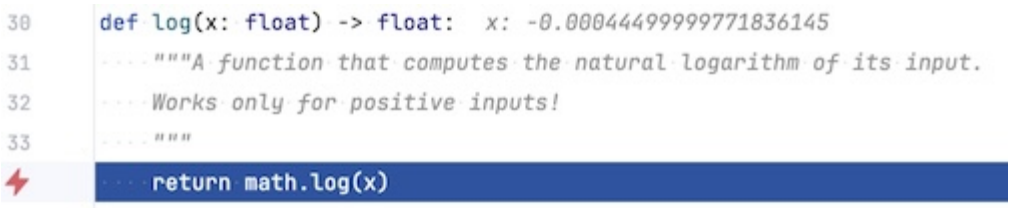
## Replay and debug function errors

*Example: the following application raises an error. However, this error can be replayed, with **the exact same inputs** to facilitate the debugging*

```python
import fiatlight as fl
import math

def float_source(x: float) -> float:
    """A source where the user can specify an input value."""
    return x
def sin(x: float) -> float:
    return math.sin(x)
def log(x: float) -> float:
    return math.log(x)

fl.run([float_source, sin, log], app_name="Replay error")
```




> *For technical readers: the function `log` will raise an error when x is negative. Once you click on the "Debug this exception" button, you will be able to debug it:*




# Full-fledged Applications

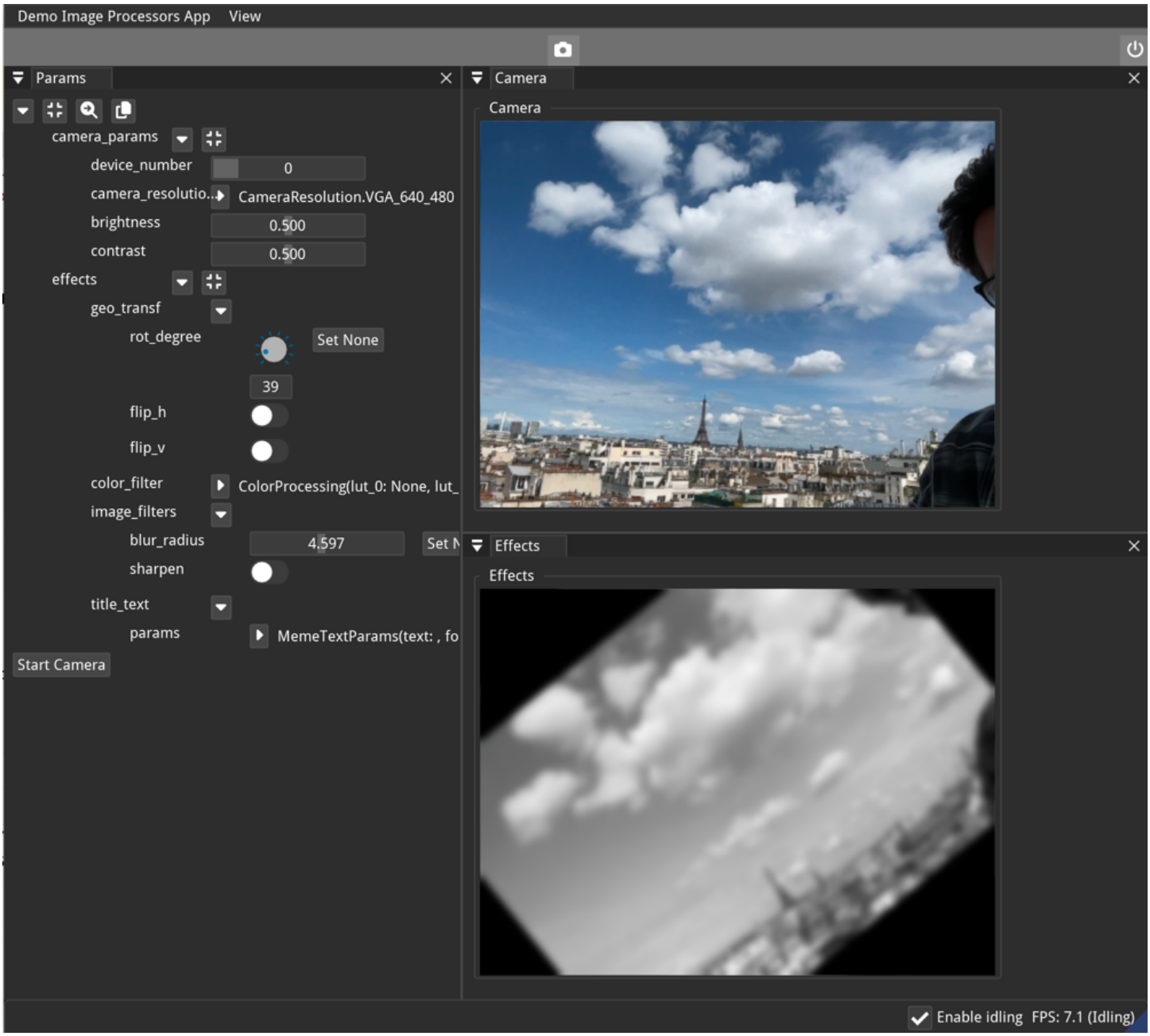
Besides being extremely powerful to generate function graphs, Fiatlight's powerful GUI capabilities can also help you generate sophisticated classic applications.

# Applications with advanced GUI

The example below shows an application which:

- reuses the sophisticated GUI provided by Fiatlight in a standard application
- automatically, Save and reloads its state, and GUI presentation options
- provides dockable windows, and a top toolbar

```
from fiatlight.demos.full_fledged_app import demo_image_processors_app
# demo_image_processors_app.main()
```



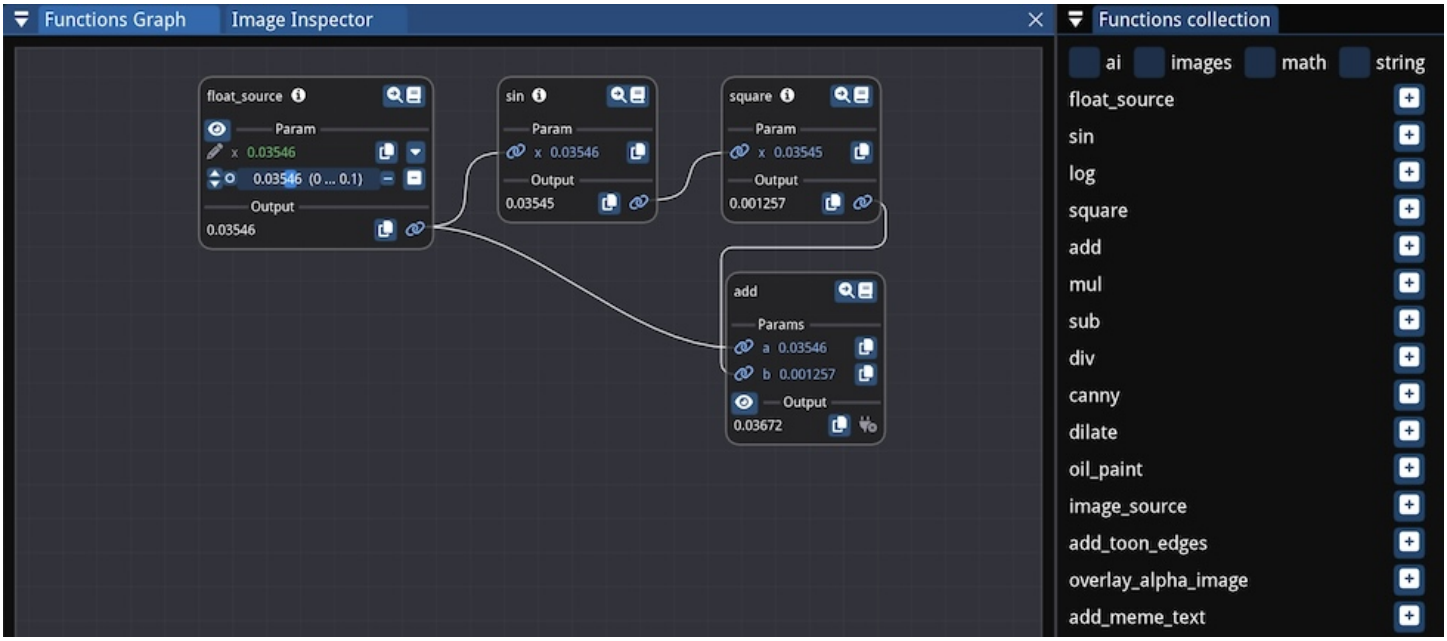

> - For technical readers: See the source code for demo_image_processors_app.py*.

# Custom Graph Creation

Create custom graphs with a drag-and-drop interface, similar to Scratch, enabling a visual approach to building workflows.

*Example: in the image below, its is possible to add and link function nodes:*

```
from fiatlight.demos.custom_graph import demo_custom_graph
# demo_custom_graph.main()
```



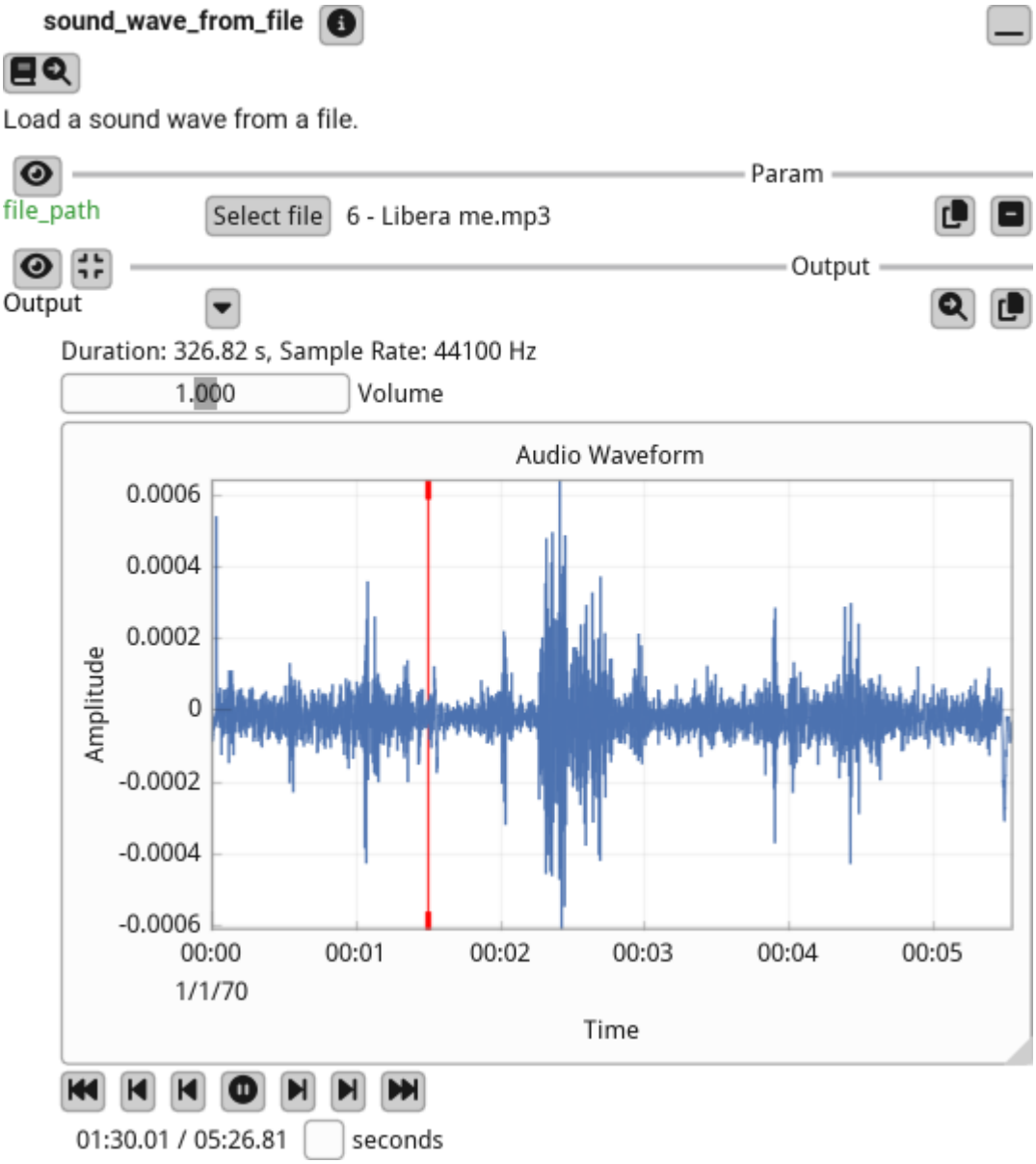

> *For technical readers: See the source code for custom_graph.py*

# Custom Widgets

Define custom ranges for data types, create custom widgets, and leverage special function attributes like async, live, and ranges for enhanced functionality and performance.

*Example: display and play a sound wave with a custom widget*

```
import fiatlight as fl
from fiatlight.fiat_kits.experimental.fiat_audio_simple import sound_wave_from_file
fl.run(sound_wave_from_file, app_name="Sound Wave Player")
```




> *For technical readers:* `sound_wave_from_file` *is a function that returns a sound wave from a file, and the widget is a custom widget that displays the sound wave and allows you to play it. See its source code*

---

Fiatlight is **best suited for**:

- **Rapid Prototyping** – Quickly transform ideas into interactive applications with minimal effort.
- **Fine-Tuning & Debugging** – Inspect intermediate states, visualize function outputs, and replay errors.
- **Education** – Teach programming, data science, and algorithm design with interactive tools.
- **Data Exploration** – Analyze, filter, and visualize complex datasets in real-time.

- **AI & Machine Learning** – Prototype AI models, fine-tune hyperparameters, and visualize results dynamically.
- **Application Development** – Prototypes built with Fiatlight can be **seamlessly transitioned into full applications** using **Dear ImGui**. Since Dear ImGui's API is nearly identical in Python and C++, porting to C++ is straightforward.

How does Fiatlight compare to other tools? See the full comparison.

# Installation

## Installation from source

```
%%bash

git clone https://github.com/pthom/fiatlight.git
cd fiatlight

# Optional: create a virtual environment
# (you can use whichever method you prefer)
python3 -m venv venv
source venv/bin/activate

pip install -r requirements.txt
pip install -v -e .
```

## Install imgui-bundle (from the main branch)

Fiatlight relies on imgui-bundle, and will depend on the latest version on the main branch (version 1.5.2 from pypi is not sufficient).

To install it, you can

- either clone it and install it from source:

```
git clone https://github.com/pthom/imgui_bundle.git
cd imgui_bundle
git submodule update --init --recursive # (1)
pip install -v . # (2)
pip install opencv-python
pip install pyGLM
```

- or download pre-compiled recent wheels from here: pthom/imgui_bundle

# Installation from PyPI

```
# Not available yet
```

# Install optional dependencies

Several requirements files are provided, which you can install via `pip install -r requirements-<name>.txt`:

- requirements.txt: basic requirements
- requirements-ai.txt: requirements for AI demos
- requirements-audio.txt: requirements for audio demos
- requirements-dev.txt: requirements for development

Note: for AI demos, you will have to install torch manually, as its installation is dependent on your system configuration. See https://pytorch.org/get-started/locally/ (you will of course need a GPU to run the demos)

# Video Tutorials

# Tutorials list

## Advanced Tutorial

This tutorial walks through the creation of an interactive sorting algorithm visualizer using Fiatlight, in order to explain more advanced features.

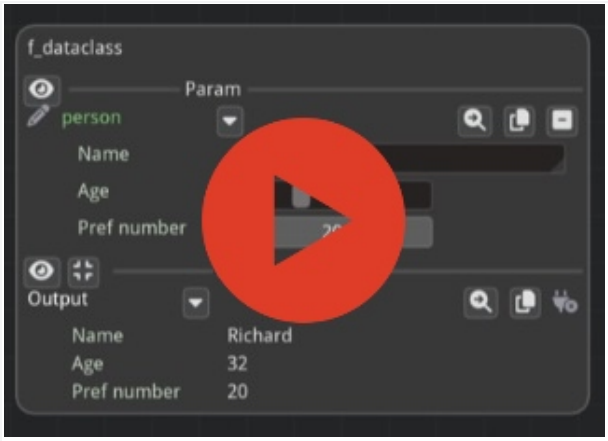
**Key Topics:**

- Create GUIs for Pydantic models and functions
- Customize outputs with ImPlot charts
- Use Fiatlight's function graph to build complex workflows
- Register custom GUIs for types
- Run functions asynchronously with real-time updates
- Build function graphs with GUI and documentation nodes
- Use Fiatlight GUIs inside a standalone app

---

## GUI for Pydantic Models

Learn how Fiatlight can instantly generate GUIs from Python dataclasses and Pydantic models.

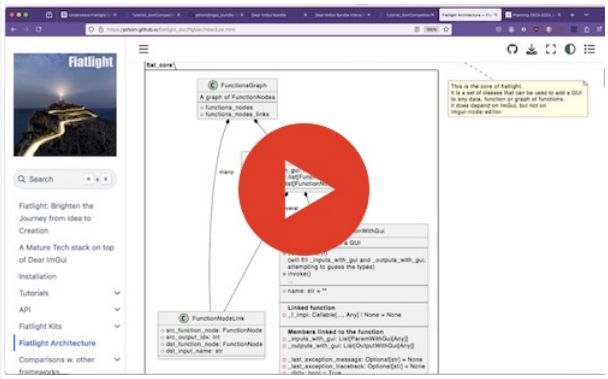



**Key Features:**

- Customizing widgets with `fiat_attributes`
- Automatic validation for user inputs
- Displaying structured data with interactive components

# Fiatlight Architecture

A high-level overview of Fiatlight's internal structure and how it automatically maps functions and data types to UI components.

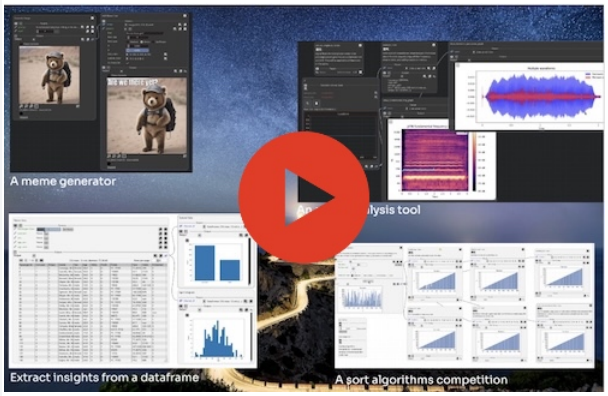



**Key Topics:**

- Core components: `AnyDataWithGui`, `FunctionWithGui`, `FunctionGraph`
- How Fiatlight registers data types to generate interactive UIs
- Customization callbacks to fine-tune how data is presented

# Full Demo of Fiatlight

**Overview:**
This video provides a complete walkthrough of Fiatlight, showcasing how it can rapidly generate UIs for various applications. It is a demo, rather than a tutorial.

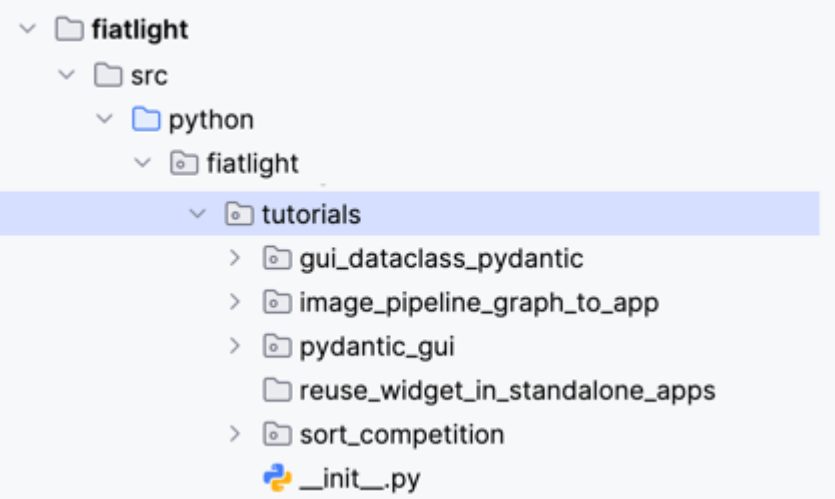



**Highlights:**

- AI-powered meme generator in just 4 lines of code
- Real-time sorting algorithm visualization
- Tailored kits for image processing, data visualization, and audio analysis
- Fine-tuning and debugging with function state introspection

# Sources for these videos

The sources for these tutorials are available in [src/python/fiatlight/demos/tutorials](src/python/fiatlight/demos/tutorials).



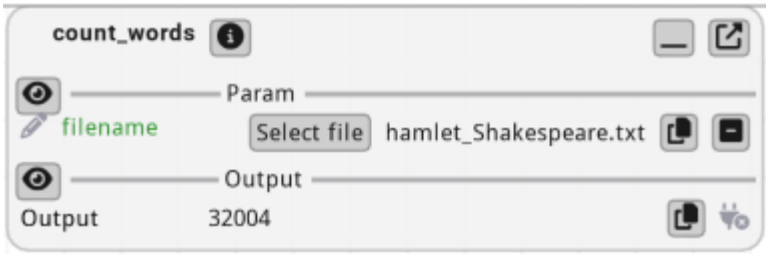

# First Steps

## Running functions via Fiatlight

### Running a single function

It is extremely simple to run and test a function with FiatLight. Below is a function that accepts a text path as a parameter and outputs the number of words in this text file.

```python
import fiatlight as fl
# Note: TextPath is a synonym for str
#       Within fiatlight, it is associated with a file dialog widget
from fiatlight.fiat_types import TextPath

def count_words(filename: TextPath) -> int:
    """Count the number of words in a text file."""
    with open(filename, "r") as f:
        text = f.read()
    return len(text.split())

# Run the application
fl.run(count_words, app_name="Count Words")
```




*Note: TextPath is an alias for str, but it is associated with a file dialog widget in Fiatlight.*

## Composing two functions

Below we create a simple application with two functions: “int_source” and “add”:

- “int_source” generates an integer value
- “add” adds two or three integer values.

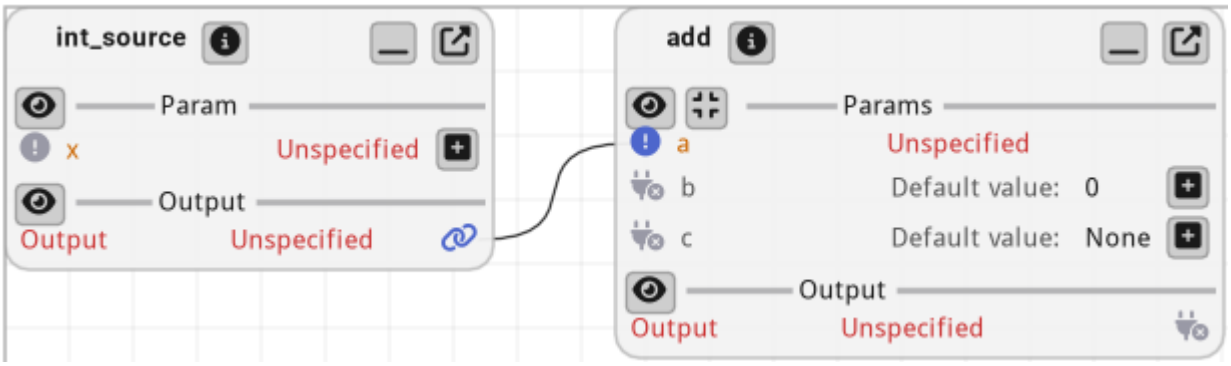
We specify the range of values for the input parameters of the functions using

- either the `fl.add_fiat_attributes` function,
- or the `@fl.with_fiat_attributes` decorator

Finally, we run the application using the “fl.run” function.

**Code**

```python
import fiatlight as fl


def int_source(x: int) -> int:
    """int_source is the first function of the application
    Since it is not linked to any other function, fiatlight will ask
    the user to specify the value of "x".
    As such, it serves as a source for the next function.
    """
    return x


# Customize the GUI for the `int_source` function. Below, we specify
# the range of values for "x" by adding "fiat_attributes"
fl.add_fiat_attributes(int_source, x__range=(0, 100))


# This second function adds the values of "a", "b", and "c"
# In this case, we add fiat_attributes using a decorator
# to specify the range of values for "a" and "b"
@fl.with_fiat_attributes(a__range=(0, 10), b__range=(0, 20))
def add(a: int, b: int = 0, c: int | None = None) -> int:
    """add is the second function of the application
    It adds the values of "a", "b", and "c" and returns the result.

    In the interface:
    - "a" is linked to the output of int_source and is unspecified
       until "x" is specified in int_source.
    - "b" is equal to its default value (0). It is shown in gray to
       indicate that it is using the default value.
    - "c" is an optional, equal to its default value (None). It is also shown in gr
      In order to specify a value for "c", the user must first click on the
      "Set" button, to specify that this optional has a value, and then specify the
    """
    if c is None:
        c = 0
    return a + b + c


# Run the application, which is a GUI around the composition of the two functions
# Notes:
#  - if running a single function, you can use fl.run(your_function)
#  - the app_name parameter is optional. It defines the name of the settings file,
fl.run([int_source, add], app_name="First Example")
```




*The image above shows the default state of the application*

> - int_source:
>   - “x” is unspecified
> - add:
>   - “a” is linked to the output of int_source and is unspecified, since int_source can not be executed (until “x” is specified)
>   - “b” is equal to its default value (0). It is shown in gray to indicate that it is using the default value.
>   - “c” is equal to its default value (None). It is also shown in gray.

# Video Tutorial of the available controls

*The video below shows how to interact with the widgets in a function node*

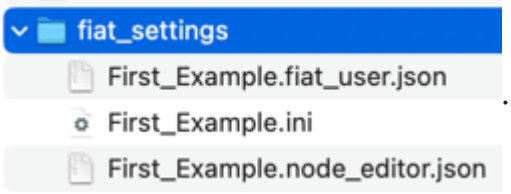
0:00

# Save / Load user settings

**Automatic user settings saving**

Upon exit, Fiatlight automatically saves the user's settings in a folder named `fiat_settings` in the current directory.

The settings are named after the app_name param passed to `fl.run (if app_name is not set, the settings file will use the name of the main application module)




Three files are saved each time the application saves the settings:

- First_Example.fiat_user.json: user settings (values of the parameters of the functions)
- First_Example.node_editor.json: settings for the node editor (positions of the nodes)
- First_Example.ini: settings for Dear ImGui (since and positions of the window)

**Manually save the user settings**

When you manually save the user inputs by clicking on the menu "File / Save user settings", the user settings are saved in a file named "xxx.fiat_user.json", where "xxx" is the file name you selected.

# Manual

Begin with the First steps section to learn how to wrap functions and dataclasses and run them via Fiatlight.

Then, explore the following sections.

## Introductory topics:

- Add types to signatures so that Fiatlight can generate a GUI for your functions
- Use fiatlight command line tool to list all supported types and their possible GUI customization options
- Customize widgets using fiat_attributes

- Fully customize any parameter's GUI by writing it by hand
- Add GUI only nodes to your functions graph (i.e. nodes that do not have a function associated with them)
- Run functions asynchronously
- Create GUI for structured data, i.e pydantic models and dataclasses

## Advanced topics:

- Validate inputs in the GUI
- Reuse Fiatlight widgets in your own apps, not only in Fiatlight's functions graphs.
- Fully customize functions GUI subclassing FunctionWithGui
- Fine-tune functions by viewing their internal status. Debug and replay exceptions.
- Create complex functions graph
- Create and register custom widgets for specific types

## Domain-specific topics:

Explore fiat_kits, collections of pre-built widgets for specific domains, such as:

- fiat_image for image processing
- fiat_matplotlib for plotting with Matplotlib
- fiat_dataframe: a widget to display explore pandas dataframes

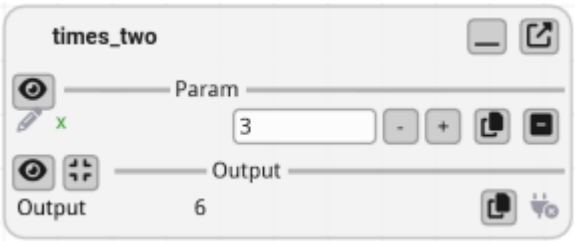
TODO: manual_reuse_widgets

## Wrapping Functions

In this tutorial, we will see how to wrap functions in order to make them compatible with Fiatlight.

Most of the time, functions are wrapped automatically. In the example below, the function `times_two` is wrapped automatically by Fiatlight into a `FunctionWithGui` object:

```python
import fiatlight as fl
def times_two(x: int) -> int:
    return x * 2
fl.run(times_two, app_name="Times two")  # the function will be wrapped automatical
```




> In order to be wrapped automatically, a function must have a typed signature (see Typed signatures).

`FunctionWithGui` is one of the core classes of FiatLight: it wraps a function with a GUI that presents its inputs and outputs.

- **Documentation**: See its API for detailed information.
- **Source code**: View its full code online.

# Typed Signatures

## Importance of Typed Signatures

To automatically create a GUI for function parameters, Fiatlight **requires** type information for both the parameters and the return value of the functions. This is achieved using type hints in the function signature.

For example, an untyped function signature looks like this:

```python
def foo(a, b):
    return a + b
```

In contrast, a typed version is:

```python
def foo(a: int, b: float) -> float:
    return a + b
```

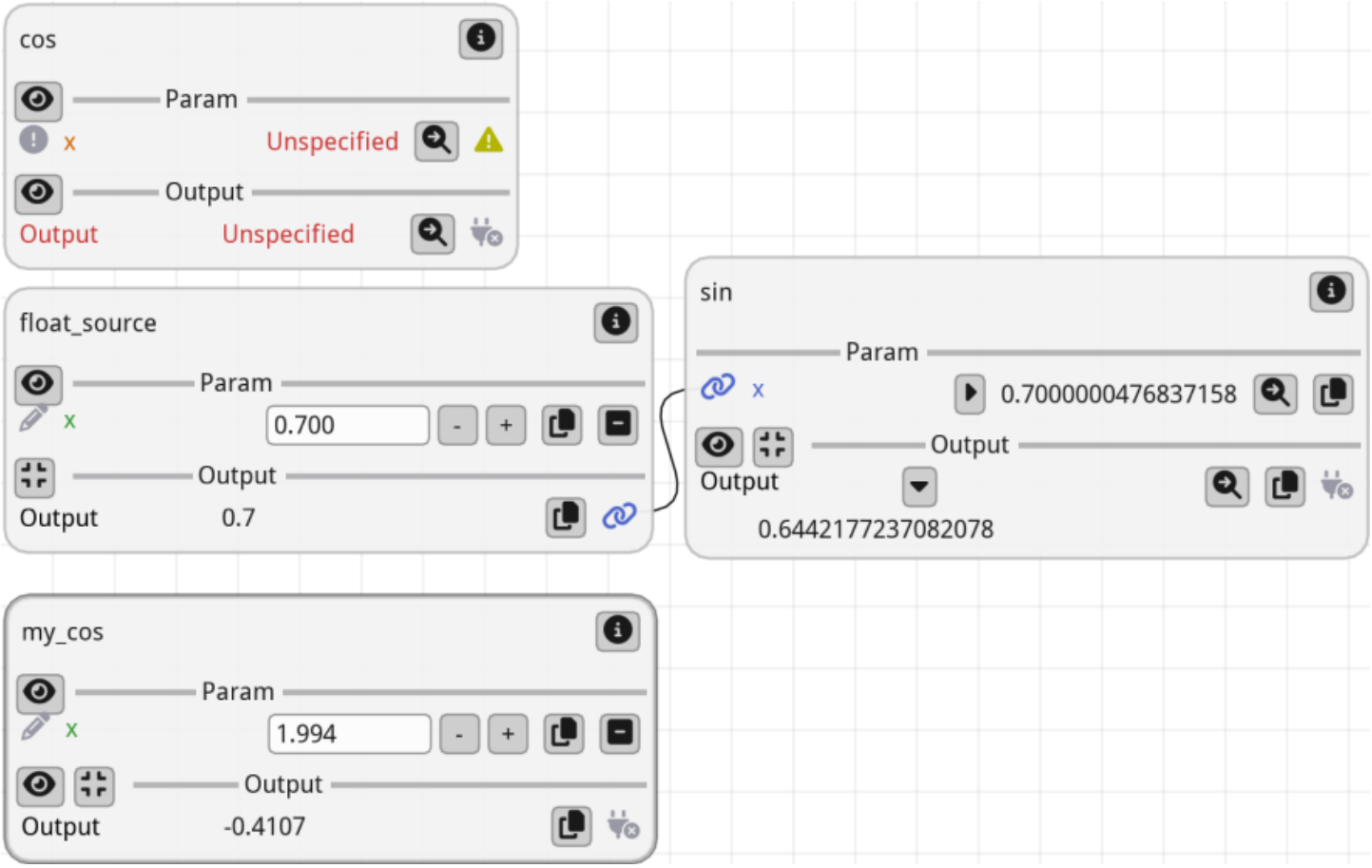
> More information about type hints can be found in PEP 484. *Type hints specify the type of a variable in Python. They are not mandatory but are a good practice, as they help catch bugs early in the development process.*

## Typed vs Untyped Functions GUI

In the example below, `math.sin` and `math.cos` are unfortunately not typed. `my_cos` is a wrapper around `math.cos` that includes type information.

**Code**

```python
import math
import fiatlight as fl


def float_source(x: float) -> float:
    """A float source, where the user can specify the value of x."""
    return x


def my_cos(x: float) -> float:
    """A wrapper around math.cos that adds types,
    so that Fiatlight can infer the widgets in the GUI."""
    return math.cos(x)


# We create a graph of functions, to which we will add functions manually
graph = fl.FunctionsGraph()

# Add a node that will run math.cos: since this function has no type information,
# Fiatlight **will not** be able to infer the widgets in the GUI)
graph.add_function(math.cos)

# Add a node that will run my_cos: since this function has type information,
# Fiatlight **will** be able to infer the widgets in the GUI
graph.add_function(my_cos)

# Add a function composition that will transfer the output of float_source to math.
# (in this case, math.sin will work correctly, since it only has to display the num
# and does not require an edition widget)
graph.add_function_composition([float_source, math.sin])

# Run the graph
fl.run(graph, app_name="Typed Signatures")
```




**Visual Output**: The image above shows the result of running the above code. Notice the differences in GUI behavior for typed vs untyped functions.

**Key Points**:

- Since `cos` is untyped, it is impossible to enter a value for its input parameter.
- The function `sin`, however, works correctly in the graph since it receives an input from `float_source` and does not require an edition widget.

## Wrapping Functions

Creating a wrapper is often extremely simple and necessary when dealing with untyped functions. Wrapping a function allows you to add type information, making it compatible with Fiatlight's GUI capabilities. Let's see how to wrap the `math.cos` function.

### Why Create a Wrapper?

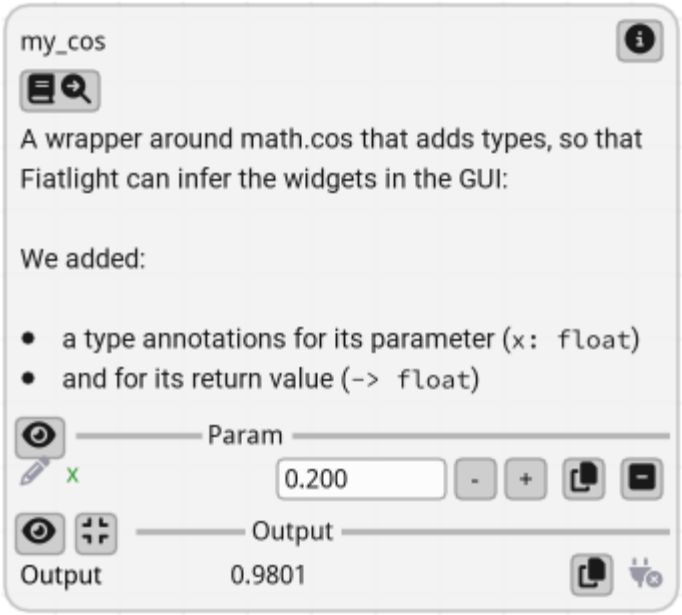
The `math.cos` function from Python's standard library does not have type annotations. Without these annotations, Fiatlight cannot automatically create a GUI for it. By creating a wrapper, we add

the necessary type information.

**Example: Wrapping `math.cos`**

Below, we create a simple wrapper for `math.cos` that includes type annotations. This allows Fiatlight to generate a GUI for the function.

```
import math
import fiatlight as fl

# Original function without type annotations
def my_cos(x: float) -> float:
  """A wrapper around math.cos that adds types, so that Fiatlight can infer the wid

  We added:
    * a type annotations for its parameter (`x: float`)
    * and for its return value (`-> float`)
  """
  return math.cos(x)

# Run the wrapped function with Fiatlight
fl.run(my_cos, app_name="Wrapped Cosine Function")
```




When running the above code, Fiatlight generates a GUI that allows you to input a float value for `x` and see the result of `math.cos(x)`.

*As an additional benefit, the documentation you wrote in the wrapper is visible in the function node!*

# Registered Types

## Introduction

Fiatlight maintains a central registry that links data types (e.g., primitive types or custom data types) with GUI types. This registry allows Fiatlight to automatically create GUIs for functions based on their type annotations.

For more information:

- **Gui Registry**: See the documentation for detailed information
- **GUI Types**: The GUI Types are all descendant of `AnyDataWithGui`, which is a generic type that can be used to create custom widgets for your data types. See its API.

## Using registered types

Registered types provide dedicated widgets, enabling automatic GUI creation for function parameters and outputs.

> Tip: use the command `fiatlight types` in a terminal (or console) to list the registered types and their associated widgets. See "Fiatlight command line utility" for more information.

Below is an extract of the output of the `fiatlight types` command:

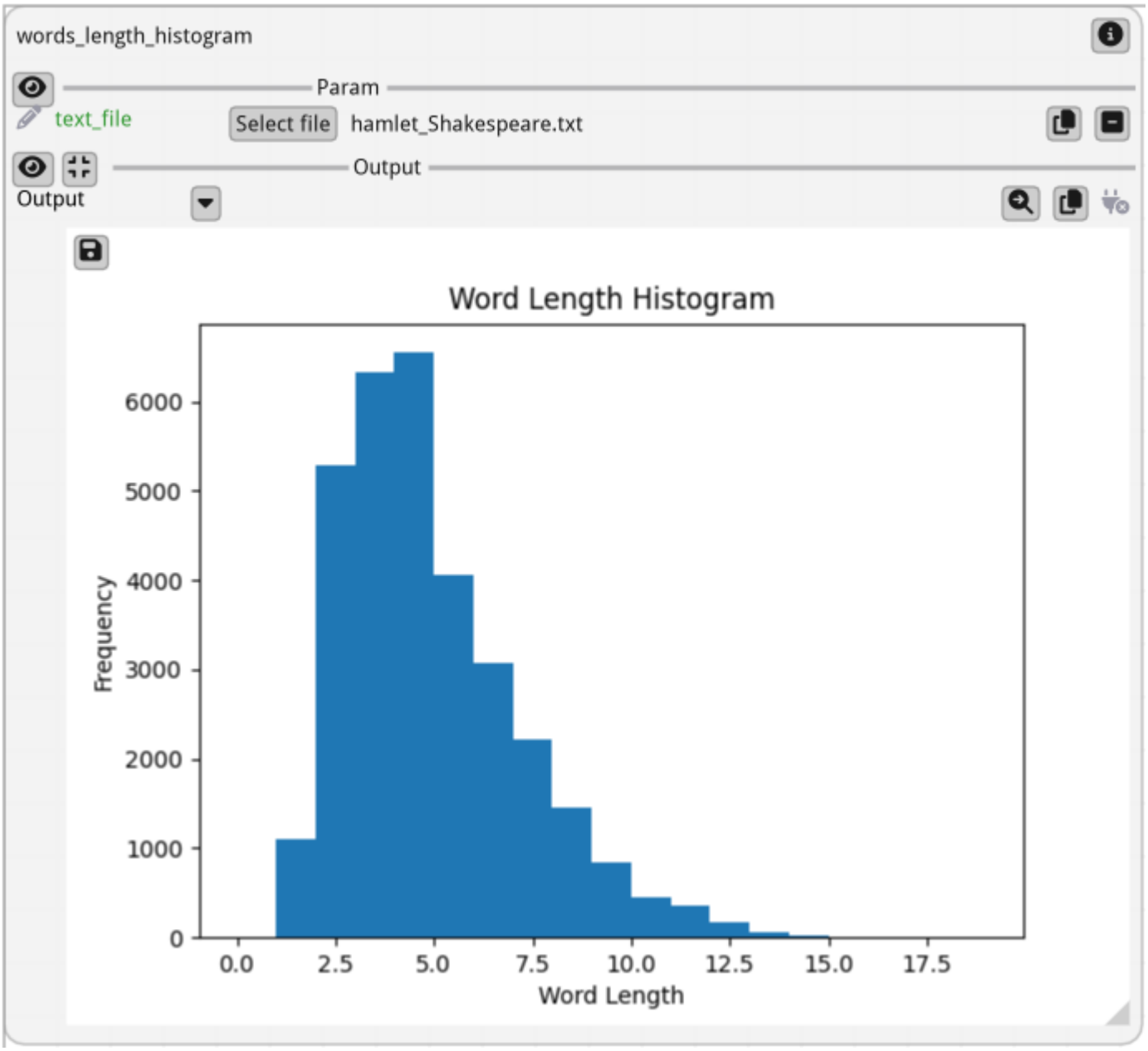
```
> fiatlight types
+-------------------------------------------------+-----------------------------
| Data Type                                       | Gui Type
+=================================================+=============================
| int                                             | IntWithGui
|                                                 |   A highly customizable int v
+-------------------------------------------------+-----------------------------
| float                                           | FloatWithGui
|                                                 |   A highly customizable floa
+-------------------------------------------------+-----------------------------
| str                                             | StrWithGui
|                                                 |   A Gui for a string with re
|                                                 | multiline editing.
+-------------------------------------------------+-----------------------------
| bool                                            | BoolWithGui
|                                                 |   A bool widget. Can use a cl
+-------------------------------------------------+-----------------------------
| ColorRgb                                        | ColorRgbWithGui
|   synonym for tuple[int, int, int] describing an|   A nice color picker for RGB
| RGB color, with values in [0, 255] (NewType)    |
+-------------------------------------------------+-----------------------------
| ...                                             | ...
```

## Example with Matplotlib Figures

Let's look at an example using `TextPath` and `matplotlib.figure.Figure`, which are registered types in Fiatlight:

- `fl.fiat_types.TextPath` is an alias for str, but it is registered to be displayed with a file selection dialog.
- `matplotlib.figure.Figure` is registered to be displayed as a plot in the GUI

```python
import fiatlight as fl
import matplotlib.figure
import matplotlib.pyplot as plt

def words_length_histogram(text_file: fl.fiat_types.TextPath) -> matplotlib.figure.
    "Create a histogram of the lengths of words in a text file."
    with open(text_file) as f:
        text = f.read()
    words = text.split()
    lengths = [len(word) for word in words]
    fig, ax = plt.subplots()
    ax.hist(lengths, bins=range(0, 20))
    ax.set_title("Word Length Histogram")
    ax.set_xlabel("Word Length")
    ax.set_ylabel("Frequency")
    return fig


fl.run(words_length_histogram, app_name="Registered types")
```




# Controlling Function Execution

By default, the function will be called only when one of its inputs has changed (either because the user entered a new value, or because an input is connected to another function that has changed).

You can control the behavior of the function by setting attributes on the function object.

- `invoke_async` (default=False): if set to True, the function will be called asynchronously
- `invoke_manually` (default=False): if set to True, the function will be called only if the user clicks on the “invoke” button

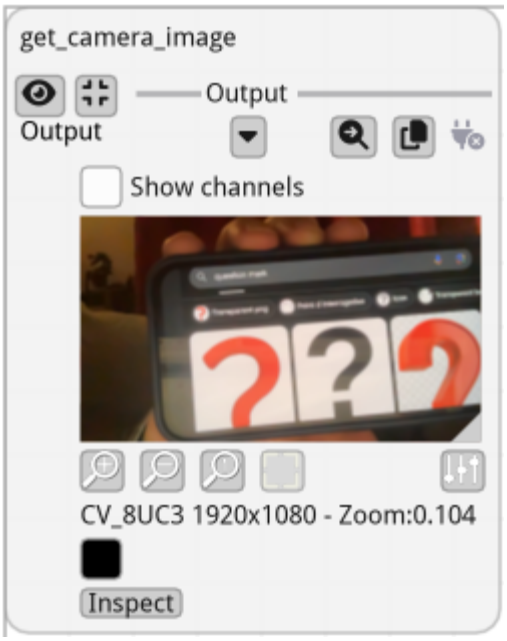
- `invoke_always_dirty` (default=False): if set to True, the function output will always be considered out of date. Depending on the value of `invoke_manually`:
  - if “invoke_manually” is True, the “Refresh needed” label will be displayed
  - if “invoke_manually” is False, the function will be called at each frame

> *Note: a “live” function is thus a function with `invoke_manually=False` and `invoke_always_dirty=True`*

## Configuring “Live” functions

**Example: a live function that display a camera image**

```
import fiatlight as fl
from fiatlight.fiat_kits.fiat_image import ImageU8_3
import cv2  # we use OpenCV to capture the camera image (pip install opencv-python)
cap = cv2.VideoCapture(0)  # you will need a camera!

def get_camera_image() -> ImageU8_3 | None:
    ret, frame = cap.read()
    return ImageU8_3(frame) if ret else None

# Set flags to make this a live function (called automatically at each frame)
fl.add_fiat_attributes(get_camera_image, invoke_always_dirty=True)

fl.run(get_camera_image, app_name="Live camera image")
```

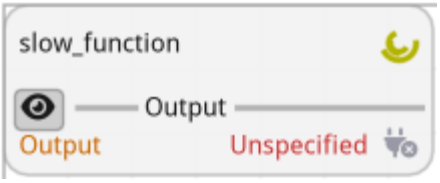


## Using Async Functions

**Example: an async function**

When your function is slow, you can set the `invoke_async` flag to True. In the example below, the yellow spinner indicates that the function is running, while keeping the GUI responsive.

```
import fiatlight as fl
import time
def slow_function() -> int:
    time.sleep(5)
    return 42

fl.add_fiat_attributes(slow_function, invoke_async=True)
fl.run(slow_function, app_name="Async function")

# Note:
# You can also use the `@fl.with_fiat_attributes` decorator to set the flags direct
# @fl.with_fiat_attributes(invoke_async=True)
# def slow_function() -> int:
#     ...
```




## “Stoppable” async Functions

In the case of async function, you may also set:

- `invoke_async_stoppable`(default=False): if true a GUI button will be displayed to stop the async function while it is running.

In this case, you will need to check the flag `invoke_async_shall_stop` in your function to know if the function should stop.

**Example:**

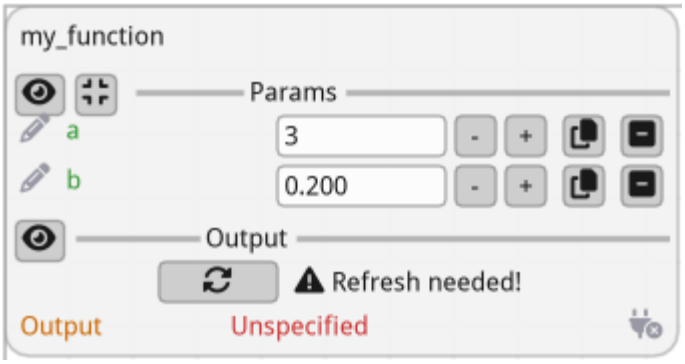
```python
def my_async_function():
    # ...  # some initialization
    while True:  # inner loop of the function processing (can be any form of loop)
        # ...  # some processing
        if hasattr(my_async_function, "invoke_async_shall_stop") and my_async_funct
            my_async_function.invoke_async_shall_stop = False  # reset the flag
            break
        # ... # continue the function processing
```

## Manual Invocation Example

**Example: a function that needs to be called manually**

If you set the `invoke_manually` flag to True, the function will be called only when the user clicks the “invoke” button (indicated by a “recycle” icon). If the inputs have changed, a “Refresh needed” label will be displayed.

```python
import fiatlight as fl
def my_function(a: int, b: float) -> float:
    return a + b

fl.add_fiat_attributes(my_function, invoke_manually=True)
fl.run(my_function, app_name="Manual invocation")
```




# Handwriting the GUI

## Using Edit and Present Callbacks

You can also customize the GUI for a parameter or output by setting custom callbacks function, namely the “edit” and “present” callbacks.

In this case, you will first wrap the function in a `FunctionWithGui` object, and then set the callbacks for the parameter or output.

```
import fiatlight as fl

def my_function(a: int) -> float:
    return a * 4

my_function_gui = fl.FunctionWithGui(my_function)

# A callback to edit the parameter. Receive the current value, and return a tuple (
# my_function_gui.input("a").callbacks.edit = ...

# A callback to present the output. Receive the current value, and return None
# my_function_gui.output().callbacks.present = ...
```

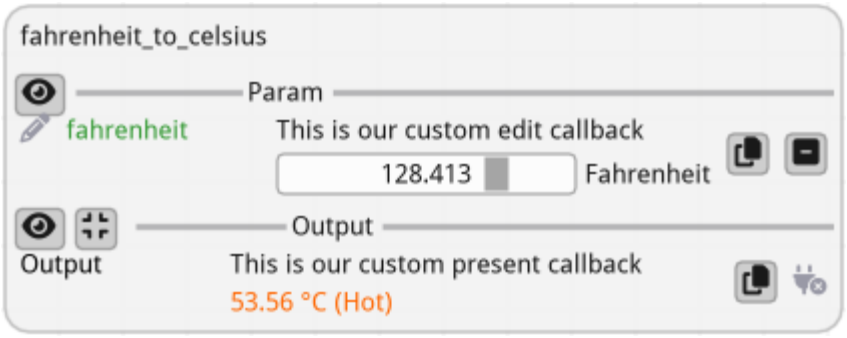
For more information, see:

- **AnyDataWithGui**: See the API for detailed information.
- **AnyDataWithGuiCallbacks**: See the API for detailed information.

## Example: Custom Callbacks

In this example, we define custom edit and present callbacks for the function fahrenheit_to_celsius. The resulting GUI allows the user to input a temperature in Fahrenheit using a custom slider and see the converted temperature in Celsius with a color-coded note indicating whether it is cold, warm, or hot.

```python
import fiatlight as fl
from imgui_bundle import imgui, hello_imgui, ImVec4


def fahrenheit_to_celsius(fahrenheit: float = 0) -> float:
    return (fahrenheit - 32) * 5 / 9

# This will be our edit callback: it accepts a float and returns a tuple (bool, flo
# where the first element is True if the value has changed, and the second element
def edit_temperature(fahrenheit: float) -> tuple[bool, float]:
    imgui.text("This is our custom edit callback")
    # Set the width of the slider field to 10 em units (using em units is a good pr
    imgui.set_next_item_width(hello_imgui.em_size(10))
    changed, new_value = imgui.slider_float("Fahrenheit", fahrenheit, -100, 200)
    return changed, new_value

# This will be our present callback: it accepts a float and returns None
def present_temperature(celsius: float) -> None:
    imgui.text("This is our custom present callback")
    note = "Cold" if celsius < 20 else "Hot" if celsius > 40 else "Warm"
    color = ImVec4(0, 0.4, 1, 1) if celsius < 20 else ImVec4(1, 0.4, 0, 1) if celsi
    imgui.text_colored(color, f"{celsius:.2f} °C ({note})")


fahrenheit_to_celsius_gui = fl.FunctionWithGui(fahrenheit_to_celsius)
fahrenheit_to_celsius_gui.output().callbacks.present = present_temperature
fahrenheit_to_celsius_gui.input("fahrenheit").callbacks.edit = edit_temperature

fl.run(fahrenheit_to_celsius_gui, app_name="Custom callbacks")
```




# Customizing Widgets with Fiat Attributes

## Introduction

Fiat attributes allow you to customize various aspects of the GUI for function nodes, dataclasses, and pydantic models. They provide a powerful way to modify the appearance and behavior of

function parameters and outputs, adjust the GUI for dataclasses, control how function nodes run (e.g., asynchronously or manually), set labels and tooltips for function nodes and parameters, and validate function parameters and dataclass fields.

There are two main ways to add attributes to a function:

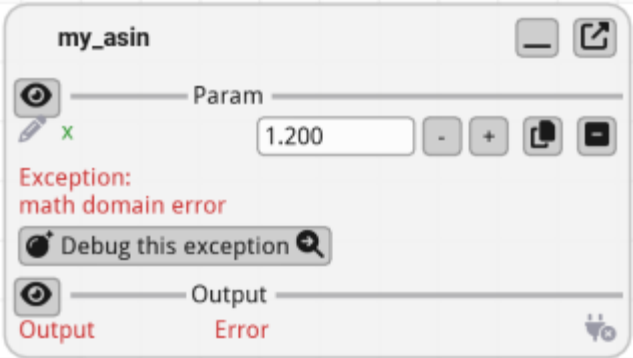
- using a decorator: `@fl.with_fiat_attributes` on top of the function definition
- using the `fl.add_fiat_attributes` function elsewhere in the code

> For more details on customizing dataclasses and pydantic models, see the GUI Registry documentation.

## Why Customize Widgets?

As an example, let's consider the function "my_asin" below: if you run this function with `run()`, the GUI will allow the user to enter any float value for x. This lets the user enter values that may not be valid for the function.

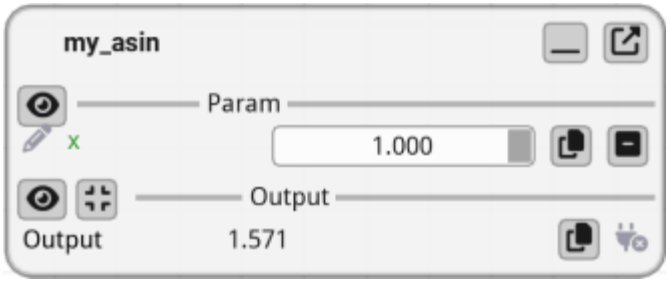
```
import fiatlight as fl

# Ideally, we would like to restrict the range of x to [-1, 1]
def my_asin(x: float = 0.5) -> float:
   import math
   return math.asin(x)

fl.run(my_asin, app_name="No range restriction")
```



## Adding Attributes with a Decorator

It is possible to customize the GUI for parameters using function attributes: below, we set the range for x. As a consequence it will be displayed with a slider widget with a range from -1 to 1.

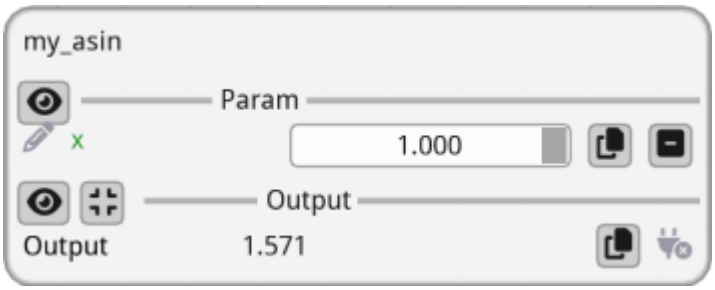
```
import fiatlight as fl

# Use the `with_fiat_attributes` decorator to set custom fiat attributes for the fu
# Here, we set the range of the x parameter.
# Important: note the double underscore ("_") after the parameter name!
@fl.with_fiat_attributes(x__range=(-1, 1))
def my_asin(x: float = 0.5) -> float:
    import math
    return math.asin(x)


fl.run(my_asin, app_name="Range restriction")
```




## Adding Attributes Manually

If you do not wish or cannot modify the function definition, you can use the `add_fiat_attributes` function to add attributes to the function. This way, your function stays unmodified, and you can specify the attributes only when creating its GUI.

```
import fiatlight as fl

def my_asin(x: float = 0.5) -> float:
    import math
    return math.asin(x)

# Important: note the double underscore ("_") after the parameter name!
fl.add_fiat_attributes(my_asin, x__range=(-1, 1))
fl.run(my_asin, app_name="Range restriction")
```




## Listing Available Fiat Attributes

To customize the GUI for function parameters or outputs, you can list the available fiat attributes for a specific type using the Fiatlight command line utility.

Use the command fiatlight gui int in a terminal (or console) to list available fiat attributes for the int type. For other types, replace int with the type you are interested in.

For more information, see "Fiatlight command line utility".

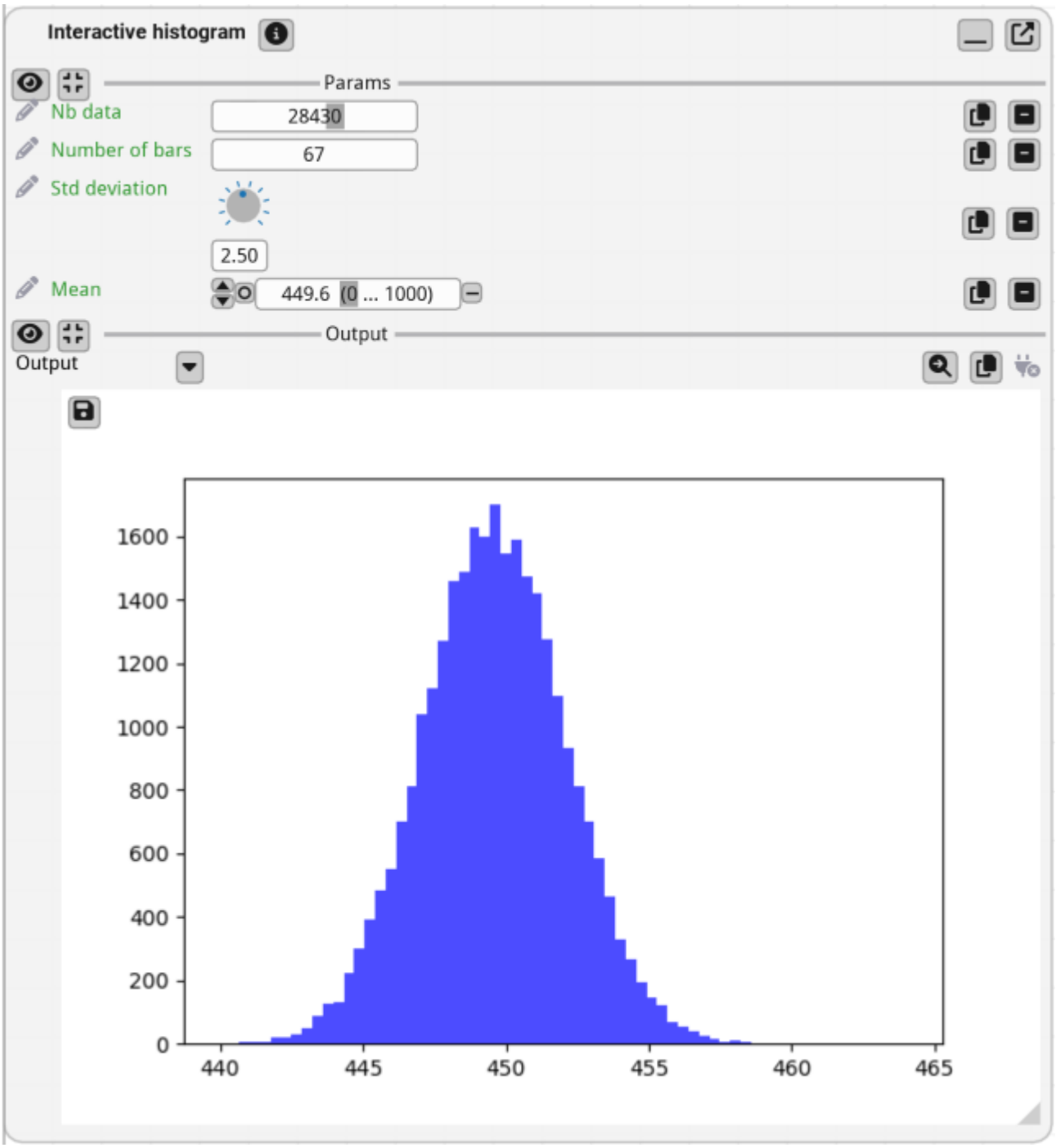
Below is an extract of the output of `fiatlight gui int`:

```
GUI type: int
==============
A highly customizable int widget.
Available fiat attributes for IntWithGui:
+--------------------+-----------------+-----------+-------------------------------
| Name               | Type            | Default   | Explanation
+====================+=================+===========+===============================
| range              | tuple[int, int] | (0, 10)   | Range of the integer
+--------------------+-----------------+-----------+-------------------------------
| edit_type          | str             | input     | Type of the edit widget. Possil
|                    |                 |           | slider, input, drag, knob,
|                    |                 |           | slider_and_minus_plus
+--------------------+-----------------+-----------+-------------------------------
| format             | str             | %d        | Format string for the value
+--------------------+-----------------+-----------+-------------------------------
| ...                | ...             |           |

Available fiat attributes for AnyDataWithGui Generic attributes:
--------------------------------------------------------------------------------
+-------------+--------+--------------------+-------------------------------------
| Name        | Type   | Default            | Explanation
+=============+========+====================+=====================================
| label       | str    |                    | A label for the parameter. If empty,
|             |        |                    | function parameter name is used
+-------------+--------+--------------------+-------------------------------------
| validator   | object | None               | Function to validate a parameter valu
|             |        |                    | raise a ValueError if invalid, or re
|             |        |                    | value (possibly modified)
+-------------+--------+--------------------+-------------------------------------
| ...         | ...    |                    |
```

## Example: Fiat Attributes in Action

In the example below, we customize the GUI for the function `interactive_histogram` by setting fiat attributes for the number of data points, the number of bars, the mean, and the standard deviation.

```python
import fiatlight
import matplotlib; matplotlib.use('Agg')  # Required to display the figure in the G
from matplotlib.figure import Figure

@fiatlight.with_fiat_attributes(
    # Label displayed as the title of the function node
    label="Interactive histogram",

    # Edit the number of data points with a logarithmic slider
    # Note: by default, you can ctrl+click on a slider to input a value directly,
    #       this is disabled here with nb_data__slider_no_input
    nb_data__label="Nb data",
    nb_data__edit_type="slider",
    nb_data__range=(100, 1_000_000),
    nb_data__slider_logarithmic=True,
    nb_data__slider_no_input=True,

    # Edit the number of bars with a knob
    n_bars__label="Number of bars",
    n_bars__edit_type="drag",
    n_bars__range=(1, 300),

    # Edit the average with a slider for a float value with any range
    # (the slider range will adapt interactively, when dragging far to the left or
    average__label="Mean",
    average__edit_type="slider_float_any_range",
    average__range=(-5, 5),

    # Edit the standard deviation with a drag
    sigma__label="Std deviation",
    sigma__edit_type="knob",
    sigma__range=(0.1, 5),

)
def interactive_histogram(
    nb_data: int = 4000, n_bars: int = 50, sigma: float = 1, average: float = 500
) -> Figure:
    '''Generate an interactive histogram with adjustable number of bars, mean, and
    import numpy as np
    import matplotlib.pyplot as plt

    data = np.random.normal(loc=average, scale=sigma, size=nb_data)
    bins = np.linspace(np.min(data), np.max(data), n_bars)
    fig, ax = plt.subplots()
    ax.hist(data, bins=bins, color="blue", alpha=0.7)
    return fig

fiatlight.run(interactive_histogram, app_name="Fiat attributes")
```




# Gui Registry

`fiatlight.fiat_togui` is the central module that is able to associate a GUI with a type.

It uses sophisticated mechanisms to inspect the type of function parameters and return values.

It handles a registry of types and their associated GUIs, to which you can add your own types, by calling `fiatlight.register_type(DataType, DataTypeWithGui)`, where `DataType` is the type you want to register, and `DataTypeWithGui` is the class that will handle the GUI for this type.

`DataTypeWithGui` must inherit from `AnyDataWithGui` and implement the necessary callbacks.

## Explore the registry

The `fiatlight` command line utility is a powerful tool that allows you to explore the available widgets and types in Fiatlight. It can be used to list the available types, to print the GUI info for a given type, and to run a GUI demo for a given type.

Here is the help message for the `fiatlight` command line utility:

```
%%bash
fiatlight --help
```

```
INFO: Showing help with the command 'fiatlight -- --help'.

NAME
    fiatlight

SYNOPSIS
    fiatlight COMMAND

COMMANDS
    COMMAND is one of the following:

     types
       List registered types, with a possible query to filter them. Add an optional

     gui
       Print the info and fiat attributes available for a given type. Add the datat

     fn_attrs
       Display the available fiat attributes for a function
```

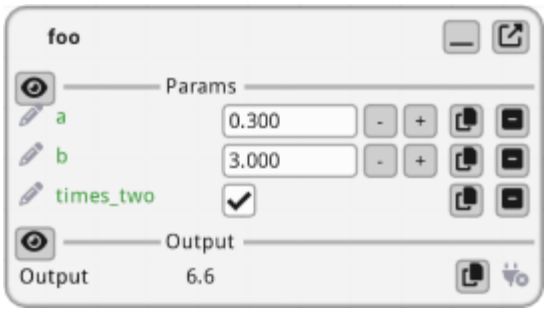
See the page Tutorials/fiatlight command line utility for more information.

## Primitive types

The primitive types `int`, `float`, `str`, `bool` are registered by default.

## Basic example

```python
import fiatlight as fl
def foo(a: float, b: float = 3.0, times_two: bool = False) -> float:
    return (a + b) * (2 if times_two else 1)

# Run an app that displays the GUI for the function
# where the user can input the values of the parameters
# (or use the default values)
fl.run(foo, app_name="Primitive Basic")
```

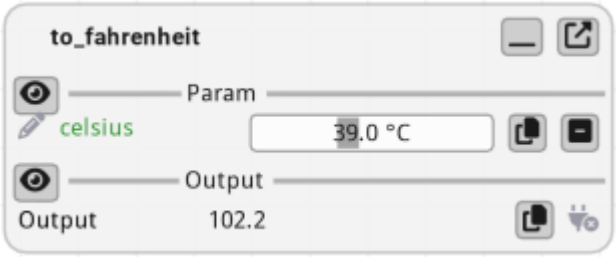



## Example with custom GUI options

The GUI for these primitive types is extensively configurable via fiat attributes. Below, we customize the GUI for the `celsius` parameter to be a slider ranging from 0 to 100, with a specific format for displaying the value.

See FunctionWithGui for a comprehensive list of all the available attributes (in the “Customizing parameters GUI” section).

```python
import fiatlight as fl


@fl.with_fiat_attributes(celsius__range=(0, 100), celsius__format="%.1f °C")
def to_fahrenheit(celsius: float) -> float:
    return celsius * 9 / 5 + 32


fl.run(to_fahrenheit, app_name="Primitive Custom")
```




## Range limited numeric types

As a convenience, Fiatlight includes those predefined types for which the GUI will take into account their boundings.

```
from typing import NewType

# Float types with specific ranges (bounds included)
Float_0_1 = NewType("Float_0_1", float)  # 0 to 1
Float_0_1.__doc__ = "synonym for float in [0, 1] (NewType)"

Float__1_1 = NewType("Float__1_1", float)  # -1 to 1
Float__1_1.__doc__ = "synonym for float in [-1, 1] (NewType)"

PositiveFloat = NewType("PositiveFloat", float)  # Any positive float ( strictly gr
PositiveFloat.__doc__ = "synonym for float > 0 (strictly greater than 0) (NewType)"

# Int types with specific ranges (bounds included)
Int_0_255 = NewType("Int_0_255", int)  # 0 to 255
Int_0_255.__doc__ = "synonym for int in [0, 255] (NewType)"
```

# File name types

Several file types names are registered by default. They are synonyms for `str` and are used to specify file paths. They will be presented with a file dialog in the GUI.

```
from fiatlight.fiat_notebook import look_at_code
%look_at_python_code fiatlight.fiat_types.file_types
```

```python
from typing import NewType


# FilePath is a synonym of str, but when used as a function parameter,
# it will be displayed as a widget where you can select a file.
FilePath = NewType("FilePath", str)
FilePath.__doc__ = "synonym for str, describing a file path (NewType)"
# FilePath_Save is a synonym of str, but when used as a function parameter,
# it will be displayed as a widget where you can select a file to save to.
FilePath_Save = NewType("FilePath_Save", str)
FilePath_Save.__doc__ = "synonym for str, describing a file path for saving (NewType

# With ImagePath, you can select an image file.
ImagePath = NewType("ImagePath", FilePath)
ImagePath.__doc__ = "synonym for str, describing an image file path (NewType)"
ImagePath_Save = NewType("ImagePath_Save", FilePath_Save)
ImagePath_Save.__doc__ = "synonym for str, describing an image file path for saving

# With TextPath, you can select a text file.
TextPath = NewType("TextPath", FilePath)
TextPath.__doc__ = "synonym for str, describing a text file path (NewType)"
TextPath_Save = NewType("TextPath_Save", FilePath_Save)
TextPath_Save.__doc__ = "synonym for str, describing a text file path for saving (Ne

# With AudioPath, you can select an audio file.
AudioPath = NewType("AudioPath", FilePath)
AudioPath.__doc__ = "synonym for str, describing an audio file path (NewType)"
AudioPath_Save = NewType("AudioPath_Save", FilePath_Save)
AudioPath_Save.__doc__ = "synonym for str, describing an audio file path for saving

# With VideoPath, you can select a video file.
VideoPath = NewType("VideoPath", FilePath)
VideoPath.__doc__ = "synonym for str, describing a video file path (NewType)"
VideoPath_Save = NewType("VideoPath_Save", FilePath_Save)
VideoPath_Save.__doc__ = "synonym for str, describing a video file path for saving (
```

# Color types

Several color types are registered by default.

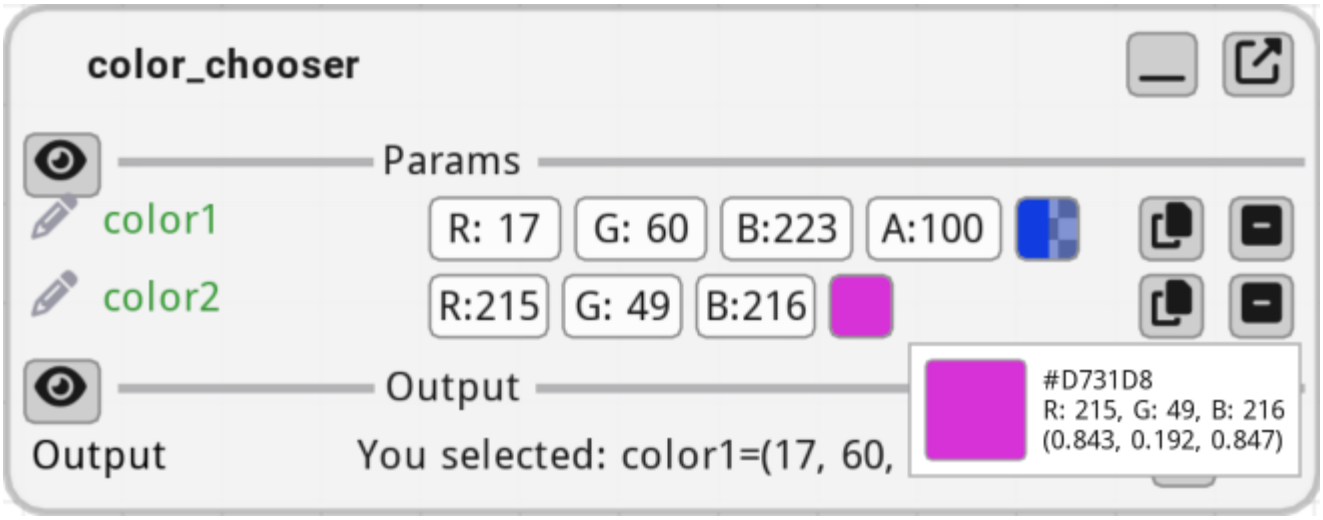
```
%look_at_python_code fiatlight.fiat_types.color_types
```

```python
from typing import NewType
from imgui_bundle import ImVec4


ColorRgb = NewType("ColorRgb", tuple[int, int, int])
ColorRgb.__doc__ = "synonym for tuple[int, int, int] describing an RGB color, with v

ColorRgba = NewType("ColorRgba", tuple[int, int, int, int])
ColorRgba.__doc__ = "synonym for tuple[int, int, int, int] describing an RGBA color,

ColorRgbFloat = NewType("ColorRgbFloat", tuple[float, float, float])
ColorRgbFloat.__doc__ = (
    "synonym for tuple[float, float, float] describing an RGB color, with values in
)

ColorRgbaFloat = NewType("ColorRgbaFloat", tuple[float, float, float, float])
ColorRgbaFloat.__doc__ = (
    "synonym for tuple[float, float, float, float] describing an RGBA color, with va
)


def _int255_to_float(value: int) -> float:
    return value / 255.0


def _float_to_int255(value: float) -> int:
    return int(value * 255)


def color_rgb_to_color_rgb_float(color_rgb: ColorRgb) -> ColorRgbFloat:
    return ColorRgbFloat(tuple(_int255_to_float(value) for value in color_rgb))  # t


def color_rgba_to_color_rgba_float(color_rgba: ColorRgba) -> ColorRgbaFloat:
    return ColorRgbaFloat(tuple(_int255_to_float(value) for value in color_rgba))  #


def color_rgb_float_to_color_rgb(color_rgb_float: ColorRgbFloat) -> ColorRgb:
    return ColorRgb(tuple(_float_to_int255(value) for value in color_rgb_float))  #


def color_rgba_float_to_color_rgba(color_rgba_float: ColorRgbaFloat) -> ColorRgba:
    return ColorRgba(tuple(_float_to_int255(value) for value in color_rgba_float))


def color_rgb_to_color_rgba(color_rgb: ColorRgb) -> ColorRgba:
    return ColorRgba(color_rgb + (255,))


def color_rgb_float_to_color_rgba_float(color_rgb_float: ColorRgbFloat) -> ColorRgba
    return ColorRgbaFloat(color_rgb_float + (1.0,))
```

```
def color_rgb_to_imvec4(v: ColorRgb) -> ImVec4:
    return ImVec4(v[0] / 255.0, v[1] / 255.0, v[2] / 255.0, 1.0)


def color_rgba_to_imvec4(v: ColorRgba) -> ImVec4:
    return ImVec4(v[0] / 255.0, v[1] / 255.0, v[2] / 255.0, v[3] / 255.0)


def color_rgb_float_to_imvec4(v: ColorRgbFloat) -> ImVec4:
    return ImVec4(v[0], v[1], v[2], 1.0)


def color_rgba_float_to_imvec4(v: ColorRgbaFloat) -> ImVec4:
    return ImVec4(v[0], v[1], v[2], v[3])
```

*Example: using color types in function*

```
import fiatlight as fl
from fiatlight.fiat_types import ColorRgb, ColorRgba

def color_chooser(color1: ColorRgba, color2: ColorRgb) -> str:
    return f"You selected: {color1=}, {color2=}"

fl.run(color_chooser, app_name="Color Chooser")
```



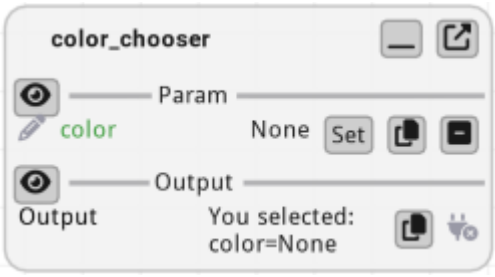

# Optional types

If a type is registered, its optional version is also registered.

*Example: using an optional color in a function*

(In this example, the user needs to click on “Set” to set a value to the optional color)

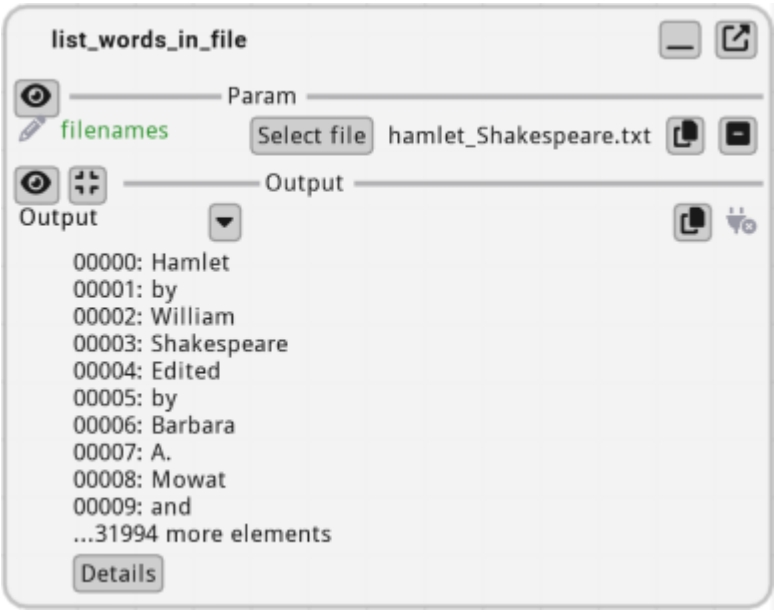
```python
import fiatlight as fl
from fiatlight.fiat_types import ColorRgb, ColorRgba

def color_chooser(color: ColorRgb | None = None) -> str:
    return f"You selected: {color=}"

fl.run(color_chooser, app_name="Optional Color")
```




## Lists

A very basic support is provided for lists. It does not allow to edit the values. However, it can present a list of values using (all of them will be rendered as string using str() function).

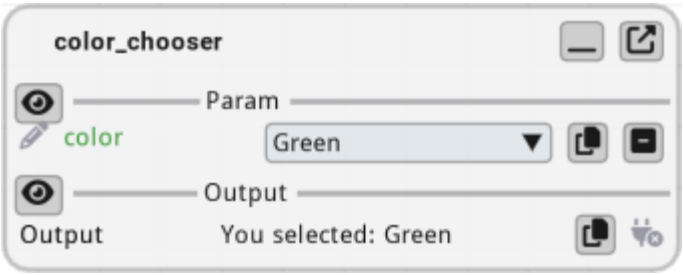
```python
import fiatlight as fl
from fiatlight.fiat_types import TextPath

def list_words_in_file(filenames: TextPath) -> list[str]:
    with open(filenames) as f:
        return f.read().split()

fl.run(list_words_in_file, app_name="List Words in File")
```




# Enum classes

Enum classes are automatically associated to a GUI.

```
import fiatlight as fl
from enum import Enum

class Color(Enum):
    Red = 1
    Green = 2
    Blue = 3

def color_chooser(color: Color) -> str:
    return f"You selected: {color.name}"

fl.run(color_chooser, app_name="Enum Color")
```



# Gui Nodes

Gui Nodes are specialized nodes in Fiatlight, dedicated to functions which do not return values but instead displaying a user interface using ImGui widgets. Gui Nodes are called at every frame, ensuring that the GUI is always responsive and up-to-date.

Gui Nodes are particularly useful for:

- Displaying interactive visualizations (plots, etc)
- Creating dashboards
- Providing user controls (e.g., sliders, buttons) that alter global variables

> Notes:
>
> - Gui Nodes are not meant to return values
> - Your GUI function should be fast. If you need to perform heavy computations, consider using AnyDataWithGui, where the “on_change” callback can be used to cache heavy computations.

## Example: Visualizing a Heart Curve with a GUI Node

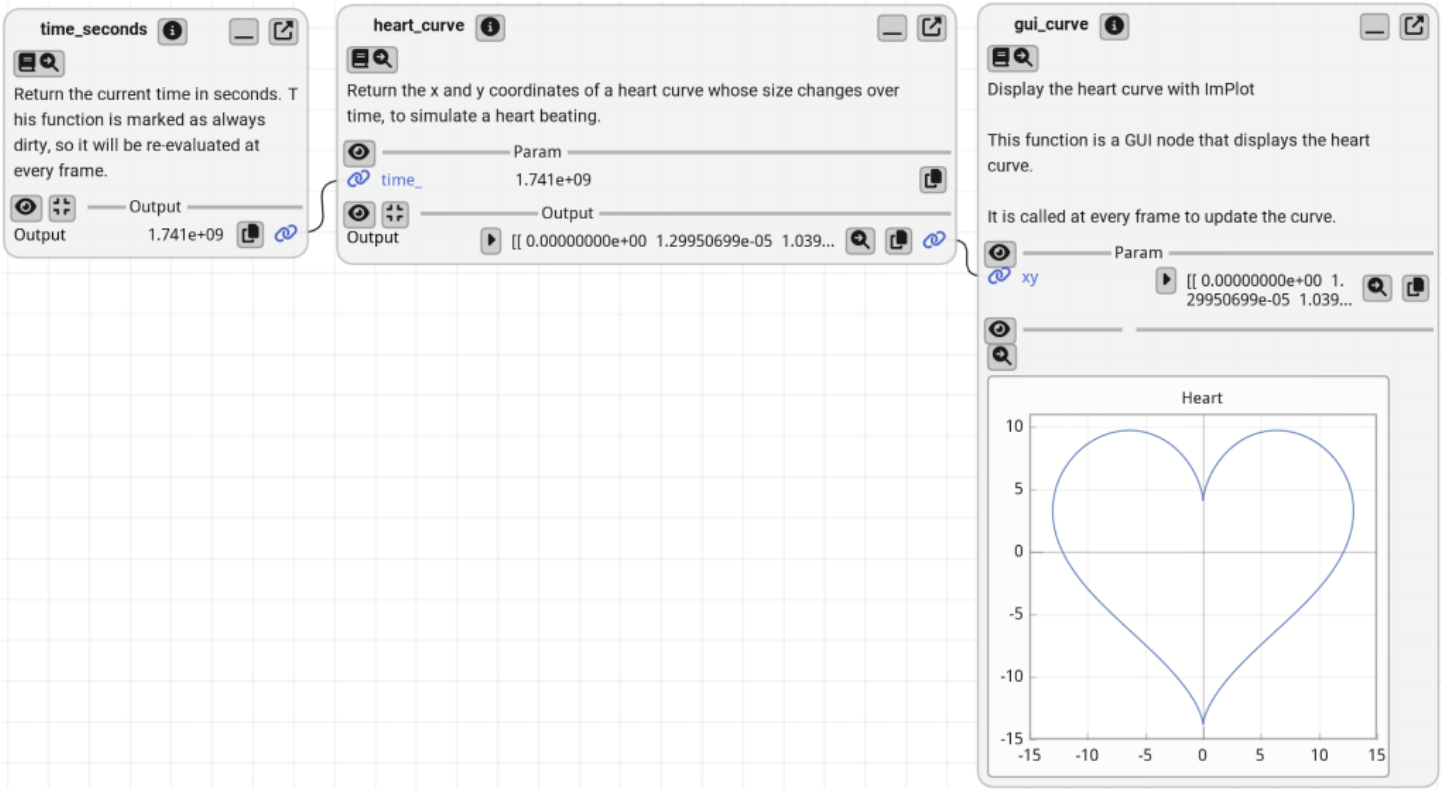
In this example, we demonstrate how to create a GUI node that visualizes a heart curve. The size of the heart dynamically changes over time to simulate a heartbeat.

**Explanation**:

1. time_seconds: This function returns the current time in seconds and is set to always be re-evaluated at every frame.
2. heart_curve: Generates the x and y coordinates of a heart curve that changes size over time to simulate a heartbeat.
3. gui_curve: A GUI node that visualizes the heart curve using ImPlot. It updates the curve at every frame to reflect the beating heart.

4. gui_curve is a gui function. So, we wrap it in a GuiNode to display the heart curve with either `GuiNode(gui_curve)` or `graph.add_gui_node(gui_curve)`.

```python
import fiatlight as fl
from imgui_bundle import hello_imgui, implot
import numpy as np
from numpy.typing import ArrayLike
import time

@fl.with_fiat_attributes(invoke_always_dirty=True)
def time_seconds() -> float:
    """Return the current time in seconds.
    This function is marked as always dirty, so it will be re-evaluated at every fr
    """
    return time.time()

def heart_curve(time_: float) -> ArrayLike:
    """Return the x and y coordinates of a heart curve whose size changes over time
    to simulate a heart beating.
    """
    vals = np.arange(0, np.pi * 2, 0.01)
    x0 = np.power(np.sin(vals), 3) * 16
    y0 = 13 * np.cos(vals) - 5 * np.cos(2 * vals) - 2 * np.cos(3 * vals) - np.cos(4

    # Heart pulse rate and time tracking
    heart_pulse_rate = 80
    phase = time_ * heart_pulse_rate / (np.pi * 2)
    k = 0.8 + 0.1 * np.cos(phase)
    return np.array([x0 * k, y0 * k])

def gui_curve(xy: ArrayLike) -> None:
    """Display the heart curve with ImPlot

    This function is a GUI node that displays the heart curve.

    It is called at every frame to update the curve.
    """
    if implot.begin_plot("Heart", hello_imgui.em_to_vec2(21, 21)):
        implot.setup_axes_limits(-15, 15, -15, 11)
        implot.plot_line("", xy[0], xy[1])
        implot.end_plot()

# Run the graph
# Method 1: Using the run function, and wrapping the gui_curve function in a GuiNod
fl.run([time_seconds, heart_curve, fl.GuiNode(gui_curve)], app_name="HeartCurve")

# Method 2: Using a FunctionsGraph
# graph = fl.FunctionsGraph()
# graph.add_function(time_seconds)
# graph.add_function(heart_curve)
# graph.add_gui_node(gui_curve)  # Add the gui_curve function as a GuiNode
# graph.add_link(time_seconds, heart_curve)
# graph.add_link(heart_curve, gui_curve)
# fl.run(graph, app_name="HeartCurve")
```

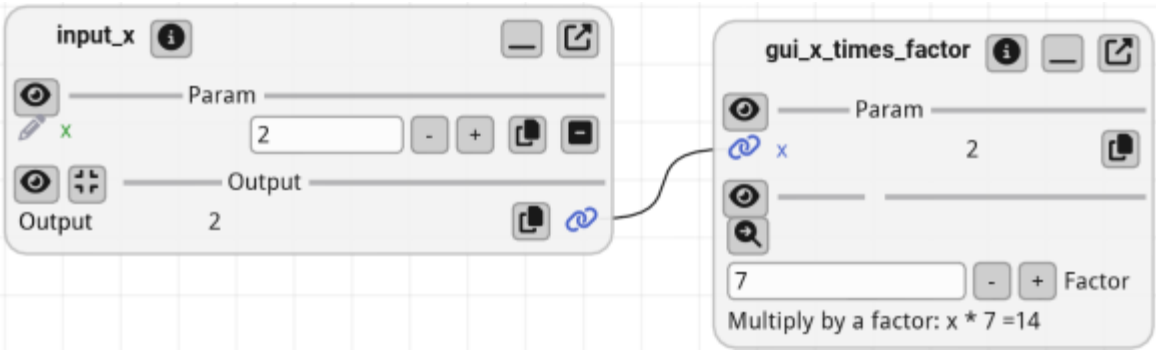



## Example: a GUI node with serializable state

When adding a GuiNode, you can pass a serializable data class to store the options of the GUI function. This allows you to save the state of the GUI function and reload it when restarting the application.

In the example below, we demonstrate how to create a GUI node that multiplies an input value by a factor. The factor can be adjusted by the user and is stored in a serializable data class. The factor value is reloaded upon restarting the application.

```python
import fiatlight as fl
from imgui_bundle import imgui
from pydantic import BaseModel

def input_x(x: int) -> int:
    """a function that will be displayed in the function graph, in order to let the
    return x

class WhatToMultiply(BaseModel):
    """A serializable data class that will be used to store the options of the GUI
    factor: int = 3

WHAT_TO_MULTIPLY = WhatToMultiply()

def gui_x_times_factor(x: int) -> None:
    """A GUI function that multiplies the input by a serializable factor.
    It will be added via graph.add_gui_node(gui_x_times_factor, gui_serializable_da
    It uses a serializable data class to store its options, which will be reloaded
    """
    _, WHAT_TO_MULTIPLY.factor = imgui.input_int("Factor", WHAT_TO_MULTIPLY.factor)
    imgui.text(f"Multiply by a factor: x * {WHAT_TO_MULTIPLY.factor} ={x * WHAT_TO_

# Run the graph
fl.run([input_x, fl.GuiNode(gui_x_times_factor, gui_serializable_data=WHAT_TO_MULTI
```




# Fiatlight command line utility

The `fiatlight` command line utility is a powerful tool that allows you to explore the available widgets and types in Fiatlight. It can be used to list the available types, to print the GUI info for a given type, and to run a GUI demo for a given type.

Here is the help message for the `fiatlight` command line utility (ignore the `%%bash` magic command, it is used to run bash commands in Jupyter notebooks):

```
%%bash
fiatlight --help
```

```
INFO: Showing help with the command 'fiatlight -- --help'.

NAME
    fiatlight

SYNOPSIS
    fiatlight COMMAND

COMMANDS
    COMMAND is one of the following:

     types
       List registered types, with a possible query to filter them. Add an optional

     gui
       Print the GUI info for a given type. Add the GUI type name as an argument (i
```

## List registered types

The `types` command lists the registered types in Fiatlight. You can filter the types by adding an optional query.

In the example below, we will run the `fiatlight types str` command to list all the types that contain the string “str”.

```
%%bash
fiatlight types str
```

| Data Type | Gui Type |
|---|---|
| str | fiatlight.fiat_togui.str_with<br>A Gui for a string with re<br>multiline editing. |
| fiatlight.fiat_types.fiat_number_types.PositiveFlo<br>at<br>synonym for float > 0 (strictly greater than 0)<br>(NewType) | fiatlight.fiat_togui.primitiv<br>A highly customizable floa |
| fiatlight.fiat_types.file_types.FilePath<br>synonym for str, describing a file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.TextPath<br>synonym for str, describing a text file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.ImagePath<br>synonym for str, describing an image file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.AudioPath<br>synonym for str, describing an audio file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.VideoPath<br>synonym for str, describing a video file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.FilePath_Save<br>synonym for str, describing a file path for<br>saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>dialog. |
| fiatlight.fiat_types.file_types.TextPath_Save<br>synonym for str, describing a text file path for<br>saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| fiatlight.fiat_types.file_types.ImagePath_Save<br>synonym for str, describing an image file path<br>for saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| fiatlight.fiat_types.file_types.AudioPath_Save<br>synonym for str, describing an audio file path<br>for saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| fiatlight.fiat_types.file_types.VideoPath_Save<br>synonym for str, describing a video file path<br>for saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| fiatlight.fiat_kits.fiat_ai.prompt.Prompt | fiatlight.fiat_kits.fiat_ai.p |

```
|   synonym for a string used as a prompt, used for  |   A Gui to edit a prompt, wi
| AI text and image generation models (NewType)      | in a popup.
+----------------------------------------------------+-----------------------------
```

Notes:

- If you do not include the `str` argument, all the types will be printed.

## Print the GUI info for a given type

The `gui_info` command prints the GUI info for a given type. You can specify the GUI type name or the data type name as an argument. If you do not provide a type name, all the GUI widget names will be printed.

### Example: Print the GUI info for StrWithGui

In the example below, we will run the `fiatlight gui_info StrWithGui` command to print the GUI info for the `StrWithGui` widget.

```bash
%%bash
fiatlight gui str
```

GUI type: str
==============
  A Gui for a string with resizable input text, with a popup for multiline editing.

  Available custom attributes for StrWithGui:
  ---------------------------------------------------------------------------

| Name | Type | Default | Explanation |
|---|---|---|---|
| width_em | float | 15.0 | Initial width of th (in em unit). Can b True |
| size_multiline_em | tuple[float, float] | (60.0, 15.0) | Initial size of the em unit) |
| hint | str | | Hint text for the i |
| allow_multiline_edit | bool | False | Whether the user ca multiline string (w |
| resizable | bool | True | Whether the single |
| wrap_multiline | bool | False | Whether the text is a multiline string |
| wrap_multiline_width | int | 80 | Width at which the presented as a mult |

  Available custom attributes for AnyDataWithGui Generic attributes:
  ---------------------------------------------------------------------------

| Name | Type | Default | Explanation |
|---|---|---|---|
| | | | **Generic attributes** |
| validate_value | object | None | Function to validate a paramete return DataValidationResult.ok( |
| label | str | | A label for the parameter. If e function parameter name is used |
| tooltip | str | | An optional tooltip to be displ |
| label_color | ImVec4 | ImVec4(0.000000, 0.000000, 0.000000, 1.000000) | The color of the label (will us text color if not provided) |

Code to test this GUI type:
---------------------------
```python
```

```
import typing
import fiatlight

@fiatlight.with_fiat_attributes(
    str_param__width_em = 15.0,
    str_param__size_multiline_em = (60.0, 15.0),
    str_param__hint = "",
    str_param__allow_multiline_edit = False,
    str_param__resizable = True,
    str_param__wrap_multiline = False,
    str_param__wrap_multiline_width = 80,
    #  Generic attributes
    str_param__validate_value = None,
    str_param__label = "",
    str_param__tooltip = "",
    str_param__label_color = ImVec4(0.000000, 0.000000, 0.000000, 1.000000))
def f(str_param: str) -> str:
    return str_param

fiatlight.run(f)
```

## Example: Print the GUI info for ImageWithGui

```bash
%%bash
fiatlight gui ImageWithGui
```

GUI type: ImageWithGui
======================
A highly sophisticated GUI for displaying and analysing images. Zoom/Pan, show cha

Available custom attributes for fiat_image.ImageWithGui:

| Name | Type | Default | Explanation |
|---|---|---|---|
| | | | **Main attribute |
| only_display | bool | False | Only display the zoom, no pan |
| image_display_size | tuple[int, int] | (200, 0) | Initial size of height). One of |
| zoom_key | str | z | Key to zoom in same zoom key w |
| is_color_order_bgr | bool | True | Color order is uses BGR by defa |
| can_resize | bool | True | Can resize the the bottom righ |
| | | | **Channels** |
| show_channels | bool | False | Show channels |
| channel_layout_vertically | bool | False | Layout channels |
| | | | **Zoom & Pan** |
| pan_with_mouse | bool | True | Pan with mouse |
| zoom_with_mouse_wheel | bool | True | Zoom with mouse |
| | | | **Info displaye |
| show_school_paper_background | bool | True | Show school pap is unzoomed |
| show_alpha_channel_checkerboard | bool | True | Show alpha chan |
| show_grid | bool | True | Show grid with |
| draw_values_on_zoomed_pixels | bool | True | Draw values on |
| | | | **Info displaye |
| show_image_info | bool | True | Show image info |

| show_pixel_info | bool | True | Show pixel info
| | | | position under
| | | | **Control butto
| show_zoom_buttons | bool | True | Show zoom butto
| show_options_panel | bool | True | Show options pa
| show_options_button | bool | True | Show options bu
| show_inspect_button | bool | True | Show the inspec
| | | | a large version
| | | | Inspector

Available custom attributes for AnyDataWithGui Generic attributes:

| Name | Type | Default | Explanation |
|---|---|---|---|
| | | | **Generic attributes** |
| validate_value | object | None | Function to validate a paramete return DataValidationResult.ok( |
| label | str | | A label for the parameter. If e function parameter name is used |
| tooltip | str | | An optional tooltip to be displ |
| label_color | ImVec4 | ImVec4(0.000000, 0.000000, 0.000000, 1.000000) | The color of the label (will us text color if not provided) |

Code to test this GUI type:

```python
import typing
import fiatlight

@fiatlight.with_fiat_attributes(
    #  Main attributes for the image viewer
    union_param__only_display = False,
    union_param__image_display_size = (200, 0),
    union_param__zoom_key = "z",
    union_param__is_color_order_bgr = True,
    union_param__can_resize = True,
    #  Channels
    union_param__show_channels = False,
    union_param__channel_layout_vertically = False,
    #  Zoom & Pan
    union_param__pan_with_mouse = True,
```

```
    union_param__zoom_with_mouse_wheel = True,
    #  Info displayed on image
    union_param__show_school_paper_background = True,
    union_param__show_alpha_channel_checkerboard = True,
    union_param__show_grid = True,
    union_param__draw_values_on_zoomed_pixels = True,
    #  Info displayed under the image
    union_param__show_image_info = True,
    union_param__show_pixel_info = True,
    #  Control buttons under the image
    union_param__show_zoom_buttons = True,
    union_param__show_options_panel = True,
    union_param__show_options_button = True,
    union_param__show_inspect_button = True,
    #  Generic attributes
    union_param__validate_value = None,
    union_param__label = "",
    union_param__tooltip = "",
    union_param__label_color = ImVec4(0.000000, 0.000000, 0.000000, 1.000000))
def f(union_param: typing.Union[fiatlight.fiat_kits.fiat_image.image_types.ImageU8_
    return union_param

fiatlight.run(f)
```

## Annex: list of registered types

By running the `fiatlight types` command, you can list all the registered types in Fiatlight. Here is a list of the available types:

```bash
%%bash
fiatlight types
```

| Data Type | Gui Type |
|---|---|
| fiatlight.fiat_types.fiat_number_types.Float_0_1<br>synonym for float in [0, 1] (NewType) | fiatlight.fiat_togui.primitiv<br>A highly customizable floa |
| fiatlight.fiat_types.fiat_number_types.Float__1_1<br>synonym for float in [-1, 1] (NewType) | fiatlight.fiat_togui.primitiv<br>A highly customizable floa |
| fiatlight.fiat_types.fiat_number_types.Int_0_255<br>synonym for int in [0, 255] (NewType) | fiatlight.fiat_togui.primitiv<br>A highly customizable int v |
| int | fiatlight.fiat_togui.primitiv<br>A highly customizable int v |
| float | fiatlight.fiat_togui.primitiv<br>A highly customizable floa |
| str | fiatlight.fiat_togui.str_witl<br>A Gui for a string with re<br>multiline editing. |
| bool | fiatlight.fiat_togui.primitiv<br>A bool widget. Can use a cl |
| fiatlight.fiat_types.color_types.ColorRgb<br>synonym for tuple[int, int, int] describing an RGB color, with values in [0, 255] (NewType) | fiatlight.fiat_togui.primitiv<br>A nice color picker for RGI |
| fiatlight.fiat_types.color_types.ColorRgba<br>synonym for tuple[int, int, int, int] describing an RGBA color, with values in [0, 255] (NewType) | fiatlight.fiat_togui.primitiv<br>A nice color picker for RGI |
| fiatlight.fiat_types.color_types.ColorRgbFloat<br>synonym for tuple[float, float, float] describing an RGB color, with values in [0, 1] (NewType) | fiatlight.fiat_togui.primitiv<br>A nice color picker for RGI |
| fiatlight.fiat_types.color_types.ColorRgbaFloat<br>synonym for tuple[float, float, float, float] describing an RGBA color, with values in [0, 1] (NewType) | fiatlight.fiat_togui.primitiv<br>A nice color picker for RGI |
| fiatlight.fiat_types.fiat_number_types.PositiveFloat<br>synonym for float > 0 (strictly greater than 0) (NewType) | fiatlight.fiat_togui.primitiv<br>A highly customizable floa |
| (dataclass) fiatlight.fiat_togui.dataclass_examples.ExampleDataclass<br>ExampleDataclass(x: int = 0, y: str = 'Hello') | fiatlight.fiat_togui.dataclas<br>A sophisticated GUI for a c |
| (BaseModel) fiatlight.fiat_togui.dataclass_example | fiatlight.fiat_togui.basemod |

| s.ExampleBaseModel | A sophisticated GUI for a |
|---|---|
| fiatlight.fiat_types.file_types.FilePath<br>synonym for str, describing a file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.TextPath<br>synonym for str, describing a text file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.ImagePath<br>synonym for str, describing an image file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.AudioPath<br>synonym for str, describing an audio file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.VideoPath<br>synonym for str, describing a video file path<br>(NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec |
| fiatlight.fiat_types.file_types.FilePath_Save<br>synonym for str, describing a file path for<br>saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>dialog. |
| fiatlight.fiat_types.file_types.TextPath_Save<br>synonym for str, describing a text file path for<br>saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| fiatlight.fiat_types.file_types.ImagePath_Save<br>synonym for str, describing an image file path<br>for saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| fiatlight.fiat_types.file_types.AudioPath_Save<br>synonym for str, describing an audio file path<br>for saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| fiatlight.fiat_types.file_types.VideoPath_Save<br>synonym for str, describing a video file path<br>for saving (NewType) | fiatlight.fiat_togui.file_ty<br>A Gui that enable to selec<br>file dialog. |
| (BaseModel) fiatlight.fiat_kits.fiat_image.cv_colo<br>r_type.ColorConversion<br>A color conversion from one color space to<br>another (color spaces use the ColorType enum). | fiatlight.fiat_togui.basemod<br>A sophisticated GUI for a |
| (BaseModel) fiatlight.fiat_kits.fiat_image.lut_typ<br>es.ColorLutParams | fiatlight.fiat_togui.basemod<br>A sophisticated GUI for a |
| (BaseModel) fiatlight.fiat_kits.fiat_image.camera_<br>image_provider.CameraParams<br>Parameters for the camera image provider | fiatlight.fiat_togui.basemod<br>A sophisticated GUI for a |

| None<br>All types whose name starts with fiatlight.fiat_kits.fiat_image.image_types.Image | fiatlight.fiat_kits.fiat_ima<br>A highly sophisticated GUI Zoom/Pan, show channels, sho etc. |
|---|---|
| None<br>Union of types whose name starts with fiatlight.fiat_kits.fiat_image.image_types.Image | fiatlight.fiat_kits.fiat_ima<br>A highly sophisticated GUI Zoom/Pan, show channels, sho etc. |
| (BaseModel) fiatlight.fiat_kits.fiat_image.lut_types.LutParams<br>Simple parameters to create a LUT (Look-Up Table) transformation to an image | fiatlight.fiat_kits.fiat_ima<br>A GUI for LutParams, allow Look-Up Table transformation |
| fiatlight.fiat_kits.fiat_implot.array_types.FloatMatrix_Dim1<br>synonym for a 1D ndarray of floats (NewType) | fiatlight.fiat_kits.fiat_imp<br>A GUI for presenting 1D or array as a line, scatter (+ small enough) |
| fiatlight.fiat_kits.fiat_implot.array_types.FloatMatrix_Dim2<br>synonym for a 2D ndarray of floats (NewType) | fiatlight.fiat_kits.fiat_imp<br>A GUI for presenting 1D or array as a line, scatter (+ small enough) |
| fiatlight.fiat_kits.fiat_ai.prompt.Prompt<br>synonym for a string used as a prompt, used for AI text and image generation models (NewType) | fiatlight.fiat_kits.fiat_ai.<br>A Gui to edit a prompt, wi in a popup. |
| pandas.core.frame.DataFrame | fiatlight.fiat_kits.fiat_dat<br>A class to present a panda and other features. Open in |
| matplotlib.figure.Figure<br>The top level container for all the plot elements. | fiatlight.fiat_kits.fiat_mat<br>A Gui that can present a r |

# Fiat Tuning: Tune functions

## Introduction

Fiatlight provides you with powerful tools to visually debug the intermediate states of your function.

By adding a fiat_tuning attribute to a function, you can provide additional information that will be displayed in the GUI node for this function. This attribute is a dictionary and can contain named

data values or descendants of AnyDataWithGui. This information can be used to fine-tune the function, debug it, or visualize intermediate states.

Moreover, this information can be updated in the GUI, even if the function is a long-running process called asynchronously.

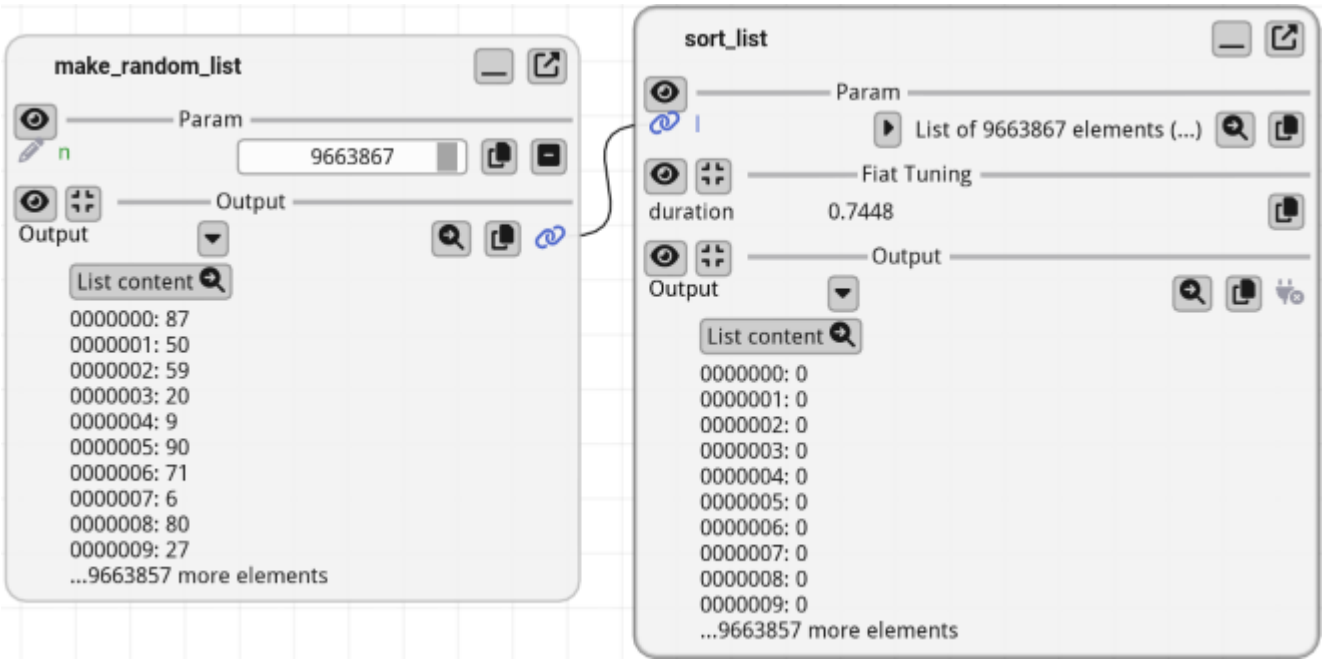
> *Example: The image below shows a sort competition between different algorithms. The GUI nodes display in real time the evolving state of each algorithm, using "fiat_tuning".*
>
> *See "Advanced Video Tutorial: Sort Algorithm Visualizer" below for more details.*
>
> 

## Example: Measure Execution Time

In the example below, we will add a simple float into the fiat_tuning attribute of the sort_list function. This float will represent the duration of the sort operation.

The collapsible region "Fiat Tuning" will display this duration:we can see that in this example, it takes about 0.75 seconds to sort a list of 10,000,000 elements.

```python
import fiatlight as fl
import time

@fl.with_fiat_attributes(n__range=(1, 10_000_000))
def make_random_list(n: int) -> list[int]:
    import random
    return [random.randint(0, 100) for _ in range(n)]


def sort_list(l: list[int]) -> list[int]:
    start = time.time()
    r = sorted(l)
    duration = time.time() - start
    fl.add_fiat_attributes(sort_list, fiat_tuning={"duration": duration})
    return r

fl.run([make_random_list, sort_list], app_name="Sort duration")
```




## Example: Tune using an Image

The fiat_tuning attribute can also be used to display widgets (which must be descendants of AnyDataWithGui) in the GUI node.

In the example below, we will add an image widget (ImageWithGui) into the fiat_tuning attribute.

> *demos/images/toon_edges.py is a good example of how to use the `fiat_tuning` attribute.*

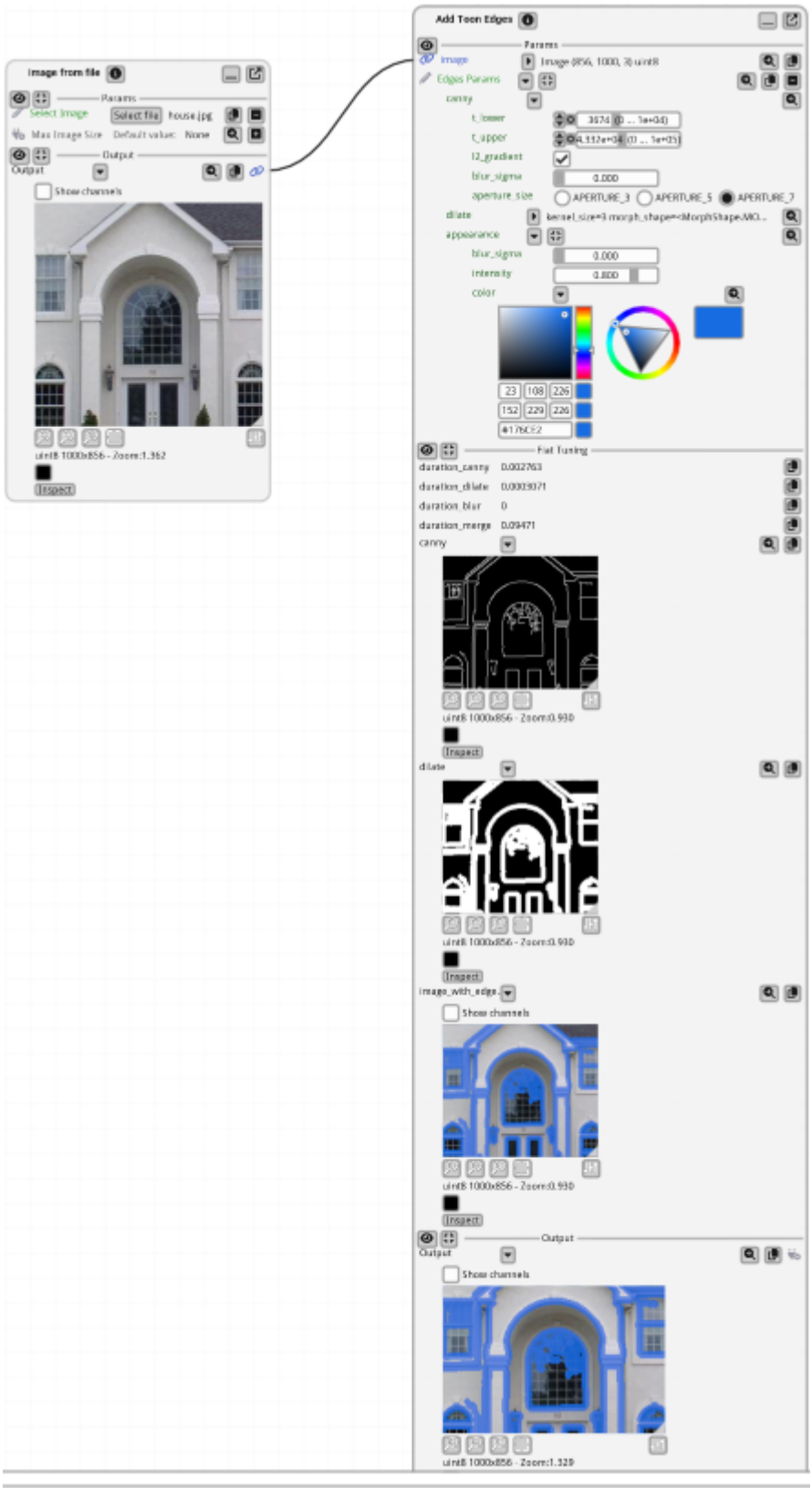
> `add_toon_edges` *is a complex function that adds a toon effect to an image, by adding colored edges to the image contours. The contour detection is extremely sensitive to the parameters, and the* `fiat_tuning` *attribute is used to display the intermediate states of the function in the GUI.*

Here are some commented extracts of the function:

```
from fiatlight.fiat_kits.fiat_image import ImageU8_3, ImageU8_1

def add_toon_edges(
image: ImageU8_3,
# ... lots of parameters ...
) -> ImageU8_3:
    edges: ImageU8_1 # = ...              (compute the edges)
    dilated_edges: ImageU8_1 #  = ...     (dilate the edges)
    image_with_edges: ImageU8_3  # = ... (superimpose the edges on the image)

    # fiat_tuning: add debug internals to ease fine-tuning the function inside the
    from fiatlight.fiat_kits.fiat_image import ImageWithGui

    # Add to fiat_tuning any variable you want to be able to fine-tune or debug in
    #     * Either a raw type (int, float, str, etc.): see durations
    #     * Or a descendant of AnyDataWithGui: see "canny", "dilate", "image_with_e
    fl.add_fiat_attributes(add_toon_edges, fiat_tuning={
        "duration_canny": duration_canny,
        "duration_dilate": duration_dilate,
        "duration_blur": duration_blur,
        "duration_merge": duration_merge,
        "canny": ImageWithGui(edges),
        "dilate": ImageWithGui(dilated_edges),
        "image_with_edges": ImageWithGui(image_with_edges),
    })

    # return the image with edges
    return image_with_edges
```

Once these internals are set, you can see the function “Internals” in the GUI:

```
import fiatlight as fl
from fiatlight.fiat_kits.fiat_image import ImageU8_GRAY, ImageU8_3, image_source
from fiatlight.demos.images.toon_edges import add_toon_edges

fl.run([image_source, add_toon_edges], app_name="Toon edges")
```




> *The image above shows the GUI node for the toon_edges function, with the expanded "Fiat Tuning" section: it displays the execution time of each step, as well as an image representation of the intermediate edges and dilated edges.*

# Debugging Functions exceptions

When a function raises an exception, Fiatlight catches and displays it without crashing the application. Instead, you will see a “Debug this exception” button that you can use to trigger the exception again.

This feature is invaluable for debugging and making your functions more robust. If you are using a debugger, you will be taken directly to the point where the exception occurred, with the correct inputs to reproduce the bug.

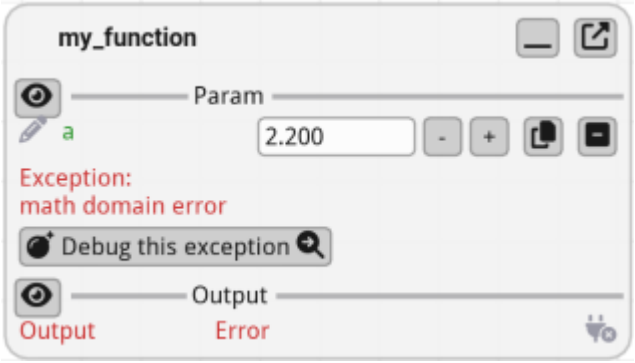
> _Note: this feature can be disabled with:
> `fl.get_fiat_config().run_config.catch_function_exceptions = False`

## Example: a Math Exception

With this setup, if the input value of a causes math.log(cos_a) to produce an error (when cos_a is negative).

Fiatlight will catch and display the exception, allowing you to debug it easily.

```python
import fiatlight as fl
import math

def my_function(a: float) -> float:
    cos_a = math.cos(a)
    r = math.log(cos_a)
    return r

fl.run(my_function, app_name="Math domain exception")
```




# Functions Graph

`FunctionsGraph` is one of the core classes of FiatLight: it represents a graph of functions, where the output of one function can be linked to the input of another function.

- **Source**: see its full code online
- **API**: FunctionsGraph API

# Creating a FunctionsGraph

## When a FunctionsGraph can be created automatically

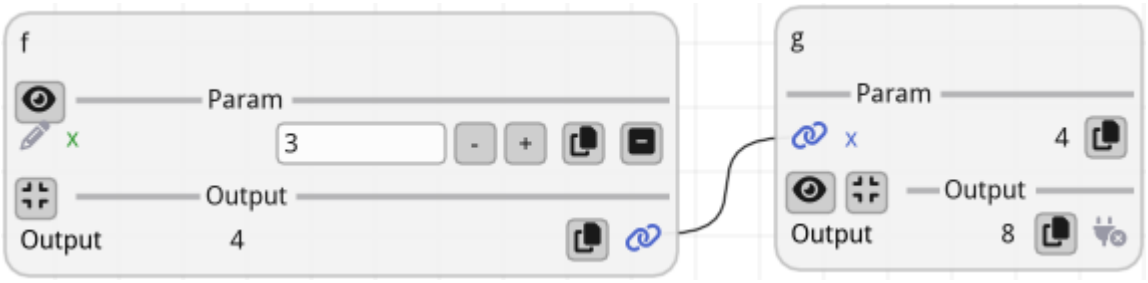
In simple cases (one function, or a list of functions that are chained together), you do not need to create a FunctionsGraph. See the examples below.

*Single function*:

```
import fiatlight as fl
def f(x: int) -> int:
    return x + 1
fl.run(f, app_name="Single function")
```

*Chained functions*:

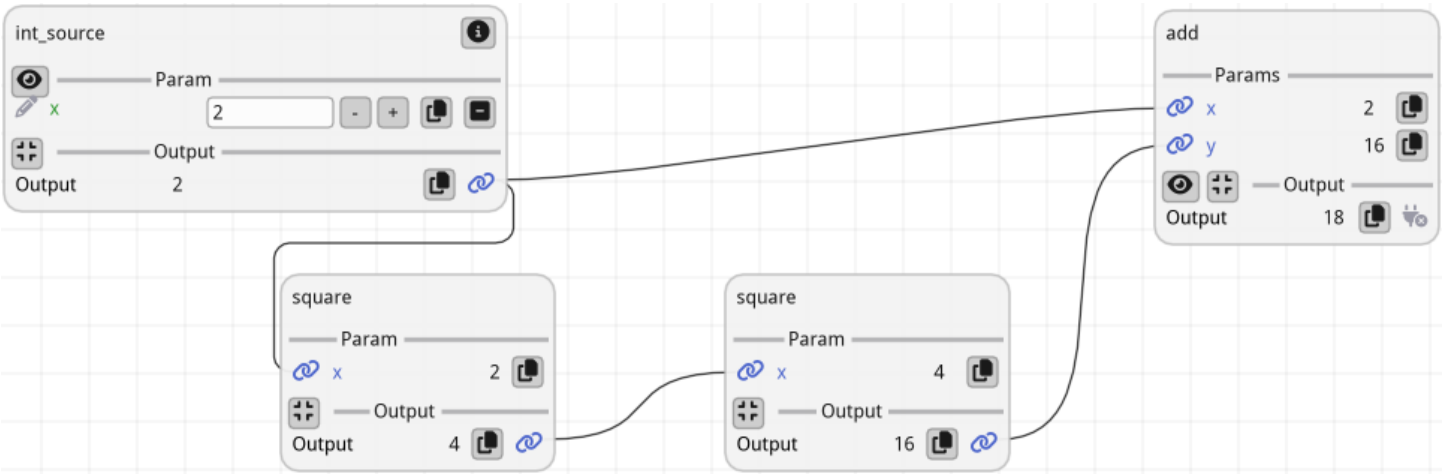
```python
import fiatlight as fl
def f(x: int) -> int:
    return x + 1
def g(x: int) -> int:
    return x * 2
fl.run([f, g], app_name="Chained functions")
```




## When you need to create a FunctionsGraph

For more complex cases, you can create a FunctionsGraph manually. This allows you to precisely control the links between the functions.

```python
import fiatlight as fl

def int_source(x : int) -> int:
    """This function will be the entry point of the graph
    Since its inputs is unlinked, fiatlight will ask the user for a value for x
    """
    return x

def square(x: int) -> int:
    return x * x

def add(x: int, y: int) -> int:
    return x + y

# 1. Create the graph
#
#    Notes:
#      - in this example we add the function `square` *two times*!
#          Each of them will have a different *unique name*: "square_1" and "square
#      - instead of creating a graph from a function composition, we could also cre
#        and add the functions manually, like show in the comment below:
#             graph = fl.FunctionsGraph.create_empty()
#             graph.add_function_composition([int_source, square, square])
#
graph = fl.FunctionsGraph.from_function_composition([int_source, square, square])


# 2. Manually add a function
graph.add_function(add)

# 3. And link it
# First, link the output of int_source to the "x" input of add
# Note: we could also specify the source output index: src_output_idx=0 (but this i
graph.add_link("int_source", "add", dst_input_name="x")

# Then, link the output of the second `square` to the "y" input of add
graph.add_link("square_2", "add", dst_input_name="y")


# 4. Run the graph
fl.run(graph, app_name="Manual graph")
```




# Validate inputs with Fiatlight

## Introduction

Validators are functions that check the validity of a parameter value and raise a ValueError (with a nice error message), or correct the value if it is not valid. They are a powerful tool to ensure that the user enters valid values for the function parameters.

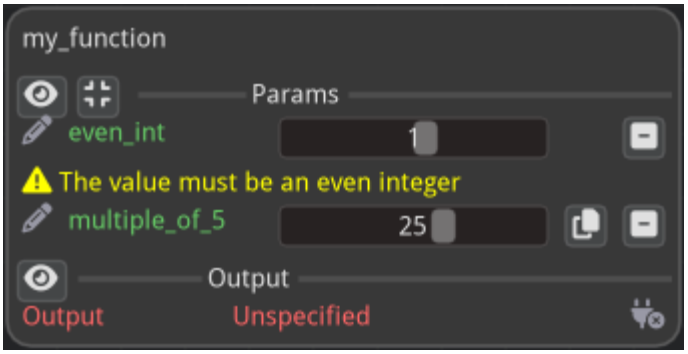
## Example: Validators for function parameters

The code below will produce a GUI where:

- The `even_int` parameter must be an even integer. If it is not, the user will see a warning.
- The `multiple_of_5` parameter will automatically correct the input to the nearest multiple of 5.

This enhances user experience by providing immediate feedback and corrections, making the application more robust and user-friendly.

```python
import fiatlight as fl

def even_int_validator(x: int) -> int:
    """This validator checks that the value is an even integer, and warns the user
    if x % 2 != 0:
        raise ValueError("The value must be an even integer")
    return x


def multiple_of_5_validator(x: int) -> int:
    """This validator will correct the user input to the closest multiple of 5."""
    return int(x / 5) * 5


def my_function(even_int: int = 0, multiple_of_5: int = 0) -> int:
    return even_int + multiple_of_5


fl.add_fiat_attributes(
    my_function,
    even_int__validator=even_int_validator,
    even_int__range=(-10, 10),
    multiple_of_5__validator=multiple_of_5_validator,
    multiple_of_5__range=(-100, 100)
)

fl.run(my_function, app_name="Validators")
```

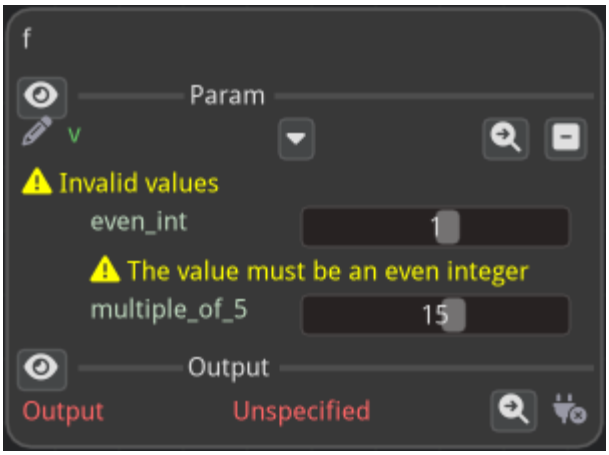



> *Note: instead of using `fl.add_fiat_attributes`, you can also use the `@fl.with_fiat_attributes` decorator on top of the function to register its validators.*

## Example: Validators for Dataclass members

The code below will produce the same GUI as the previous example, but this time using a dataclass.

```python
import fiatlight as fl
from dataclasses import dataclass  # optional, since fiatlight will add the @datacl
                                   # when using the @fl.dataclass_with_gui_registra

def even_int_validator(x: int) -> int:
    """This validator checks that the value is an even integer, and warns the user
    if x % 2 != 0:
        raise ValueError("The value must be an even integer")
    return x


def multiple_of_5_validator(x: int) -> int:
    """This validator will correct the user input to the closest multiple of 5."""
    return int(x / 5) * 5


# Note: the decorator @fl.dataclass_with_gui_registration will also apply
#       the @dataclass decorator to the class
@fl.dataclass_with_gui_registration(
    even_int__validator=even_int_validator,
    even_int__range=(-10, 10),
    multiple_of_5__validator=multiple_of_5_validator,
    multiple_of_5__range=(-100, 100)
)
class MyData:
    even_int: int = 0
    multiple_of_5: int = 0


def f(v: MyData) -> MyData:
  return v

fl.run(f, app_name="Validators in a Dataclass")
```




> *Note: instead of using the decorator `@fl.dataclass_with_gui_registration` on top of the dataclass, you can also use the `the function` fl.register_dataclass ` to register the dataclass,*

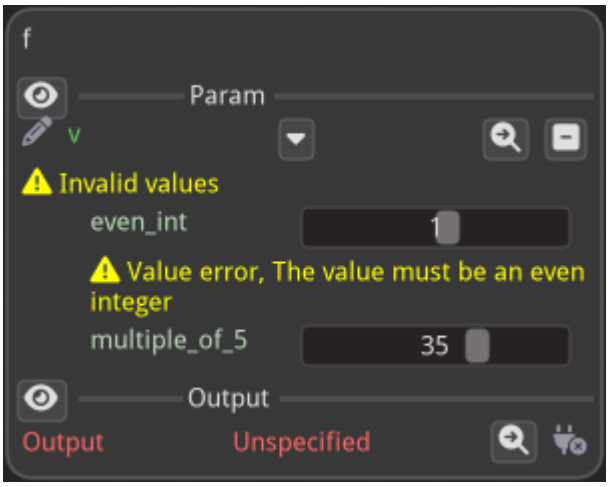
> *and add fiat attributes, such as the validators.*

## Example: Validators for BaseModel members

The code below will produce the same GUI as the previous example, but this time using a Pydantic model. In this case we can also use standard Pydantic validators.

> Note: Fiatlight will also interpret the range from the less than (le) and greater than (ge) constraints in the Pydantic model.

```python
import fiatlight as fl
from pydantic import BaseModel, Field, field_validator

@fl.base_model_with_gui_registration()
class MyData(BaseModel):
  even_int: int = Field(0, ge=-10, le=10)
  multiple_of_5: int = Field(0, ge=-100, le=100)

  @field_validator("even_int")
  def even_int_validator(cls, v):
    if v % 2 != 0:
      raise ValueError("The value must be an even integer")
    return v

  @field_validator("multiple_of_5")
  def multiple_of_5_validator(cls, v):
    return int(v / 5) * 5

def f(v: MyData) -> MyData:
  return v

fl.run(f, app_name="Validators in a Pydantic model")
```




> *Note: instead of using the decorator* `@fl.base_model_with_gui_registration` *on top of the Pydantic model, you can also use the* `the function` *fl.register_base_model ` to register the Pydantic model, and add fiat attributes, such as the validators.*

# Dataclasses & Pydantic Models

Dataclasses and Pydantic models can easily be registered with their GUI.

## Dataclasses

### Example: automatically create a GUI for a “Person” dataclass

To create a GUI for a dataclass, you first need to register the dataclass with its GUI.

For this, you can use `fl.register_dataclass(dataclass_type, **fiat_attributes)` or the `@fl.dataclass_with_gui_registration(**fiat_attributes)` decorator.

In either case, you can specify GUI options for the fields using the fiat_attributes mechanism.

**Option 1: using `register_dataclass`:**

```python
import fiatlight as fl
from dataclasses import dataclass

class Person:
    name: str
    age: int

fl.register_dataclass(Person, age__range=(0, 120))
```

**Option 2: using the decorator `dataclass_with_gui_registration`:**

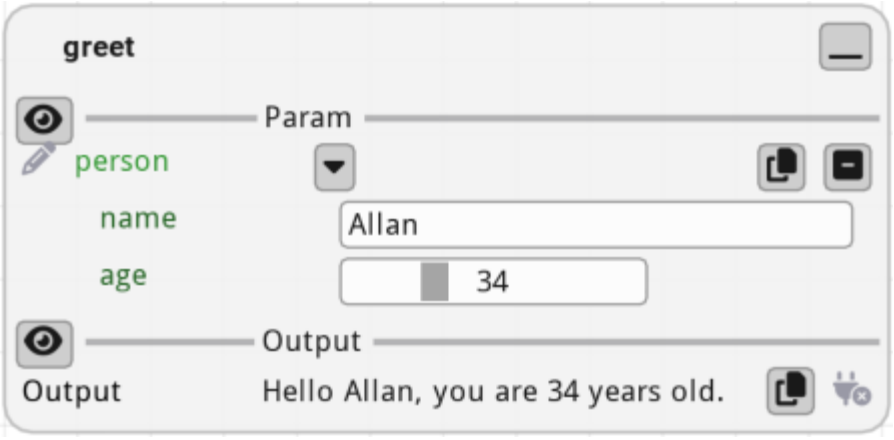
```python
import fiatlight as fl
from dataclasses import dataclass

@fl.dataclass_with_gui_registration(age__range=(0, 120))
class Person:
    name: str
    age: int
```

(This option is shorter, but more intrusive, as it modifies the original class definition.)

## Use the generated GUI in Fiatlight

You can use the dataclass as a function parameter, and fiatlight will generate a GUI for it.

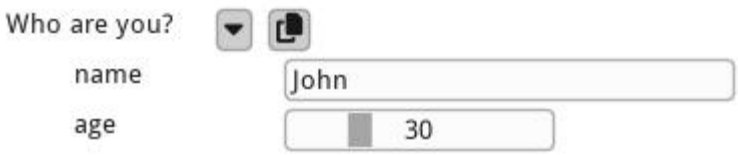
```python
def greet(person: Person) -> str:
    return f"Hello {person.name}, you are {person.age} years old."

# Note: this app *will not* remember the values of the dataclass fields between run
fl.run(greet, app_name="Dataclass Person")
```



### Or use the generate GUI in standalone application

Alternatively, you can use the generated GUI in a standalone application. Below we run an application using hello_imgui. For more info, see Dear ImGui Bundle doc.

```
from imgui_bundle import hello_imgui

PERSON = Person(name="John", age=30)

def gui():
    global PERSON
    _changed, PERSON = fl.immediate_edit("Who are you?", PERSON)

hello_imgui.run(gui)
```




## Pydantic models

Below is a more complete example of how to use Pydantic models with fiatlight.

### Example: automatically create a GUI for nested Pydantic models

**Notes:**

- fiatlight will automatically generate a serialization/deserialization mechanism for Pydantic models, so that user entries can be saved and reloaded (when used as function parameters). This is not available for dataclasses.
- Pydantic models can be nested: in the example below, you will see 3 levels of nesting, and fiatlight will generate a nice GUI for those.
- You can use `fl.register_base_model` to register a Pydantic model with its GUI and add fiat attributes Alternatively, you can use the `@fl.base_model_with_gui_registration` decorator (but this is more intrusive, as it modifies the original class definition).
- You can specify GUI options for the fields using the fiat_attributes mechanism.

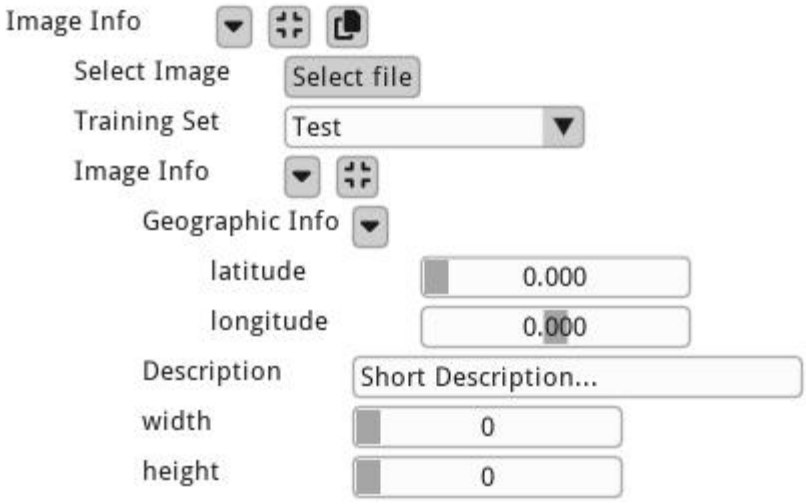
- Pydantic field validators (such as `Field(ge=0, le=90, ...)`) are supported and will be reflected in the GUI.
- Custom validators can be used, as shown in the example below.
- Validation errors will be displayed in the GUI (in yellow)

```python
import fiatlight as fl
from enum import Enum
from pydantic import BaseModel, Field


# An Enum which will be associated to a Gui automatically
class TrainingDataType(Enum):
    Test = "test"
    Train = "train"
    Validation = "validation"


# GeographicInfo: a pydantic model, with validation on latitude and longitude
# which will be reflected in the GUI
class GeographicInfo(BaseModel):
    latitude: float = Field(ge=0, le=90, default=0)
    longitude: float = Field(ge=-180, lt=180, default=0)

# We register the GeographicInfo model with its GUI
# (the sliders for lon/lat will be limited to the ranges specified in the Fields)
fl.register_base_model(GeographicInfo)


# A custom validator, which will be used to validate the short description
def validate_short_description(value: str) -> str:
    if len(value) > 30:
        raise ValueError("Description is too long")
    return value


# A second model, which nests the first one (GeographicInfo)
class ImageInfo(BaseModel):
    geo_info: GeographicInfo = GeographicInfo()
    description: str = "Short Description..."
    width: int = 0
    height: int = 0

# We register the ImageInfo model with its GUI, and add some fiat attributes
# Also, we add a custom Fiatlight validator for the description field
fl.register_base_model(
    ImageInfo,
    width__range=(0, 2000),
    height__range=(0, 2000),
    description__label="Description",
    description__validator=validate_short_description,
    geo_info__label="Geographic Info",
)



# A third model, which nests the second one (ImageInfo)
# In total, it has 3 levels: TrainingImage -> ImageInfo -> GeographicInfo
# In this case, we use the decorator to register the model with its GUI
```

```python
@fl.base_model_with_gui_registration(
    image_path__label="Select Image",
    training_type__label="Training Set",
    info__label="Image Info",
)
class TrainingImage(BaseModel):
    image_path: fl.fiat_types.ImagePath = ""  # type: ignore
    training_type: TrainingDataType = TrainingDataType.Test
    info: ImageInfo = ImageInfo(width=0, height=0)
```

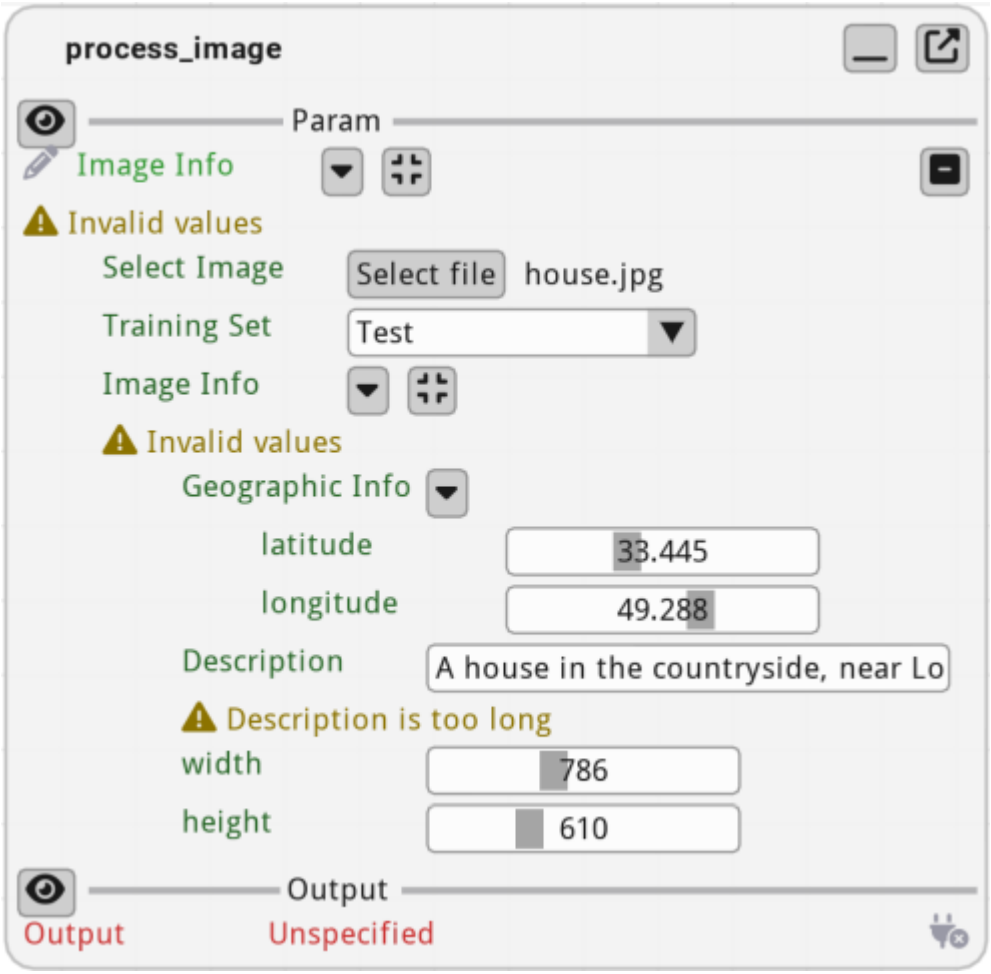
## Use the generated GUI in a standalone application

```python
from imgui_bundle import hello_imgui

# We create an instance of the Pydantic model
IMAGE = TrainingImage()

def gui():
    global IMAGE
    _changed, IMAGE = fl.immediate_edit("Image Info", IMAGE)

hello_imgui.run(gui)
```




## Or use the generated GUI in Fiatlight

```python
def process_image(image: TrainingImage) -> str:
    import os
    basename = os.path.basename(image.image_path)
    return basename

# Note: this app *will* remember the values of the Pydantic model fields between ru
fl.run(process_image, app_name="Pydantic Image Model")
```




In the previous screenshot, the GUI generated by `fl.base_model_with_gui_registration` will automatically validate the data according to the model's constraints, and thus display an error message (because in this case, the description is too long).

# Video Tutorial

A short video tutorial is available for this topic.

🔗 Watch Video

# Fully customize a Function Gui

## Introduction

By subclassing `FunctionWithGui`, you can fully customize the behavior of the function:

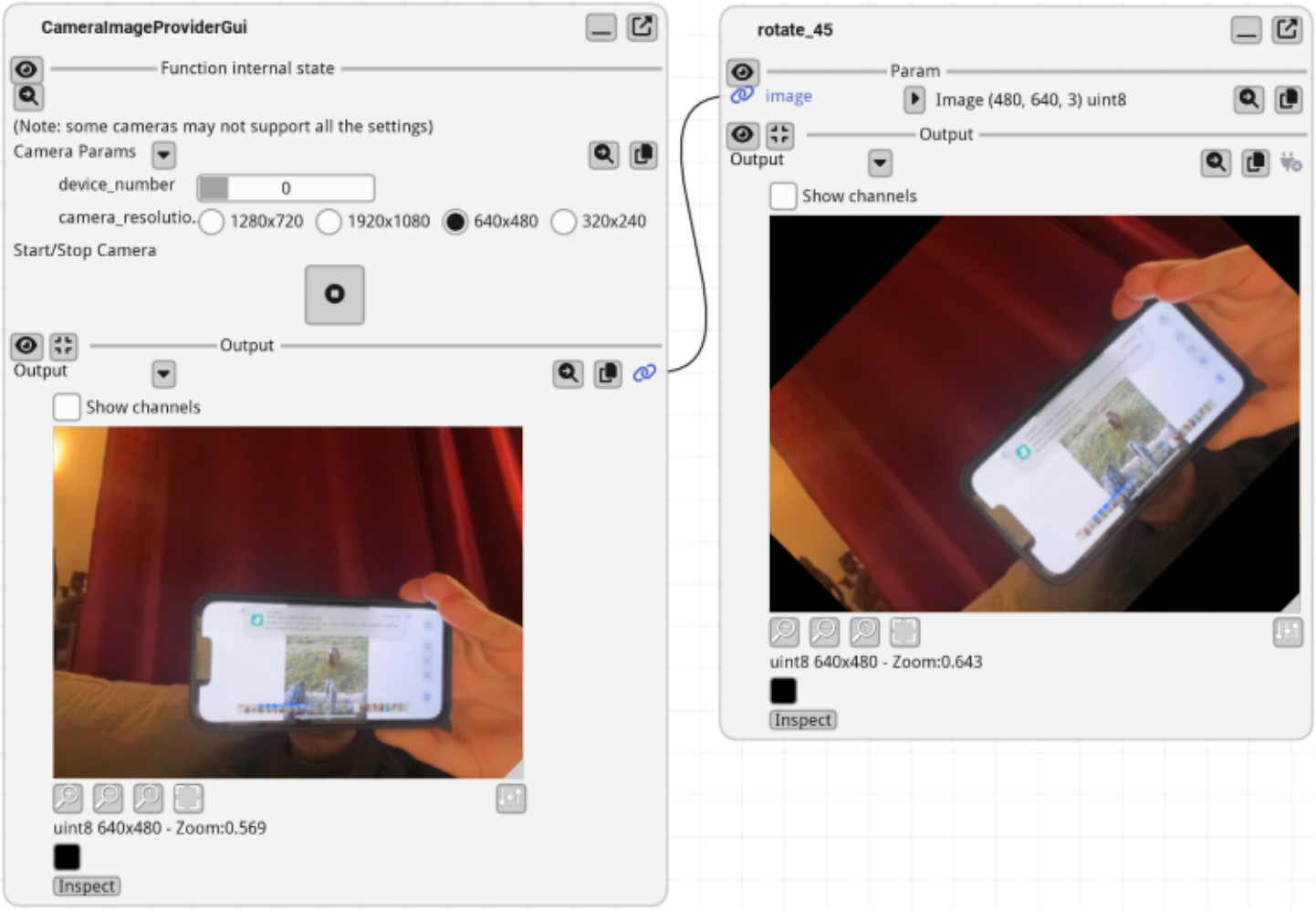
- you can add a GUI for the internal state of the function (e.g. displaying a live plot of a sound signal)
- you can add a heartbeat function that will be called at each frame (e.g. get the latest data from a sensor)
- you can save and load the internal GUI presentation options to/from a JSON file (e.g. to save the layout of a plot)

## Example: Camera & Internal State

fiatlight.fiat_kits.fiat_image.CameraImageProviderGui is a good example of a custom FunctionWithGui class.

You can see it in action below:

```
import fiatlight as fl
from fiatlight.fiat_kits.fiat_image import CameraImageProviderGui, ImageU8_3
import cv2

def rotate_45(image: ImageU8_3) -> ImageU8_3:
    transform = cv2.getRotationMatrix2D((image.shape[1] / 2, image.shape[0] / 2), 4
    return cv2.warpAffine(image, transform, (image.shape[1], image.shape[0]))  # ty

camera_provider_gui = CameraImageProviderGui()
fl.run([camera_provider_gui, rotate_45], app_name="Camera provider with rotation")
```




**Commented extracts of [camera_image_provider.py]**

Look at the `CameraImageProviderGui` class that extends `FunctionWithGui`:

> Notes:
>
> - `CameraImageProviderGui` der uses a `CameraImageProvider` class that provides images from a camera.
> - `CameraParams` contains the state of the camera (device number, brightness, contrast, etc.). This state is serialized. As it is as a Pydantic model, a GUI for it is automatically created by Fiatlight, and its state is serialized.

**CameraImageProviderGui (a descendant of FunctionWithGui)**:

```
from fiatlight.fiat_notebook import look_at_code
%look_at_python_code
fiatlight.fiat_kits.fiat_image.camera_image_provider.CameraImageProviderGui
```

**CameraParams (serialized internal state)**:

```
import fiatlight as fl
from enum import Enum
from pydantic import BaseModel
import cv2

class CameraResolution(Enum):
    HD_1280_720 = [1280, 720]
    FULL_HD_1920_1080 = [1920, 1080]
    VGA_640_480 = [640, 480]

@fl.base_model_with_gui_registration(device_number__range= (0, 5), brightness__rang
class CameraParams(BaseModel):
    device_number: int = 0
    brightness: float = 0.5
    contrast: float = 0.5
    camera_resolution: CameraResolution = CameraResolution.VGA_640_480


class CameraImageProvider:
    '''A class that provides images from a camera'''
    camera_params: CameraParams
    cv_cap: cv2.VideoCapture | None = None
    ...
```

# Custom types registration

By calling `fiatlight.register_type(DataType, DataTypeWithGui)`, it is possible to register a custom type with its GUI.

For a given type's GUI, it is possible to customize many aspects. Basically all the callbacks and options inside AnyDataGuiCallbacks can be customized.

## Example 1: a customizable Normal Distribution type

**Step 1: Define the Custom Type**

First, let's define a new type called NormalDistribution.

```
class NormalDistribution:
    mean: float = 0.0
    stddev: float = 1.0

    def __init__(self, mean: float = 0.0, stddev: float = 1.0) -> None:
        self.mean = mean
        self.stddev = stddev
```

**Step 2: Create a Class to Handle the Custom Type**

Next, we create a class NormalDistributionWithGui that extends AnyDataWithGui and defines the custom presentation and editing logic for the NormalDistribution type.

It will handle:

- A custom GUI for editing the NormalDistribution type
- A custom GUI for presenting the NormalDistribution type (using a cached figure, which is updated when the distribution changes)
- Serialization and deserialization of the custom type
- A default value provider

```python
import fiatlight as fl
from imgui_bundle import imgui, imgui_fig
import matplotlib.pyplot as plt
from matplotlib.figure import Figure
import numpy as np


class NormalDistributionWithGui(fl.AnyDataWithGui[NormalDistribution]):
    # Cached figure for the distribution plot
    figure: Figure | None = None
    # boolean to indicate if the figure image should be refreshed
    shall_refresh_figure_image: bool = True

    def __init__(self) -> None:
        super().__init__(NormalDistribution)

        # Edit and present callbacks
        self.callbacks.edit = self._edit_gui
        self.callbacks.present = self._present_gui
        self.callbacks.present_str = lambda value: f"Normal Distrib: Mean={value.me

        # Default value provider
        self.callbacks.default_value_provider = lambda: NormalDistribution()

        # Serialization of the custom type
        # (note it would be automatic if we used a Pydantic model)
        self.callbacks.save_to_dict = lambda value: {"mean": value.mean, "stddev":
        self.callbacks.load_from_dict = lambda data: NormalDistribution(mean=data["

        # Callback for handling changes: we need to subscribe to this event
        # in order to update the self.figure when the distribution changes
        self.callbacks.on_change = self._on_change

    def _on_change(self, value: NormalDistribution) -> None:
        # remember to close the previous figure to avoid memory leaks
        if self.figure is not None:
            plt.close(self.figure)

        # Create the figure
        x = np.linspace(value.mean - 4 * value.stddev, value.mean + 4 * value.stdde
        y = (1 / (value.stddev * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - value.m
        figure = plt.figure(figsize=(4, 3))
        plt.plot(x, y)
        plt.title("Normal Distribution")
        plt.xlabel("x")
        plt.ylabel("Density")
        plt.grid(True)

        # Cache the figure
        self.figure = figure

        # Indicate that the figure image should be refreshed
        self.shall_refresh_figure_image = True
```

```
    @staticmethod
    def _edit_gui(value: NormalDistribution) -> tuple[bool, NormalDistribution]:
        # Note: we receive the current value and return a tuple with
        # a boolean indicating if the value was modified
        modified = False
        imgui.text("Edit Normal Distribution:")
        imgui.set_next_item_width(100)
        changed, new_mean = imgui.slider_float("Mean", value.mean, -10.0, 10.0)
        if changed:
            value.mean = new_mean
            modified = True
        imgui.set_next_item_width(100)
        changed, new_stddev = imgui.slider_float("StdDev", value.stddev, 0.1, 10.0)
        if changed:
            value.stddev = new_stddev
            modified = True

        return modified, value

    def _present_gui(self, _value: NormalDistribution) -> None:
        # We do not use the value which was passed as a parameter as we use the cac
        # which was updated in the _on_change callback
        imgui_fig.fig("Normal Distribution", self.figure, refresh_image=self.shall_
        self.shall_refresh = False
```

**Step 3: Register the type**

Finally, we register the custom type with its GUI, simply by calling the register_type function.

```
fl.register_type(NormalDistribution, NormalDistributionWithGui)
```

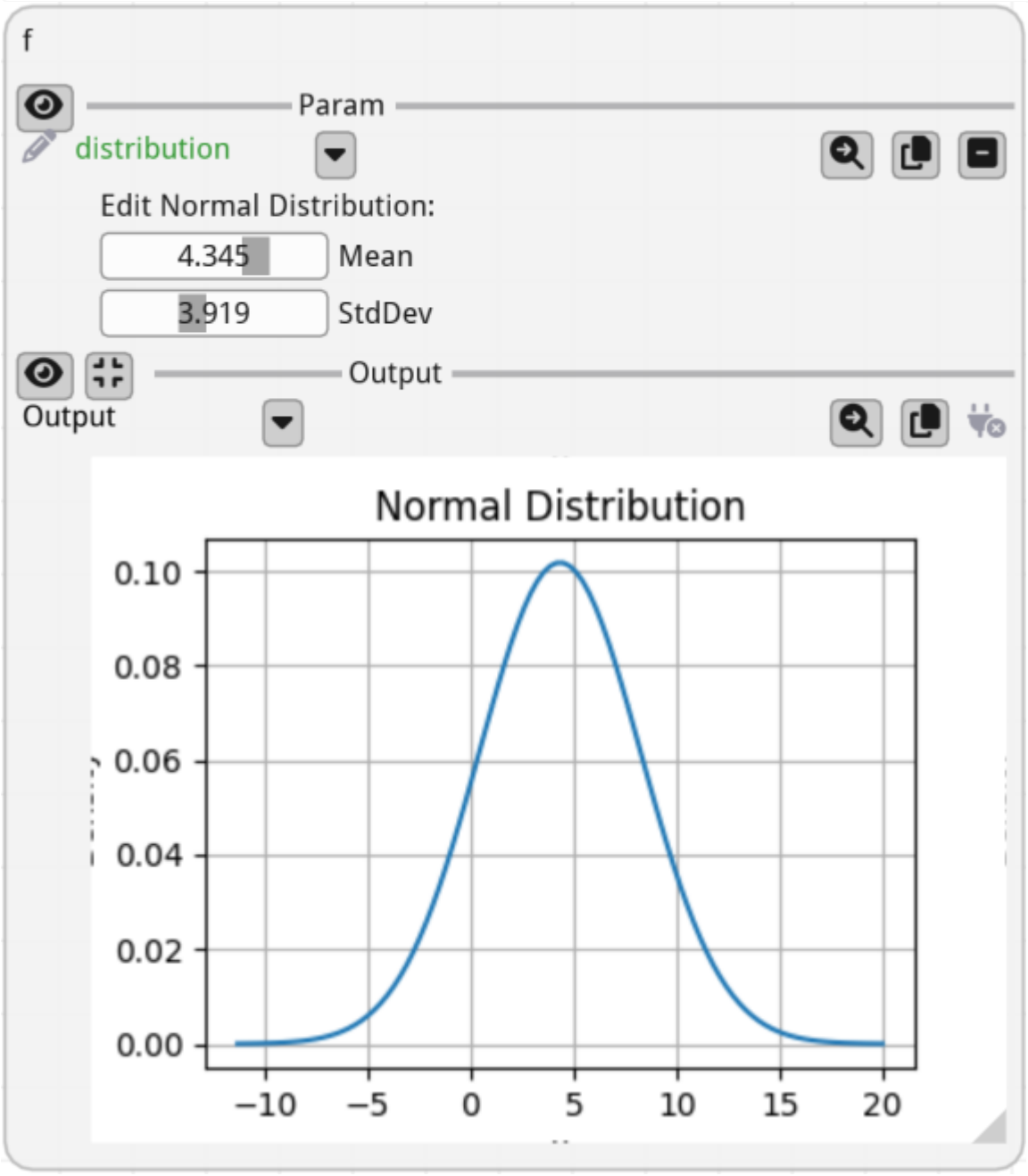
From now on, the NormalDistribution type will be associated with the NormalDistributionWithGui GUI: any function that uses NormalDistribution as a parameter or as a return type will automatically have a GUI for editing and presenting the NormalDistribution type.

**Step 4: Use the custom type in a function**

In this example, our function simply returns the NormalDistribution instance that was passed to it. In the screenshot, you can see the “edit” callback in action in the Param edition section, and the “present” callback in the Output section.

```python
def f(distribution: NormalDistribution) -> NormalDistribution:
    return distribution

fl.run(f, app_name="Normal Distribution")
```

```
2024-07-04 23:40:32.252 Python[68193:11625526] ApplePersistenceIgnoreState: Existing
```

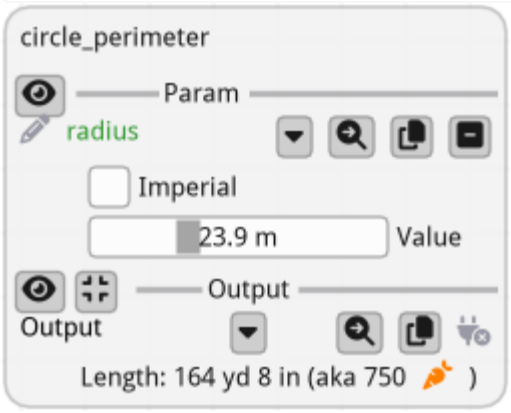


Example 2: a Length type with imperial units

```python
# Step 1: Define the custom type for which we want to create a GUI
# ================================================================
from typing import NewType

Length = NewType("Length", float)

# Step 2: Create a class to handle the custom type
# ================================================
import fiatlight
from fiatlight import AnyDataWithGui
from fiatlight.fiat_widgets import fontawesome_6_ctx, icons_fontawesome_6
from typing import NewType, Any, Dict
from imgui_bundle import imgui, hello_imgui, imgui_ctx, ImVec4


# The specific GUI for our custom type
class LengthWithGui(AnyDataWithGui[Length]):
    use_imperial_units: bool = False

    def __init__(self) -> None:
        super().__init__(Length)
        self.callbacks.edit = self._edit  # A custom callback for editing the data
        self.callbacks.present = self._present  # A custom callback for presenting
        self.callbacks.present_str = self._present_str  # A custom callback for pre
        self.callbacks.default_value_provider = lambda: Length(1.0)  # A custom cal
        # custom callback for saving the GUI options (here, we save the imperial un
        self.callbacks.save_gui_options_to_json = self._save_gui_options_to_json
        self.callbacks.load_gui_options_from_json = self._load_gui_options_from_jso

    def _edit(self, value: Length) -> tuple[bool, Length]:
        _, self.use_imperial_units = imgui.checkbox("Imperial", self.use_imperial_u

        format = "%.3g m" if not self.use_imperial_units else "%.3g yd"
        value_unit = value * 1.09361 if self.use_imperial_units else value
        imgui.set_next_item_width(hello_imgui.em_size(10))
        changed, new_value_unit = imgui.slider_float(
            "Value", value_unit, 1e-5, 1e11, format, imgui.SliderFlags_.logarithmic
        )
        if changed:
            value = Length(new_value_unit / 1.09361 if self.use_imperial_units else
        return changed, value

    @staticmethod
    def _present_str(value: Length) -> str:
        return f"Length: {value:.2f} m"

    @staticmethod
    def _present(value: Length) -> None:
        with fontawesome_6_ctx():
            yd = int(Length(value * 1.09361))
            inches = int((Length(value * 1.09361 - yd) * 36))
            bananas = int(value / 0.2)
            imgui.text(f"Length: {yd} yd {inches:.0f} in (aka {bananas}")
```

```
            imgui.same_line()
            with imgui_ctx.push_style_color(imgui.Col_.text.value, ImVec4(1, 0.5, 0
                imgui.text(icons_fontawesome_6.ICON_FA_CARROT)
            imgui.same_line()
            imgui.text(")")

    def _save_gui_options_to_json(self) -> Dict[str, Any]:
        return {"use_imperial_units": self.use_imperial_units}

    def _load_gui_options_from_json(self, json: Dict[str, Any]) -> None:
        self.use_imperial_units = json.get("use_imperial_units", False)


# Step 3: Register the custom type with its GUI
# =============================================
from fiatlight import register_type

register_type(Length, LengthWithGui)


# Step 4: Use the custom type in a function
# =========================================
# A function that uses our custom type
def circle_perimeter(radius: Length) -> Length:
    return Length(2 * 3.14159 * radius)


# Run the function with the GUI
fiatlight.run(circle_perimeter, app_name="Circle Perimeter in banana units")
```

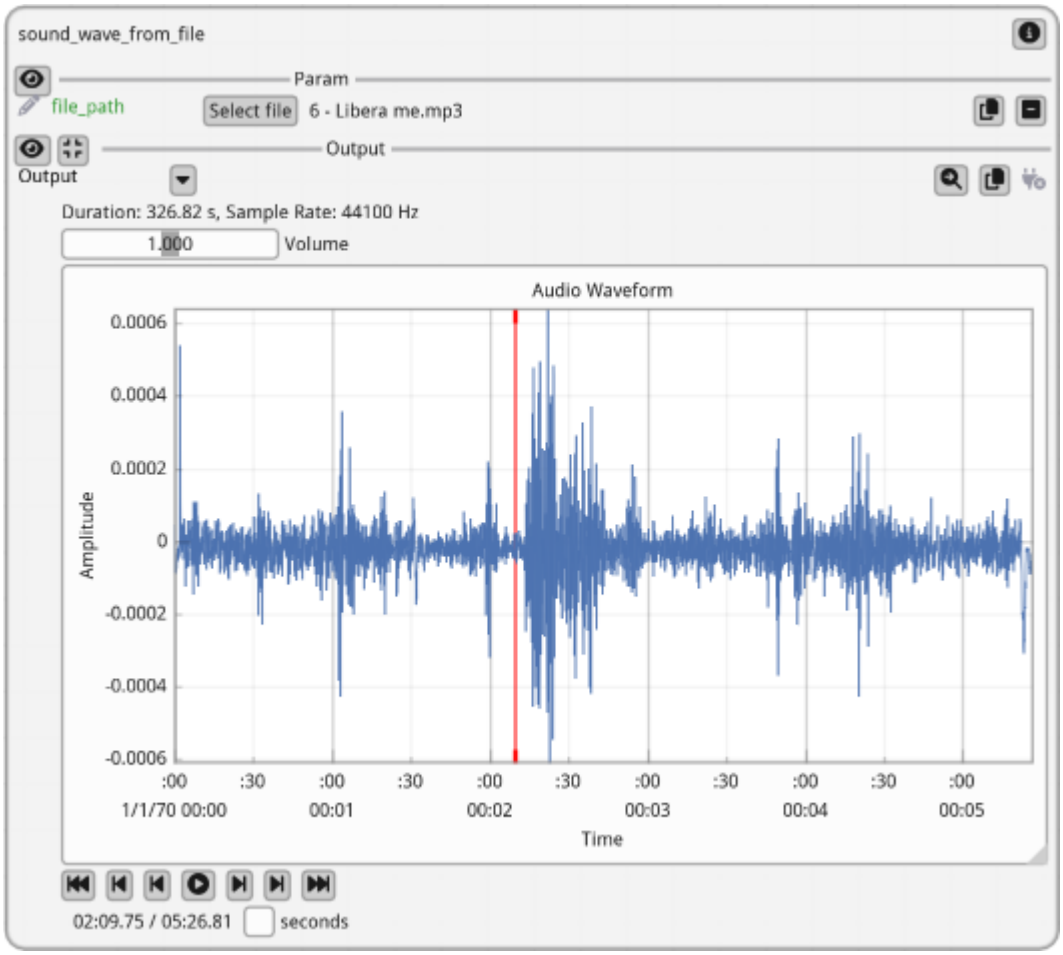
```
2024-07-04 23:40:37.134 Python[68193:11625526] WARNING: Secure coding is not enable
```




## Example 3: a sound player

The sound wave player also uses a custom type with a GUI.

```
from fiatlight.fiat_kits.experimental.fiat_audio_simple.demos import demo_sound_wav

sound_wave_player_gui_demo.main()
```




For more info, see its source code).

# How to create a new “fiat kit”

## fiat_kit_skeleton

fiatlight.fiat_kits.fiat_skeleton is a starting point for creating new widgets: it is a minimalistic kit that contains the necessary files to create a new widget.

```
fiat_kit_skeleton
├── __init__.py
├── mydata.py                       # An example data or library that you want to pr
├── mydata_presenter.py             # The presenter of the data
|                                   # Also contains a derivate of PossibleCustomAttr
|                                   # where all the custom attributes are defined
|
└── mydata_with_gui.py              # MyDataWithGui: the widget that will be displaye
# (inherits from AnyDataWithGui, implements all the callbacks
#  of AnyDataGuiCallbacks, and uses MyDataPresenter for
# complex data presentation)
```

See files:

- mydata.py
- mydata_presenter.py
- mydata_with_gui.py

## fiat_kit_skeleton in action

fiatlight.fiat_kits.fiat_dataframe it was developed starting from the skeleton. It is a good example on how it can be customized.

```
fiat_dataframe
├── dataframe_presenter.py                  # The presenter of the data (presentati
|                                           # Also contains a derivate of PossibleCu
|
├── dataframe_with_gui.py                   # The widget that will be displayed in t
|                                           # (inherits from AnyDataWithGui, impleme
|                                          #  of AnyDataGuiCallbacks, and uses Datal
|                                          # complex data presentation)
```

See files:

- dataframe_presenter.py
- dataframe_with_gui.py

# Run the demos

**Install optional dependencies**

In order to run the demos, you may need to install per domain dependencies:

- For AI demos: `pip install -r requirements-ai.txt`
- For audio demos: `pip install -r requirements-audio.txt`

### Standard demos

Several demos are available in the src/python/fiatlight/demos folder:

```
%%bash
tree -I "__pycache__|fiat_settings|priv_experiments|fonts|__init__.py" ../demos/ |
```

```
../demos/
├── ai
│   ├── demo_sdxl_meme.py
│   └── demo_sdxl_toon_edges.py
├── audio
│   ├── demo_audio_processing_link.py
│   ├── microphone_gui_demo_link.py
│   └── sound_wave_player_gui_demo_link.py
├── custom_graph
│   └── demo_custom_graph.py
├── hello_rosetta
│   ├── example_validation.py
│   └── hello_rosetta.py
├── images
│   ├── demo_computer_vision.py
│   ├── demo_oil_paint.py
│   ├── old_school_meme.py
│   ├── opencv_wrappers.py
│   └── toon_edges.py
├── math
│   ├── demo_binomial.py
│   ├── demo_float_functions.py
│   ├── demo_plot_array.py
│   └── demo_plot_manual_present.py
├── plots
│   └── demo_matplotlib.py
└── string
    ├── demo_word_count.py
    └── str_functions.py
```

### Notebook demos

You can also run all the demos that are present in the documentation (there are a lot of interesting demos, together with screenshots)

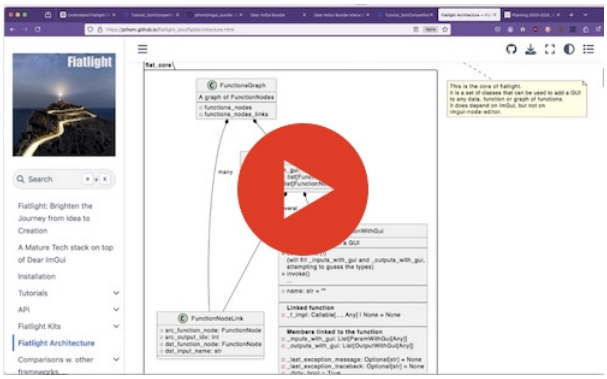
- install Jupyter: `pip install jupyter`
- Launch Jupyter with the following command: `jupyter lab`
- After Jupyter is launched, a browser page will open: navigate to the “src/python/fiatlight/doc” folder to find the demos.

# API

## High level overview video

The video below present a high-level overview of Fiatlight’s architecture and how it automatically maps functions and data types to UI components.

🔗 Watch Video




## Architecture

- Fiatlight Architecture: Overview of the Fiatlight architecture, including the class diagrams and folders structure.

## fiat_core

fiat_core is the foundational package of the fiatlight framework. It focuses on wrapping data and functions with GUI elements to facilitate interaction.

Its most important classes are:

- `FunctionWithGui`: Encapsulates a function, enriching it with a GUI based on inferred input and output types. It handles function invocation and manages internal states like exceptions and execution flags.
- `AnyDataWithGui`: Wraps any type of data with a GUI. This class manages the data value and its associated callbacks, and it provides methods to serialize/deserialize the data to/from JSON.
- `AnyDataGuiCallbacks`: Stores callback functions for AnyDataWithGui, enhancing interactivity by allowing custom widgets and presentations.
- `FunctionsGraph`: Represents a graph of functions, where the output of one function can be linked to the input of another function. It allows the user to create complex workflows by chaining functions together.

# Fiatlight Architecture

```
# Necessary imports for this doc page
from fiatlight.fiat_notebook import plantuml_magic, display_markdown_from_file
```

# Class diagrams

## fiat_core

This is the foundational package of the fiatlight framework. It focuses on wrapping data and functions with GUI elements to facilitate interaction.

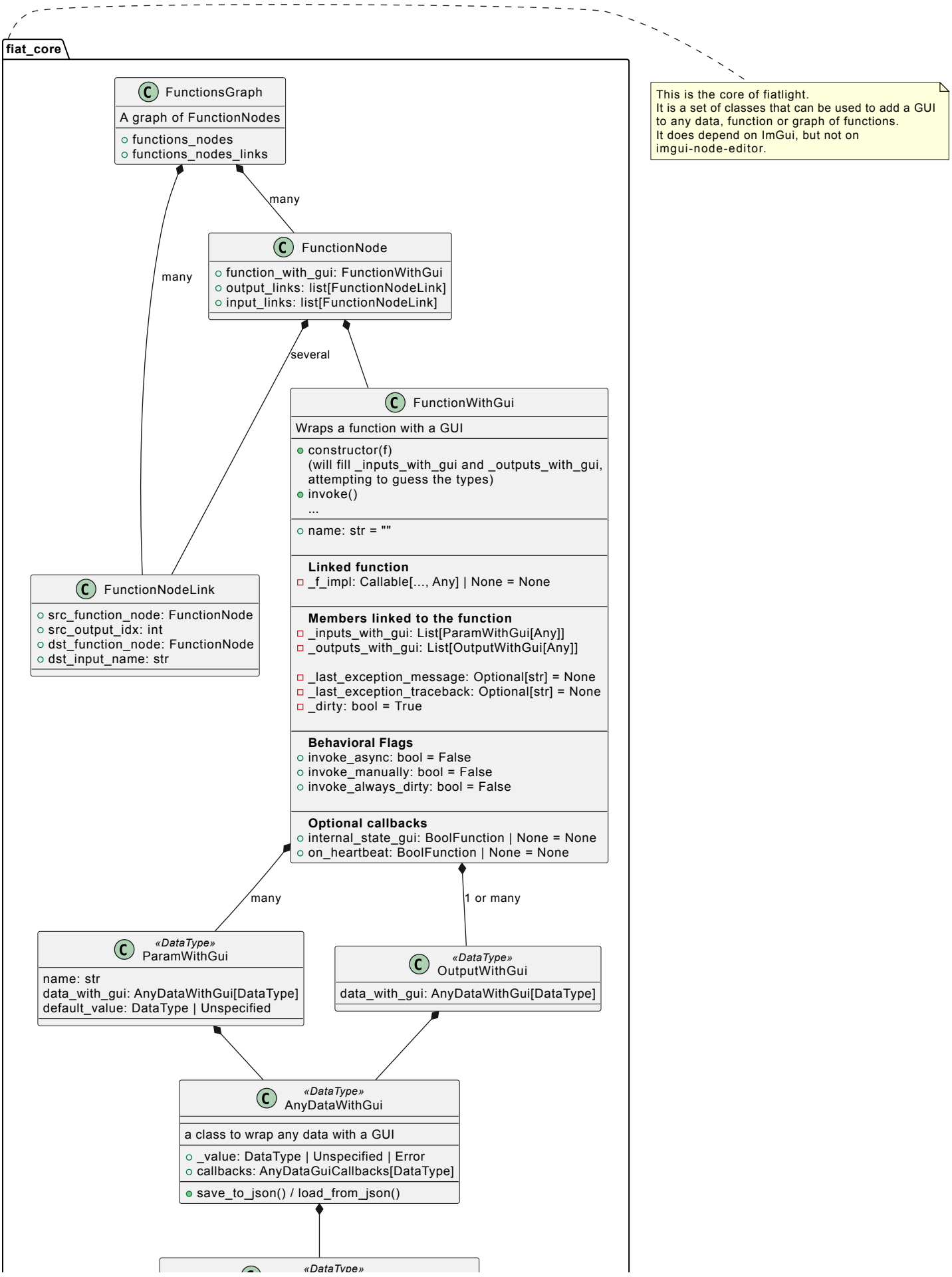
#### Classes

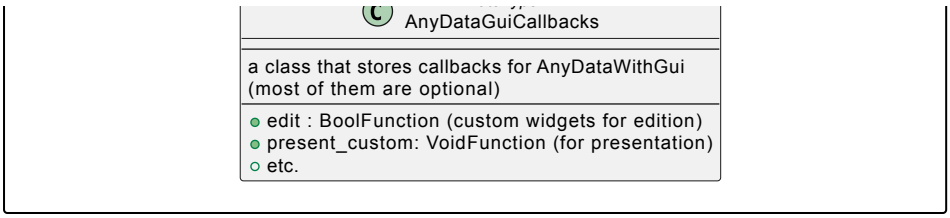
- `AnyDataWithGui`: Wraps any type of data with a GUI. This class manages the data value and its associated callbacks, and it provides methods to serialize/deserialize the data to/from JSON.
- `AnyDataGuiCallbacks`: Stores callback functions for AnyDataWithGui, enhancing interactivity by allowing custom widgets and presentations.
- `FunctionWithGui`: Encapsulates a function, enriching it with a GUI based on inferred input and output types. It handles function invocation and manages internal states like exceptions

and execution flags.

- `ParamWithGui` and `OutputWithGui`: These classes link parameters and outputs of functions to their GUI representations.
- `FunctionNode`: Represents a node in a function graph, containing links to other function nodes and managing data flow between them.
- `FunctionNodeLink`: Defines a link between outputs of one function node and inputs of another, facilitating data flow in the function graph.
- `FunctionsGraph`: Represents a graph of interconnected FunctionNode instances, effectively mapping the entire functional structure.

```
%plantuml_include class_diagrams/fiat_core.puml
```


fiat_core
This is the core of fiatlight.
It is a set of classes that can be used to add a GUI
to any data, function or graph of functions.
It does depend on ImGui, but not on
imgui-node-editor.
FunctionsGraph
A graph of FunctionNodes
functions_nodes
functions_nodes_links
many
many
FunctionNode
function_with_gui: FunctionWithGui
output_links: list[FunctionNodeLink]
input_links: list[FunctionNodeLink]
several
FunctionWithGui
Wraps a function with a GUI
constructor(f)
(will fill _inputs_with_gui and _outputs_with_gui,
attempting to guess the types)
invoke()
...
name: str = ""
Linked function
_f_impl: Callable[..., Any] | None = None
Members linked to the function
_inputs_with_gui: List[ParamWithGui[Any]]
_outputs_with_gui: List[OutputWithGui[Any]]
_last_exception_message: Optional[str] = None
_last_exception_traceback: Optional[str] = None
_dirty: bool = True
Behavioral Flags
invoke_async: bool = False
invoke_manually: bool = False
invoke_always_dirty: bool = False
Optional callbacks
internal_state_gui: BoolFunction | None = None
on_heartbeat: BoolFunction | None = None
FunctionNodeLink
src_function_node: FunctionNode
src_output_idx: int
dst_function_node: FunctionNode
dst_input_name: str
many
1 or many
«DataType»
ParamWithGui
name: str
data_with_gui: AnyDataWithGui[DataType]
default_value: DataType | Unspecified
«DataType»
OutputWithGui
data_with_gui: AnyDataWithGui[DataType]
«DataType»
AnyDataWithGui
a class to wrap any data with a GUI
_value: DataType | Unspecified | Error
callbacks: AnyDataGuiCallbacks[DataType]
save_to_json() / load_from_json()




## fiat_togui

fiat_togui provides functions to register new types (classes, dataclasses, enums) so that they are associated with a GUI.

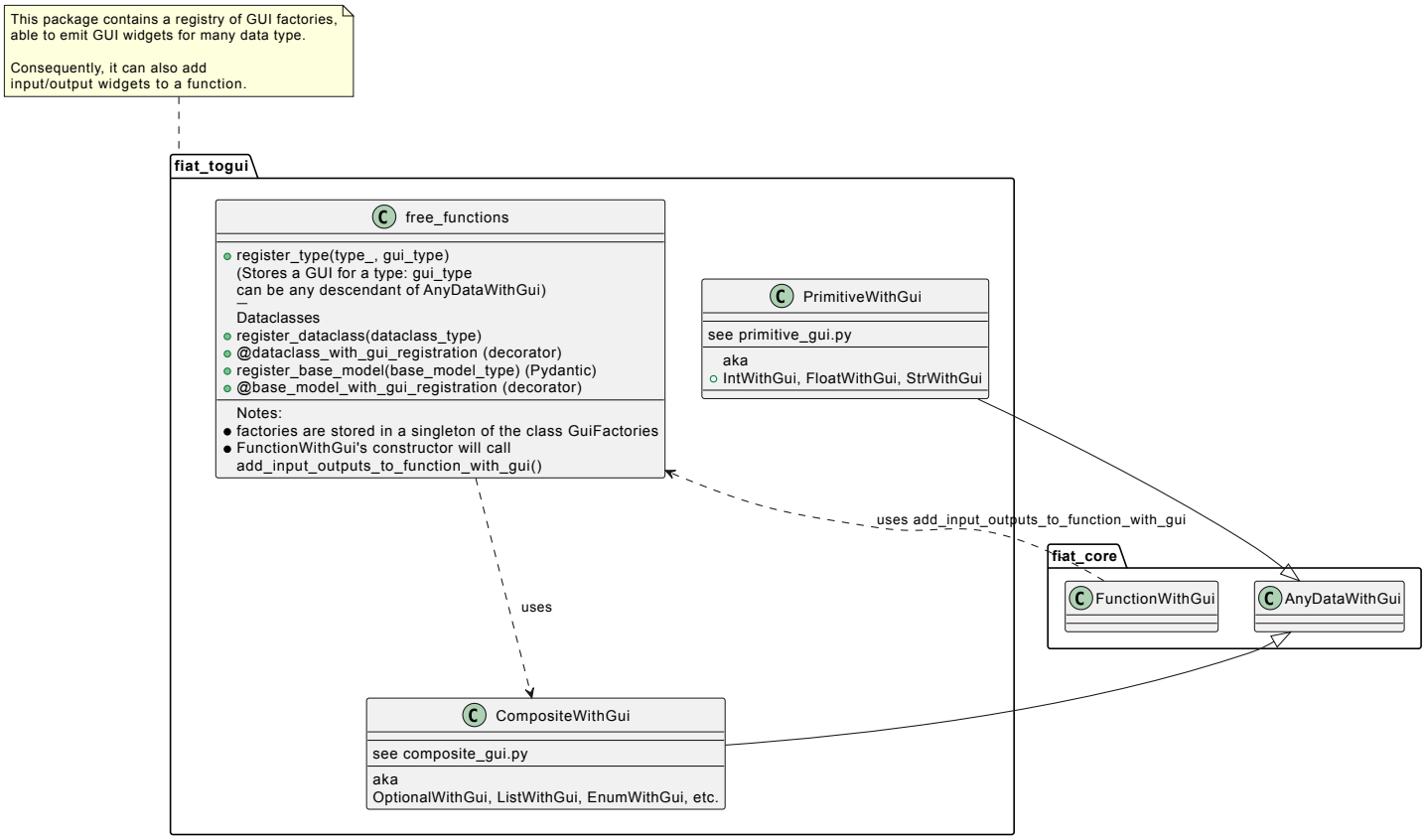
### Functions

- `register_type(type_, gui_type)`: register a GUI for a given type. gui_type must be a descendant of AnyDataWithGui
- `register_enum(enum_type)` and the equivalent decorator `enum_with_gui_registration`: register an enum
- `register_dataclass` and the equivalent decorator `dataclass_with_gui_registration`: register a dataclass base model
- `register_base_model` and the equivalent decorator `base_model_with_gui_registration`: register a pydantic base model

### Classes

- `IntWithGui`, `FloatWithGui`, etc.: provides GUI for primitive types (int, str, float, bool)
- `OptionalWithGui`: able to add GUI to Optional[DataType] (if DataType is registered)

```
%plantuml_include class_diagrams/fiat_togui.puml
```




# fiat_runner

fiat_runner is the package that contains the “run” functions:

### Free function

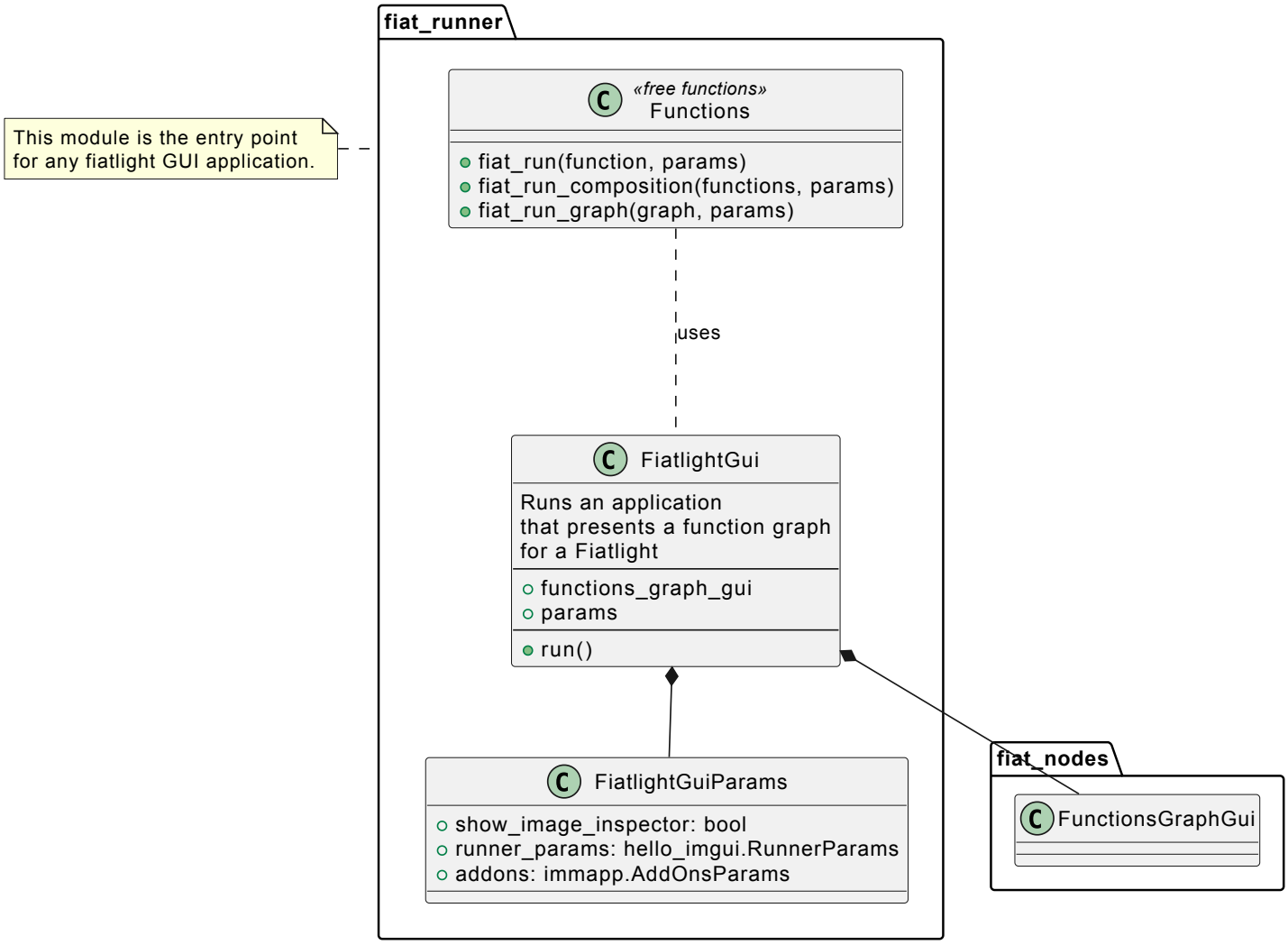
`fiat_run` accepts either a standard function, a list of functions, or a graph of functions. It executes the function(s) and displays the results in a GUI.

- `fiat_run(fn)` # fn is a function or a FunctionWithGui
- `fiat_run([fn1, fn2, ...])` # list of functions or FunctionWithGui
- `fiat_run(graph)` # A FunctionsGraph

### Classes

- `FiatlightGui`: The main runtime class that presents a GUI for interacting with a function graph. It orchestrates the execution and user interaction.
- `FiatlightGuiParams`: Stores configuration and parameters for the GUI application, such as visibility toggles and other settings.

```
%plantuml_include class_diagrams/fiat_runner.puml
```




## fiat_nodes

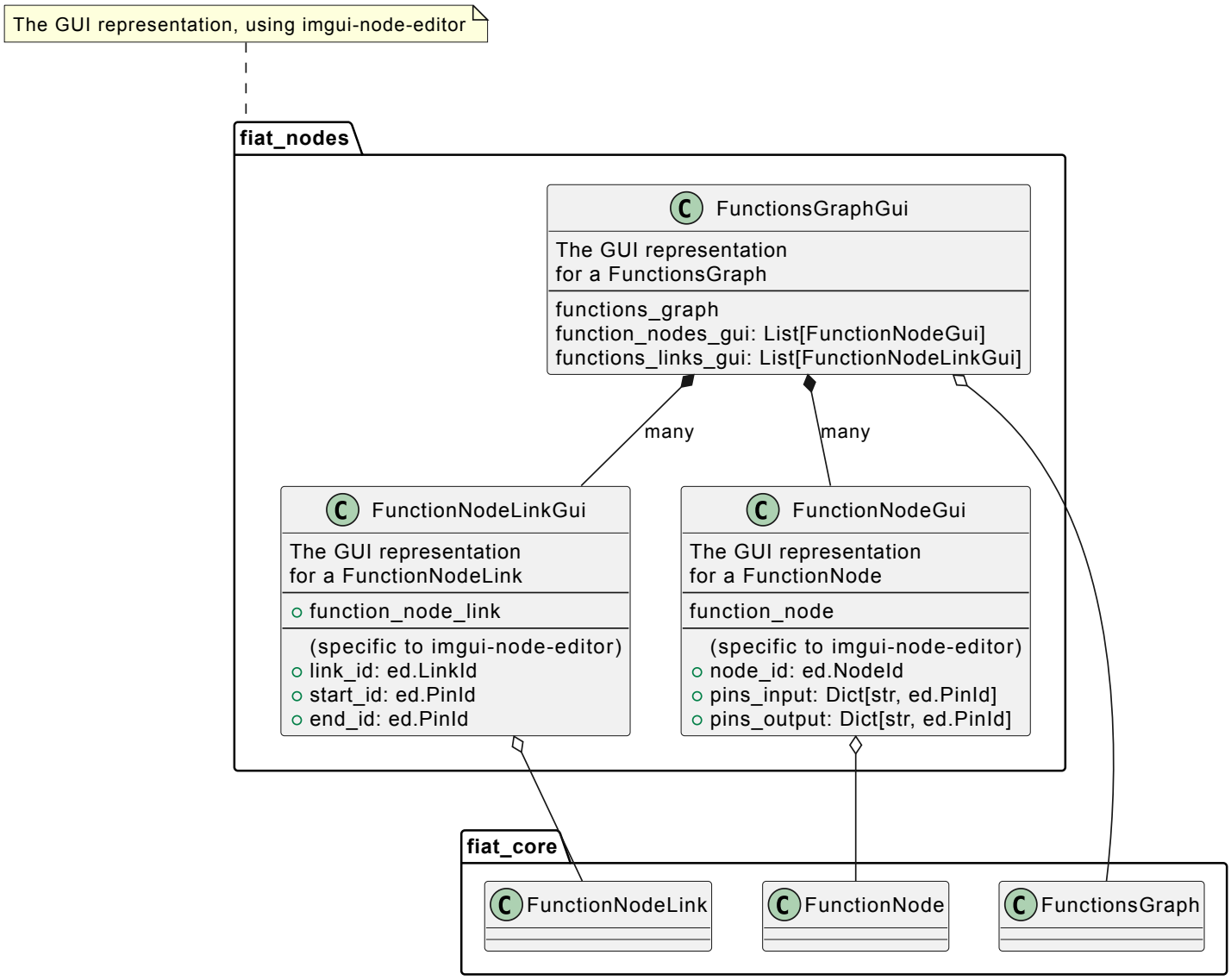
fiat_nodes is the package that is able to display a function graph in a node editor (using [imgui-node-editor])

As a final user, you will probably not interact with it.

### Classes

- `FunctionNodeGui`: The GUI representation of a FunctionNode
- `FunctionNodeLinkGui`: The GUI representation aspect of a FunctionNodeLink
- `FunctionsGraphGui`: The GUI representation of a FunctionsGraph

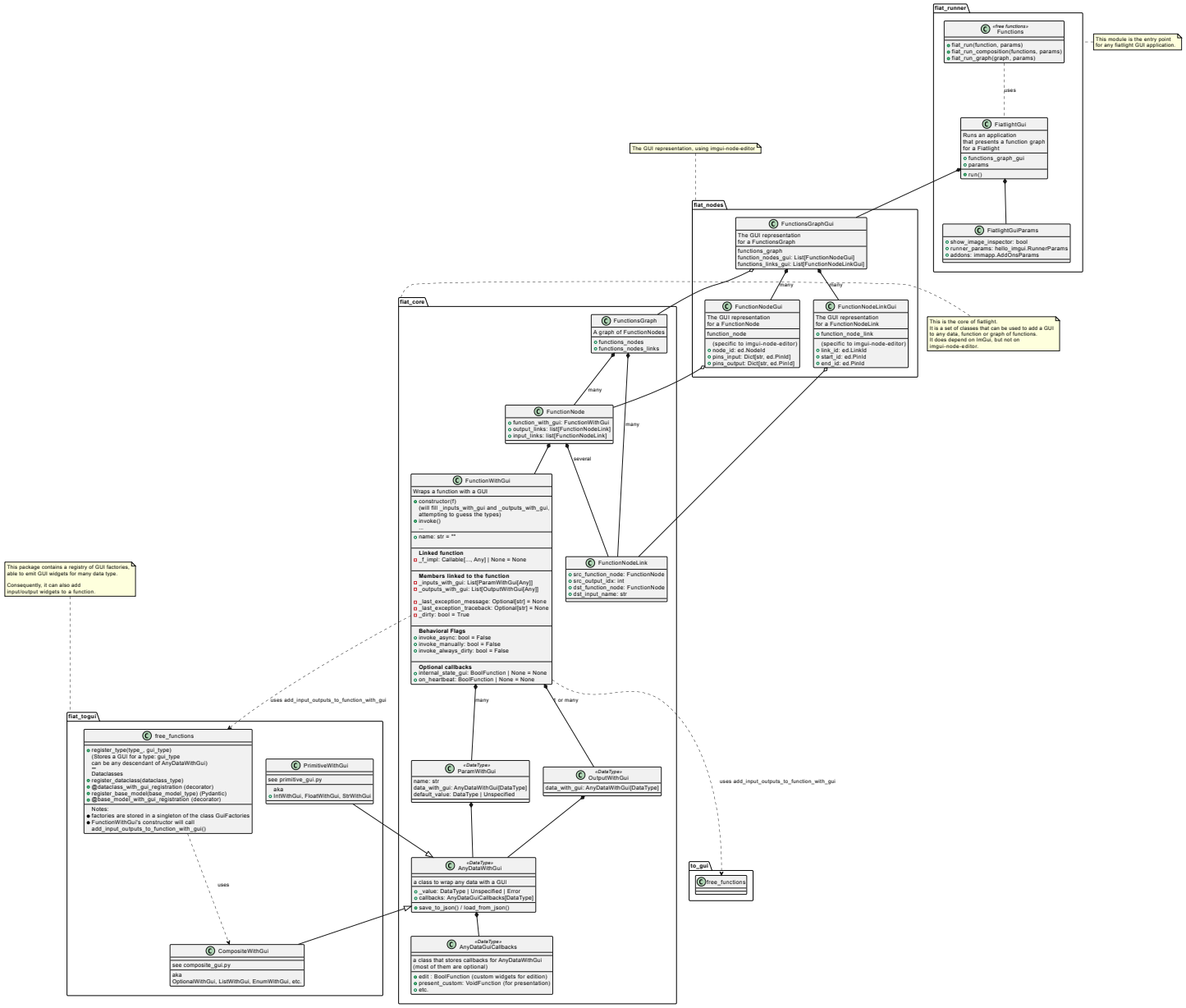
```
%plantuml_include class_diagrams/fiat_nodes.puml
```




## Full diagram

Below is the full class diagram

```
%plantuml_include class_diagrams/all.puml
```




# Folder structure

Below is the folder structure of the fiatlight framework.

```
display_markdown_from_file("folder_structure.md")
```

```
src/python/fiatlight//
├── __init__.py
├── fiat_core/                                  # core classes
│   ├── __init__.py                             # (AnyDataWithGui, FunctionWithGui, et
│   ├── any_data_gui_callbacks.md
│   ├── any_data_gui_callbacks.py
│   ├── any_data_with_gui.md
│   ├── any_data_with_gui.py
│   ├── detailed_type.py
│   ├── function_node.py
│   ├── function_with_gui.py
│   ├── functions_graph.py
│   ├── output_with_gui.py
│   ├── param_with_gui.py
│   ├── possible_fiat_attributes.py
│   └── togui_exception.py
│
├── fiat_runner/                                # runner classes
│   ├── __init__.py                             # (fiat_run)
│   ├── fiat_gui.py
│   ├── fiat_run_notebook.py
│   └── functions_collection.py
│
├── fiat_togui/                                 # fiat_togui utilities
│   ├── Readme.md -> fiat_togui.md              # This is the core of the powerful intr
│   ├── __init__.py                             # capabilities of Fiatlight
│   ├── composite_gui.py
│   ├── composite_gui_demo.py
│   ├── dataclass_examples.py
│   ├── dataclass_gui.py
│   ├── dataclass_gui_demo.py
│   ├── explained_value_gui.py
│   ├── fiat_togui.md
│   ├── file_types_gui.py
│   ├── file_types_gui_demo.py
│   ├── function_signature.py
│   ├── make_gui_demo_code.py
│   ├── primitive_gui_demo.py
│   ├── primitives_gui.py
│   ├── str_with_gui.py
│   ├── str_with_gui_demo.py
│   ├── to_gui.py
│
├── fiat_config/                                # global configuration (style, colors,
│   ├── __init__.py
│   ├── fiat_config_def.py
│   └── fiat_style_def.py
│
├── fiat_types/                                 # Some basic types used throughout Fia
│   ├── __init__.py
│   ├── base_types.py
│   ├── color_types.py
│   ├── error_types.py
```

```
│   ├── fiat_number_types.py
│   ├── file_types.py
│   ├── function_types.py
│   ├── str_types.py
│
├── fiat_kits/                              # domain specific kits
│   ├── Readme.md -> fiat_kits.md
│   ├── __init__.py
│   │
│   ├── fiat_image/                         # image widgets (ImageWithGui, LutGui,
│   │   ├── Readme.md -> fiat_image.md
│   │   ├── __init__.py
│   │   ├── camera_image_provider.py
│   │   ├── camera_image_provider_demo.py
│   │   ├── cv_color_type.py
│   │   ├── fiat_image.md
│   │   ├── fiat_image_attrs_demo.py
│   │   ├── image_gui.py
│   │   ├── image_to_from_file_gui.py
│   │   ├── image_to_from_file_gui_demo.py
│   │   ├── image_types.py
│   │   ├── lut_functions.py
│   │   ├── lut_gui.py
│   │   ├── lut_gui_demo.py
│   │   ├── lut_types.py
│   │   ├── overlay_alpha_image.py
│   │   └── paris.jpg
│   │
│   ├── fiat_matplotlib/                     # Matplotlib widget (FigureWithGui)
│   │   ├── __init__.py
│   │   ├── comparison_dash/
│   │   │   ├── __init__.py
│   │   │   └── figure_demo_dash.py
│   │   ├── comparison_streamlit/
│   │   │   ├── __init__.py
│   │   │   ├── anim_wave_streamlit.py
│   │   │   └── figure_demo_streamlit.py
│   │   ├── fiat_matplotlib.md
│   │   ├── figure_with_gui.py
│   │   └── figure_with_gui_demo.py
│   │
│   ├── fiat_dataframe/                      # Pandas DataFrame widget (DataFrameWi
│   │   ├── Readme.md -> fiat_dataframe.md
│   │   ├── __init__.py
│   │   ├── dataframe_presenter.py
│   │   ├── dataframe_with_gui.py
│   │   ├── dataframe_with_gui_demo_titanic.py
│   │   └── fiat_dataframe.md
│   │
│   ├── fiat_implot/                         # Plots with ImPlot:
│   │   ├── __init__.py                 # SimplePlotGui presents 1D/2D arrays with I
│   │   ├── array_types.py
│   │   ├── simple_plot_gui.py
│   │   └── simple_plot_gui_demo.py
```

```
│   │
│   ├── fiat_ai/                              # Artificial Intelligence kit
│   │   ├── __init__.py                    # (Prompt + Stable Diffusion)
│   │   ├── invoke_sdxl_turbo.py
│   │   ├── invoke_sdxl_turbo_demo.py
│   │   ├── prompt.py
│   │   ├── prompt_with_gui.py
│   │   └── prompt_with_gui_demo.py
│   │
│   ├── fiat_kit_skeleton/                     # Kit skeleton
│   │   ├── Readme.md -> fiat_skeleton.md      # (a starting point to create new kits
│   │   ├── __init__.py
│   │   ├── fiat_skeleton.md
│   │   ├── mydata.py
│   │   ├── mydata_presenter.py
│   │   └── mydata_with_gui.py
│   │
│   ├── experimental/
│   │   ├── __init__.py
│   │   └── fiat_audio_simple/                # audio processing kit (experimental)
│   │       ├── __init__.py
│   │       ├── audio_types.py
│   │       ├── audio_types_gui.py
│   │       ├── microphone_gui.py
│   │       ├── microphone_gui_demo.py
│   │       ├── microphone_io.py
│   │       ├── sound_wave_player.py
│   │       ├── sound_wave_player_demo.py
│   │       ├── sound_wave_player_gui.py
│   │       └── sound_wave_player_gui_demo.py
│
├── fiat_nodes/                                # Present function inside Nodes
│   ├── __init__.py
│   ├── function_node_gui.py
│   ├── functions_graph_gui.py
│
├── fiat_utils/                               # internal utilities
│   ├── __init__.py
│   ├── docstring_utils.py
│   ├── fiat_attributes_decorator.py
│   ├── fiat_math.py
│   ├── functional_utils.py
│   ├── lazy_module.py
│   ├── print_repeatable_message.py
│   ├── registry.py
│   └── str_utils.py
│
├── fiat_widgets/                              # internal widgets
│   ├── __init__.py
│   ├── fiat_osd.py
│   ├── float_widgets.py
│   ├── fontawesome6_ctx_utils.py
│   ├── group_panel.py
│   ├── mini_buttons.py
```

```
│   ├── misc_widgets.py
│   ├── node_separator.py
│   └── ribbon_panel.py
│
├── fiat_cli/                                  # Command Line Interface
│   ├── __init__.py
│   ├── fiatlight_cli.py*
│
├── fiat_doc/                                  # documentation utilities
│   ├── __init__.py
│   ├── code_utils.py
│   ├── make_class_header.py
├── fiat_notebook/                            # notebook utilities
│   ├── look_at_code.py
│   ├── notebook_utils.py
│   └── plantuml_magic.py
├── py.typed
```

# FunctionWithGui

## Introduction

`FunctionWithGui` is one of the core classes of FiatLight: it wraps a function with a GUI that presents its inputs and outputs.

- **Manual**: Read the manual for a detailed guide on how to use it.
- **Source code**: View its full code online.

## Signature

Below, you will find the "signature" of the `FunctionWithGui` class, with its main attributes and methods (but not their bodies)

Its full source code is available online.

```
from fiatlight.fiat_notebook import look_at_code
%look_at_class_header fiatlight.fiat_core.FunctionWithGui
```

```
class FunctionWithGui:
    """FunctionWithGui: add GUI to a function

    `FunctionWithGui` is one of the core classes of FiatLight: it wraps a function w
    inputs and its output(s).

    Public Members
    ==============
    # the name of the function
    name: str = ""

    #
    # Behavioral Flags
    # ----------------
    # invoke_async: if true, the function shall be called asynchronously
    invoke_async: bool = False

    # invoke_manually: if true, the function will be called only if the user clicks
    # (if inputs were changed, a "Refresh needed" label will be displayed)
    invoke_manually: bool = False

    # invoke_always_dirty: if true, the function output will always be considered ou
    #   - if invoke_manually is true, the "Refresh needed" label will be displayed
    #   - if invoke_manually is false, the function will be called at each frame
    # Note: a "live" function is thus a function with invoke_manually=False and invo
    invoke_always_dirty: bool = False

    # Optional user documentation to be displayed in the GUI
    #     - doc_display: if True, the doc string is displayed in the GUI (default: F
    #     - doc_is_markdown: if True, the doc string is in Markdown format (default:
    #     - doc_user: the documentation string. If not provided, the function docstr
    #     - doc_show_source: if True, the source code of the function will be displa
    doc_display: bool = True
    doc_markdown: bool = True
    doc_user: str = ""
    doc_show_source: bool = False

    #
    # Internal state GUI
    # ------------------
    # internal_state_gui: optional Gui for the internal state of the function
    # (this function may display a GUI to show the internal state of the function,
    #  and return True if the state has changed, and the function needs to be called
    internal_state_gui: BoolFunction | None = None

    # internal_state_gui_node_compatible:
    # If True, the internal_state_gui function is incompatible with being presented
    # (this is due to a limitation of the node editor, which cannot render scrollabl
    # Note: instead of setting edit_node_compatible to False, you may query
    #       `fiatlight.is_rendering_in_node()` to know if you are rendering in a nod
    #       and choose alternative widgets in this case.
    internal_state_gui_node_compatible: bool = True
```

```
#
# Heartbeat
# ---------
# on_heartbeat: optional function that will be called at each frame
# (and return True if the function needs to be called to update the output)
on_heartbeat: BoolFunction | None = None

#
# Serialization
# -------------
# save/load_internal_gui_options_from_json (Optional)
# Optional serialization and deserialization of the internal state GUI presentat
# (i.e. anything that deals with how the GUI is presented, not the data itself)
# If provided, these functions will be used to recreate the GUI presentation opt
# so that the GUI looks the same when the application is restarted.
save_internal_gui_options_to_json: Callable[[], JsonDict] | None = None
load_internal_gui_options_from_json: Callable[[JsonDict], None] | None = None

"""
function_name: str = ''
label: str = ''
invoke_async: bool = False
invoke_manually: bool = False
invoke_always_dirty: bool = False
invoke_is_gui_only: bool = False
doc_display: bool = True
doc_markdown: bool = True
doc_user: str = ''
doc_show_source: bool = False
internal_state_gui: BoolFunction | None = None
internal_state_gui_node_compatible: bool = True
save_internal_gui_options_to_json: Callable[[], JsonDict] | None = None
load_internal_gui_options_from_json: Callable[[JsonDict], None] | None = None
on_heartbeat: BoolFunction | None = None
_dirty: bool = True
_f_impl: Callable[..., Any] | None = None
_inputs_with_gui: List[ParamWithGui[Any]]
_outputs_with_gui: List[OutputWithGui[Any]]
_last_exception_message: Optional[str] = None
_last_exception_traceback: Optional[str] = None
_accept_none_as_output: bool = False

class _Construct_Section:
    """
    # ------------------------------------------------------------------------
    #        Construction
    #  input_with_gui and output_with_gui should be filled soon after constructi
    # ------------------------------------------------------------------------
    """
    pass

def __init__(self, fn: Callable[..., Any] | None, fn_name: str | None=None, *, s
    """Create a FunctionWithGui object, with the given function as implementatio
```

```
    The function signature is automatically parsed, and the inputs and outputs a
    with the correct GUI types.

    :param fn: the function for which we want to create a FunctionWithGui

    Notes:
    This function will capture the locals and globals of the caller to be able t
    Make sure to call this function *from the module where the function and its

    If the function has attributes like invoke_manually or invoke_async, they wi
        - if `invoke_async` is True, the function will be called asynchronously
        - if `invoke_manually` is True, the function will be called only if the


    Advanced parameters:
    ********************
    :param signature_string: a string representing the signature of the functior
                             used when the function signature cannot be retrieve
    """
    pass

class _FiatAttributes_Section:
    """
    # ------------------------------------------------------------------------
    #        Fiat Attributes
    # ------------------------------------------------------------------------
    """
    pass

def handle_fiat_attributes(self, fiat_attributes: dict[str, Any]) -> None:
    """Handle custom attributes for the function"""
    pass

def set_invoke_live(self) -> None:
    """Set flags to make this a live function (called automatically at each fram
    pass

def set_invoke_manually(self) -> None:
    """Set flags to make this a function that needs to be called manually"""
    pass

def set_invoke_manually_io(self) -> None:
    """Set flags to make this a IO function that needs to be called manually
    and that is always considered dirty, because it depends on an external devic
    or state (and likely has no input)"""
    pass

def is_invoke_manually_io(self) -> bool:
    """Return True if the function is an IO function that needs to be called mar
    pass

def set_invoke_async(self) -> None:
    """Set flags to make this a function that is called asynchronously"""
    pass
```

```
    def is_live(self) -> bool:
        """Return True if the function is live"""
        pass

    class _Utilities_Section:
        """
        # ------------------------------------------------------------------------
        #        Utilities
        # ------------------------------------------------------------------------
        """
        pass

    def call_for_tests(self, **params: Any) -> Any:
        """Call the function with the given parameters, for testing purposes"""
        pass

    def is_dirty(self) -> bool:
        """Return True if the function needs to be called, because the inputs have c
        pass

    def set_dirty(self) -> None:
        """Set the function as dirty."""
        pass

    def get_last_exception_message(self) -> str | None:
        """Return the last exception message, if any"""
        pass

    def shall_display_refresh_needed_label(self) -> bool:
        """Return True if the "Refresh needed" label should be displayed
        i.e. if the function is dirty and invoke_manually is True"""
        pass

    def __str__(self) -> str:
        pass

    class _Inputs_Section:
        """
        # ------------------------------------------------------------------------
        #        Inputs, aka parameters
        # ------------------------------------------------------------------------
        """
        pass

    def nb_inputs(self) -> int:
        """Return the number of inputs of the function"""
        pass

    def all_inputs_names(self) -> List[str]:
        """Return the names of all the inputs of the function"""
        pass

    def input(self, name: str) -> AnyDataWithGui[Any]:
```

```
        """Return the input with the given name as a AnyDataWithGui[Any]
        The inner type of the returned value is Any in this case.
        You may have to cast it to the correct type, if you rely on type hints.

        Use input_as() if you want to get the input with the correct type.
        """
        pass

    def input_as(self, name: str, gui_type: Type[GuiType]) -> GuiType:
        """Return the input with the given name as a GuiType

        GuiType can be any descendant of AnyDataWithGui, like
            fiatlight.fiat_core.IntWithGui, fiatlight.fiat_core.FloatWithGui, etc.

        Raises a ValueError if the input is not found, and a TypeError if the input
        """
        pass

    def input_of_idx(self, idx: int) -> ParamWithGui[Any]:
        """Return the input with the given index as a ParamWithGui[Any]"""
        pass

    def input_of_idx_as(self, idx: int, gui_type: Type[GuiType]) -> GuiType:
        """Return the input with the given index as a GuiType"""
        pass

    def inputs_guis(self) -> List[AnyDataWithGui[Any]]:
        pass

    def set_input_gui(self, name: str, gui: AnyDataWithGui[Any]) -> None:
        """Set the GUI for the input with the given name"""
        pass

    def has_param(self, name: str) -> bool:
        """Return True if the function has a parameter with the given name"""
        pass

    def param(self, name: str) -> ParamWithGui[Any]:
        """Return the input with the given name as a ParamWithGui[Any]"""
        pass

    def param_gui(self, name: str) -> AnyDataWithGui[Any]:
        """Return the input with the given name as a AnyDataWithGui[Any]"""
        pass

    def set_param_value(self, name: str, value: Any) -> None:
        """Set the value of the input with the given name
        This is useful to set the value of an input programmatically, for example ir
        """
        pass

    def toggle_expand_inputs(self) -> None:
        pass
```

```
def toggle_expand_outputs(self) -> None:
    pass

class _Outputs_Section:
    """
    # ------------------------------------------------------------------------
    #        Outputs
    # ------------------------------------------------------------------------
    """
    pass

def nb_outputs(self) -> int:
    """Return the number of outputs of the function.
    A function typically has 0 or 1 output, but it can have more if it returns a
    """
    pass

def output(self, output_idx: int=0) -> AnyDataWithGui[Any]:
    """Return the output with the given index as a AnyDataWithGui[Any]
    The inner type of the returned value is Any in this case.
    You may have to cast it to the correct type, if you rely on type hints.

    Use output_as() if you want to get the output with the correct type.
    """
    pass

def output_as(self, output_idx: int, gui_type: Type[GuiType]) -> GuiType:
    """Return the output with the given index as a GuiType

    GuiType can be any descendant of AnyDataWithGui, like
        fiatlight.fiat_core.IntWithGui, fiatlight.fiat_core.FloatWithGui, etc.

    Raises a ValueError if the output is not found, and a TypeError if the outpu
    """
    pass

def outputs_guis(self) -> List[AnyDataWithGui[Any]]:
    pass

class _Invoke_Section:
    """
    # ------------------------------------------------------------------------
    #        Invoke the function
    # This is the heart of fiatlight: it calls the function with the current inp
    # and stores the result in the outputs, stores the exception if any, etc.
    # ------------------------------------------------------------------------
    """
    pass

@final
def has_bad_inputs(self) -> bool:
    pass

@final
```

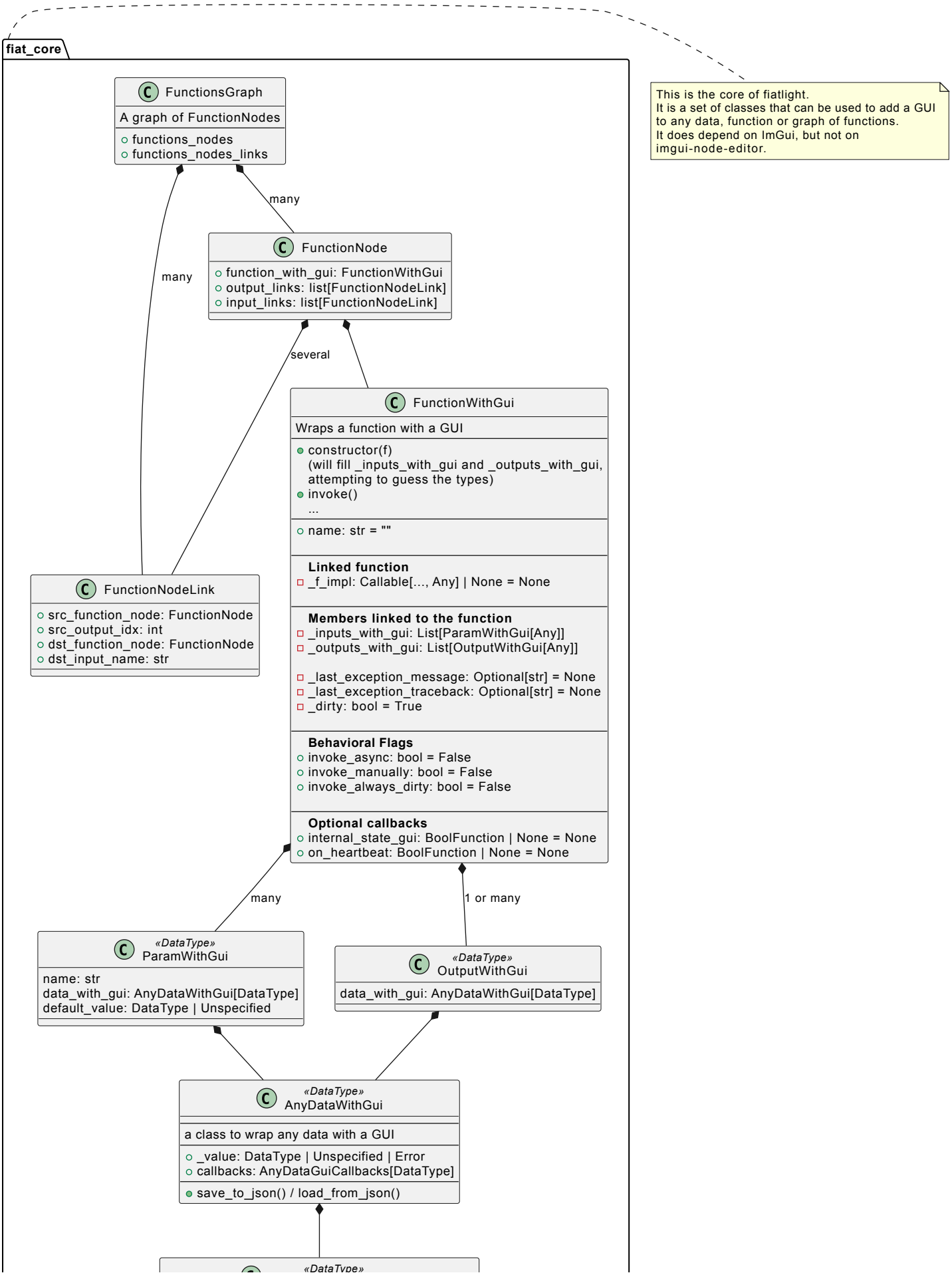
```
def invoke(self) -> None:
    """Invoke the function with the current inputs, and store the result in the

    Will call the function if:
     - the inputs have changed since the last call
     - the function is dirty
     - none of the inputs is an error or unspecified

    If an exception is raised, the outputs will be set to ErrorValue, and the ex

    If the function returned None and the output is not allowed to be None, a Va
    (this is inferred from the function signature)
    """
    pass

def invoke_gui(self) -> None:
    pass

@final
def _invoke_impl(self) -> None:
    pass

def on_exit(self) -> None:
    """Called when the application is exiting
    Will call the on_exit callback of all the inputs and outputs
    """
    pass

def _can_emit_none_output(self) -> bool:
    """Return True if the function can emit None as output
    i.e.
    - either the function has no output
    - or the output can be None (i.e. the signature looks like `def f() -> int |
    if the function has multiple outputs, we consider that it can not emit None
    """
    pass

class _Serialize_Section:
    """
    # ------------------------------------------------------------------------
    #        Save and load to json
    # Here, we only save the options that the user entered manually in the GUI:
    #   - the options of the inputs
    #   - the options of the outputs
    # ------------------------------------------------------------------------
    """
    pass

def save_user_inputs_to_json(self) -> JsonDict:
    pass

def load_user_inputs_from_json(self, json_data: JsonDict) -> None:
    pass
```

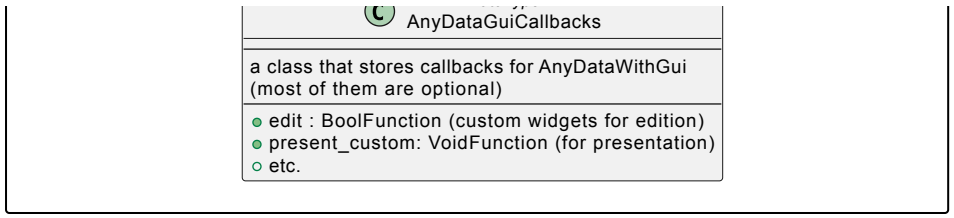
```
    def save_gui_options_to_json(self) -> JsonDict:
        """Save the GUI options to a JSON file
        (i.e. any presentation options of the inputs and outputs, as well as of the
        """
        pass

    def load_gui_options_from_json(self, json_data: JsonDict) -> None:
        """Load the GUI options from a JSON file"""
        pass

    class _Doc_Section:
        pass

    def get_function_doc(self) -> FunctionWithGuiDoc:
        pass

    def _get_function_userdoc(self) -> str | None:
        """Return the user documentation of the function"""
        pass

    def _get_function_docstring(self) -> str | None:
        """Return the docstring of the function"""
        pass

    def _get_function_source_code(self) -> str | None:
        """Return the source code of the function"""
        pass
```

## Architecture

Below is a PlantUML diagram showing the architecture of the `fiat_core` module. See the architecture page for the full architecture diagrams.

```
from fiatlight.fiat_notebook import plantuml_magic
%plantuml_include class_diagrams/fiat_core.puml
```


fiat_core
This is the core of fiatlight.
It is a set of classes that can be used to add a GUI
to any data, function or graph of functions.
It does depend on ImGui, but not on
imgui-node-editor.
FunctionsGraph
A graph of FunctionNodes
functions_nodes
functions_nodes_links
many
many
FunctionNode
function_with_gui: FunctionWithGui
output_links: list[FunctionNodeLink]
input_links: list[FunctionNodeLink]
several
FunctionWithGui
Wraps a function with a GUI
constructor(f)
(will fill _inputs_with_gui and _outputs_with_gui,
attempting to guess the types)
invoke()
...
name: str = ""
Linked function
_f_impl: Callable[..., Any] | None = None
Members linked to the function
_inputs_with_gui: List[ParamWithGui[Any]]
_outputs_with_gui: List[OutputWithGui[Any]]
_last_exception_message: Optional[str] = None
_last_exception_traceback: Optional[str] = None
_dirty: bool = True
Behavioral Flags
invoke_async: bool = False
invoke_manually: bool = False
invoke_always_dirty: bool = False
Optional callbacks
internal_state_gui: BoolFunction | None = None
on_heartbeat: BoolFunction | None = None
FunctionNodeLink
src_function_node: FunctionNode
src_output_idx: int
dst_function_node: FunctionNode
dst_input_name: str
many
1 or many
«DataType»
ParamWithGui
name: str
data_with_gui: AnyDataWithGui[DataType]
default_value: DataType | Unspecified
«DataType»
OutputWithGui
data_with_gui: AnyDataWithGui[DataType]
«DataType»
AnyDataWithGui
a class to wrap any data with a GUI
_value: DataType | Unspecified | Error
callbacks: AnyDataGuiCallbacks[DataType]
save_to_json() / load_from_json()




# AnyDataWithGui

## Introduction

AnyDataWithGui associate a GUI to any type, with associated GUI callbacks, allowing for custom rendering, editing, serialization, and event handling within the Fiatlight framework.

It uses callbacks which are stored inside AnyDataGuiCallback.

## Signature

Below, we display the class header, i.e., the class without its methods bodies, to give a quick overview of its structure.

You can see its full code at AnyDataWithGui.

```
from fiatlight.fiat_notebook import look_at_code
%look_at_class_header fiatlight.fiat_core.AnyDataWithGui
```

```
class AnyDataWithGui(Generic[DataType]):
    """AnyDataWithGui: a GUI associated to a type.

    AnyDataWithGui[DataType]
    ========================

    This class manages data of any type with associated GUI callbacks, allowing for
    serialization, and event handling within the Fiatlight framework.

    Members:
    --------
    # The type of the data, e.g. int, str, typing.List[int], typing.Tuple[int, str],
    _type: Type[DataType]

    # The value of the data - can be a DataType, Unspecified, or Error
    # It is accessed through the value property, which triggers the on_change callba
    _value: DataType | Unspecified | Error = UnspecifiedValue

    # Callbacks for the GUI
    # This is the heart of FiatLight: the GUI is defined by the callbacks.
    # Think of them as __dunder__ methods for the GUI.
    callbacks: AnyDataGuiCallbacks[DataType]

    # If True, the value can be None. This is useful when the data is optional.
    # Otherwise, any None value will be considered as an Error.
    # Note: when using Optional[any registered type], this flag is automatically set
    can_be_none: bool = False

    Property:
    ---------
    # Custom attributes that can be set by the user, to give hints to the GUI.
    # For example, with this function declaration,
    #         def f(x: int, y: int) -> int:
    #             return x + y
    #         f.x__range = (0, 10)
    # fiat_attributes["range"] will be (0, 10) for the parameter x.
    @property
    fiat_attributes -> dict[str, Any]

    """
    _type: Type[DataType] | None
    _value: DataType | Unspecified | Error | Invalid[DataType] = UnspecifiedValue
    callbacks: AnyDataGuiCallbacks[DataType]
    can_be_none: bool = False
    _fiat_attributes: FiatAttributes
    _expanded: bool = False
    _can_set_unspecified_or_default: bool = False
    label: str | None = None
    label_color: ImVec4 | None = None
    tooltip: str | None = None
    status_tooltip: str | None = None

    class CollapseOrExpand(Enum):
```

```
    collapse = 'Collapse All'
    expand = 'Expand All'

class PresentOrEdit(Enum):
    present = 'View'
    edit = 'Edit'

class _Init_Section:
    """
    # ----------------------------------------------------------------------
    #            Initialization
    # ----------------------------------------------------------------------
    """
    pass

def __init__(self, data_type: Type[DataType] | None) -> None:
    """Initialize the AnyDataWithGui with a type, an unspecified value, and no c
    pass

class _Value_Section:
    """
    # ----------------------------------------------------------------------
    #            Value getter and setter + get_actual_value (which returns a Dat
    # ----------------------------------------------------------------------
    """
    pass

@property
def value(self) -> DataType | Unspecified | Error | Invalid[DataType]:
    """The value of the data, accessed through the value property.
    Warning: it might be an instance of `Unspecified` (user did not enter any va
    """
    pass

@value.setter
def value(self, new_value: DataType | Unspecified | Error | Invalid[DataType]) -
    """Set the value of the data. This triggers the on_change callback (if set)'
    pass

def get_actual_value(self) -> DataType:
    """Returns the actual value of the data, or raises an exception if the value
    When we are inside a callback, we can be sure that the value is of the corre
    instead of accessing the value directly and checking for Unspecified or Erro
    """
    pass

def get_actual_or_invalid_value(self) -> DataType:
    """Returns the actual value of the data, or raises an exception if the value
    pass

class _CustomAttributes_Section:
    """
    # ----------------------------------------------------------------------
    #            Custom Attributes
```

```
    # ------------------------------------------------------------------------
    """
    pass

@staticmethod
def possible_fiat_attributes() -> PossibleFiatAttributes | None:
    """Return the possible custom attributes for this type, if available.
    Should be overridden in subclasses, when custom attributes are available.

    It is strongly advised to return a class variable, or a global variable
    to avoid creating a new instance each time this method is called.
    """
    pass

@final
def possible_fiat_attributes_with_generic(self) -> tuple[PossibleFiatAttributes
    pass

@property
def fiat_attributes(self) -> FiatAttributes:
    pass

def merge_fiat_attributes(self, fiat_attrs: FiatAttributes) -> None:
    """Merge custom attributes with the existing ones"""
    pass

def _handle_generic_attrs(self) -> None:
    """Handle generic custom attributes"""
    pass

@staticmethod
def propagate_label_and_tooltip(a: 'AnyDataWithGui[Any]', b: 'AnyDataWithGui[Any
    """Propagate label and tooltip from one AnyDataWithGui to another
    Meant to be used with CompositeGui
    """
    pass

class _Gui_Section:
    """
    # ------------------------------------------------------------------------
    #            Gui sections
    #      (Can also be used outside a function Node)
    # ------------------------------------------------------------------------
    """

def sub_items_can_collapse(self, _present_or_edit: PresentOrEdit) -> bool:
    """Overwrite this in derived classes if they provide multiple sub-items that
    pass

def sub_items_collapse_or_expand(self, _collapse_or_expand: CollapseOrExpand) ->
    """Overwrite this in derived classes if they provide multiple sub-items that
    pass

def sub_items_will_collapse_or_expand(self, _present_or_edit: PresentOrEdit) ->
```

```
        """Overwrite this in derived classes if they provide multiple sub-items that
        pass

    def _show_collapse_sub_items_buttons(self, present_or_edit: PresentOrEdit) -> No
        pass

    def can_show_present_popup(self) -> bool:
        pass

    def can_show_edit_popup(self) -> bool:
        pass

    def _show_collapse_button(self) -> None:
        pass

    def _show_copy_to_clipboard_button(self) -> None:
        pass

    def can_collapse_present(self) -> bool:
        pass

    def can_collapse_edit(self) -> bool:
        pass

    def can_edit_on_header_line(self) -> bool:
        pass

    def can_present_on_header_line(self) -> bool:
        pass

    def _can_edit_on_next_lines_if_expanded(self) -> bool:
        pass

    def _can_present_on_next_lines_if_expanded(self) -> bool:
        pass

    def _is_editing_on_next_lines(self) -> bool:
        pass

    def _is_presenting_on_next_lines(self) -> bool:
        pass

    def _popup_window_name(self, params: GuiHeaderLineParams[DataType], present_or_e
        pass

    def _gui_present_header_line(self, params: GuiHeaderLineParams[DataType]) -> Nor
        """Present the value as a string in one line, or as a widget if it fits on c
        pass

    def _gui_edit_header_line(self, params: GuiHeaderLineParams[DataType]) -> bool:
        pass

    def _show_set_unspecified_or_default_button(self) -> bool:
        pass
```

```
def _gui_edit_next_lines(self, in_popup: bool) -> bool:
    pass

def _gui_present_next_lines(self, in_popup: bool) -> None:
    pass

def gui_present_customizable(self, params: GuiHeaderLineParams[DataType]) -> Nor
    """Present the value using either the present callback or the default str cc
    May present on one line (if possible) or on multiple lines with an expand bu
    """
    pass

def gui_present(self) -> None:
    pass

def gui_edit_customizable(self, params: GuiHeaderLineParams[DataType]) -> bool:
    """Call the edit callback. Returns True if the value has changed
    May edit on one line (if possible) or on multiple lines with an expand butto
    """
    pass

def gui_edit(self) -> bool:
    pass

class _Callbacks_Section:
    """
    # ------------------------------------------------------------------------
    #            Callbacks sections
    # ------------------------------------------------------------------------
    """

def set_edit_callback(self, edit_callback: DataEditFunction[DataType]) -> None:
    """Helper function to set the edit callback from a free function"""
    pass

def set_present_callback(self, present_callback: DataPresentFunction[DataType],
    """Helper function to set the present custom callback from a free function"'
    pass

def add_validate_value_callback(self, cb: Callable[[DataType], None]) -> None:
    pass

def _Serialization_Section(self) -> None:
    """
    # ------------------------------------------------------------------------
    #            Serialization and deserialization
    # ------------------------------------------------------------------------
    """
    pass

@final
def call_save_to_dict(self, value: DataType | Unspecified | Error | Invalid[Data
    """Serialize the value to a dictionary
```

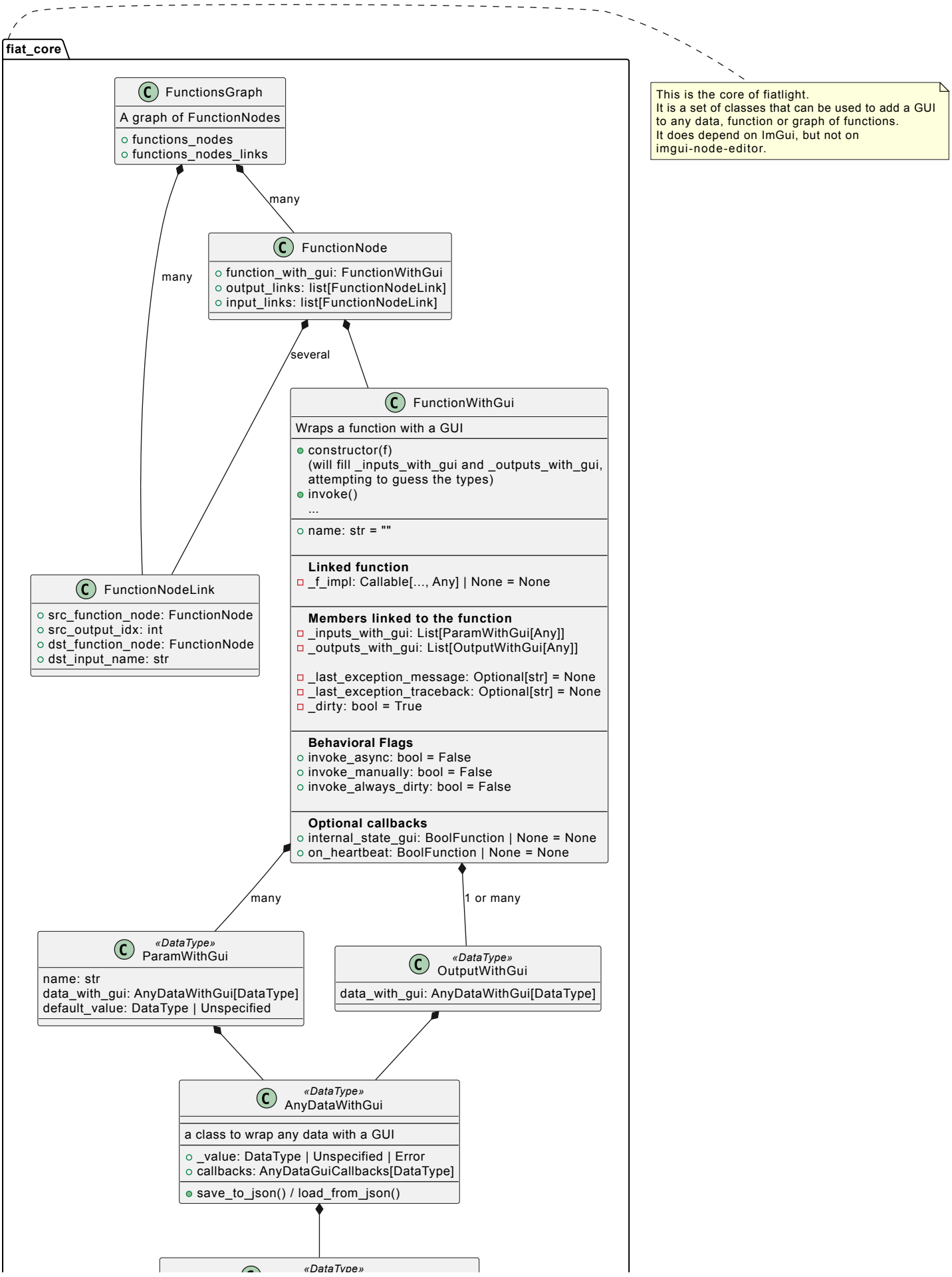
```
        Will call the save_to_dict callback if set, otherwise will use the default s
        A default serialization is available for primitive types, tuples, and Pydant

        (This is how fiatlight saves the data to a JSON file)

        Do not override these methods in descendant classes!
        """
        pass

    @final
    def call_load_from_dict(self, json_data: JsonDict) -> DataType | Unspecified | E
        """Deserialize the value from a dictionary
        Do not override these methods in descendant classes!
        """
        pass

    @final
    def call_save_gui_options_to_json(self) -> JsonDict:
        pass

    @final
    def call_load_gui_options_from_json(self, json_data: JsonDict) -> None:
        pass

    class _Utilities_Section:
        """
        # ----------------------------------------------------------------------
        #             Utilities
        # ----------------------------------------------------------------------
        """

    def can_construct_default_value(self) -> bool:
        pass

    def construct_default_value(self) -> DataType:
        pass

    def datatype_qualified_name(self) -> str:
        pass

    def datatype_basename(self) -> str:
        pass

    def datatype_base_and_qualified_name(self) -> str:
        pass

    def datatype_value_to_str(self, value: DataType) -> str:
        """Convert the value to a string
        Uses either the present_str callback, or the default str conversion
        """
        pass

    def datatype_value_to_clipboard_str(self, value: DataType) -> str:
```

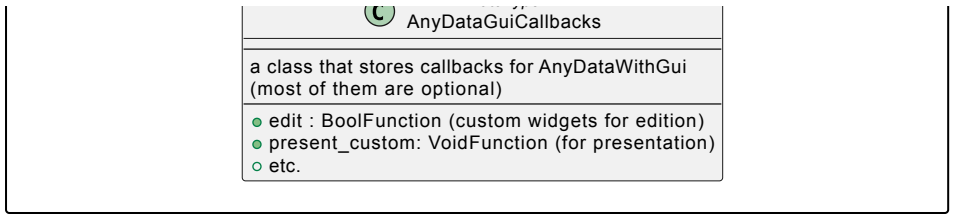
```
        """Convert the value to a string for the clipboard
        Uses either the clipboard_copy_str callback, or the default str conversion
        """
        pass

    def docstring_first_line(self) -> str | None:
        """Return the first line of the docstring, if available"""
        pass
```

## Architecture

Below is a PlantUML diagram showing the architecture of the `fiat_core` module. See the architecture page for the full architecture diagrams.

```
from fiatlight.fiat_notebook import plantuml_magic
%plantuml_include class_diagrams/fiat_core.puml
```


fiat_core
This is the core of fiatlight.
It is a set of classes that can be used to add a GUI
to any data, function or graph of functions.
It does depend on ImGui, but not on
imgui-node-editor.
FunctionsGraph
A graph of FunctionNodes
functions_nodes
functions_nodes_links
many
many
FunctionNode
function_with_gui: FunctionWithGui
output_links: list[FunctionNodeLink]
input_links: list[FunctionNodeLink]
several
FunctionWithGui
Wraps a function with a GUI
constructor(f)
(will fill _inputs_with_gui and _outputs_with_gui,
attempting to guess the types)
invoke()
...
name: str = ""
Linked function
_f_impl: Callable[..., Any] | None = None
Members linked to the function
_inputs_with_gui: List[ParamWithGui[Any]]
_outputs_with_gui: List[OutputWithGui[Any]]
_last_exception_message: Optional[str] = None
_last_exception_traceback: Optional[str] = None
_dirty: bool = True
Behavioral Flags
invoke_async: bool = False
invoke_manually: bool = False
invoke_always_dirty: bool = False
Optional callbacks
internal_state_gui: BoolFunction | None = None
on_heartbeat: BoolFunction | None = None
FunctionNodeLink
src_function_node: FunctionNode
src_output_idx: int
dst_function_node: FunctionNode
dst_input_name: str
many
1 or many
«DataType»
ParamWithGui
name: str
data_with_gui: AnyDataWithGui[DataType]
default_value: DataType | Unspecified
«DataType»
OutputWithGui
data_with_gui: AnyDataWithGui[DataType]
«DataType»
AnyDataWithGui
a class to wrap any data with a GUI
_value: DataType | Unspecified | Error
callbacks: AnyDataGuiCallbacks[DataType]
save_to_json() / load_from_json()




# AnyDataGuiCallbacks

## Introduction

AnyDataGuiCallbacks provides a set of callbacks that define how a particular data type should be presented, edited, and managed within the Fiatlight GUI framework.

These callbacks are used by AnyDataWithGui.

## Source

Below, is the class source, which you can also see online.

```
from fiatlight.fiat_notebook import look_at_code
%look_at_python_code fiatlight.fiat_core.AnyDataGuiCallbacks
```

```python
class AnyDataGuiCallbacks(Generic[DataType]):
    """AnyDataGuiCallbacks: Collection of callbacks for a given type

    AnyDataGuiCallbacks
    ===================

    This class provides a set of callbacks that define how a particular data type sh
    presented, edited, and managed within the Fiatlight GUI framework.

    These callbacks are used by [AnyDataWithGui](any_data_with_gui).

    """

    #                         Presentation
    # ------------------------------------------------------------------------------
    # present_str: (Mandatory if str() is not enough, optional otherwise)
    # Provide a function that returns a short string info about the data content
    # This string will be presented as a short description of the data in the GUI
    #
    # If possible, it should be short enough to fit in a single line inside a node (
    # If the result string is too long, or occupies more than one line, it will be t
    # (and the rest of the string will be displayed in a tooltip)
    # For example, on complex types such as images, return something like "128x128x3
    # If not provided, the data will be presented using str()
    present_str: Callable[[DataType], str] | None = None

    # present: (Optional)
    # a function that provides a more complex, custom GUI representation of the data
    # It will be presented when a function param is in "expanded" mode, and can use
    # If not provided, the data will be presented using present_str
    #
    # Note: Some widgets cannot be presented in a Node (e.g., a multiline text input
    #       You can query `fiatlight.is_rendering_in_node()` to know if you are rend
    #       Also, when inside a Node, you may want to render a smaller version, to s
    #       (as opposed to rendering a larger version in a detached window).
    present: Callable[[DataType], None] | None = None

    # present_collapsible:
    # Set this to False if your custom presentation is small and fits in one line
    # (i.e. it does not need to be collapsible)
    # If True, the gui presentation will either:
    #     - show present_str + an expand button
    #     - show the custom presentation + a collapse button
    present_collapsible: bool = True

    # present_node_compatible: (Optional: set to False if using input_text_multiline
    # If True, the present function is incompatible with being presented in a node (
    # of the node editor, which cannot render scrollable widgets)
    # Note: instead of setting edit_node_compatible to False, you may query
    #       `fiatlight.is_rendering_in_node()` to know if you are rendering in a nod
    #       and choose alternative widgets in this case.
    present_node_compatible: bool = True
```

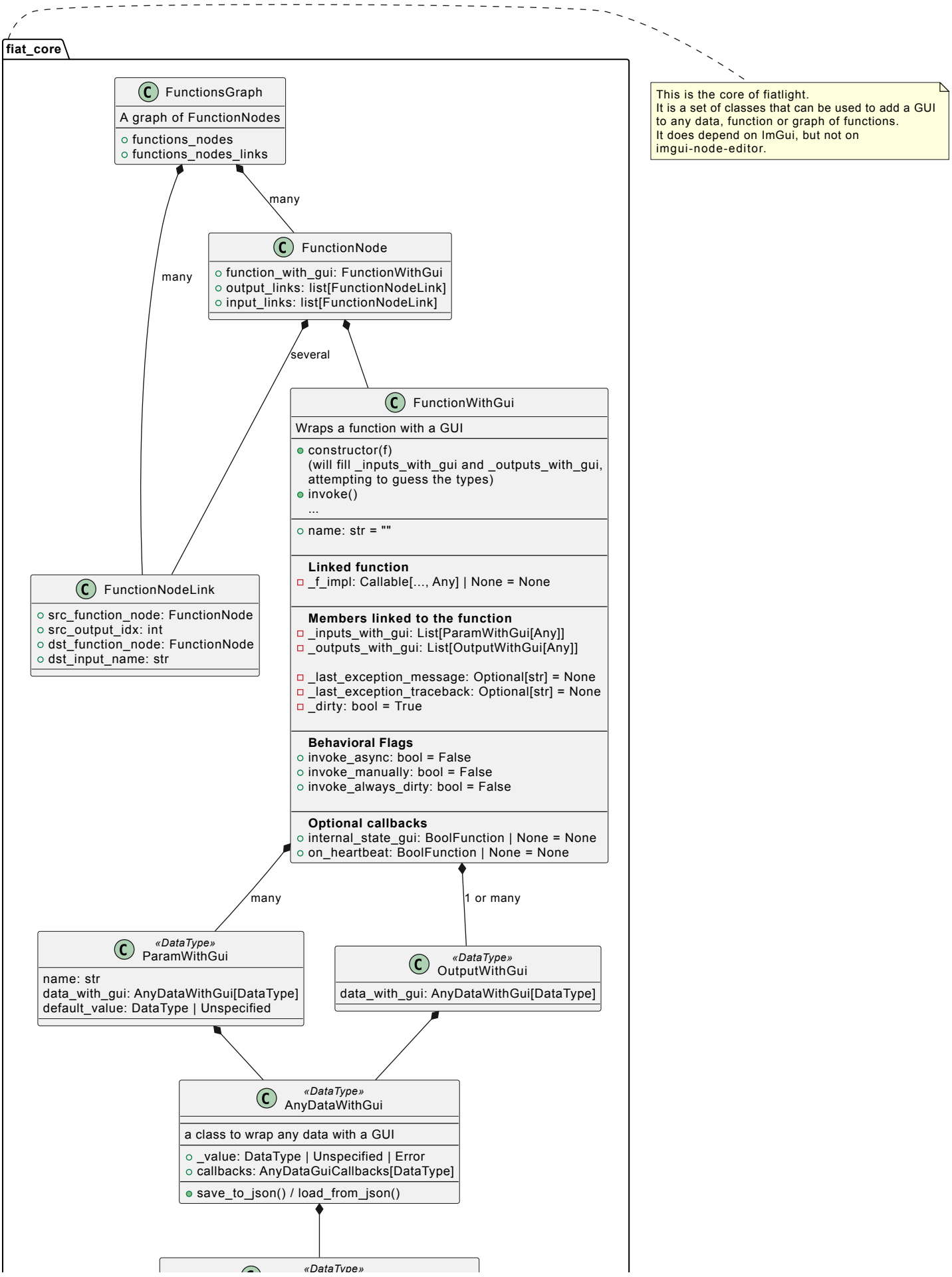
```
    # ------------------------------------------------------------------------

    #                        Edition
    # ------------------------------------------------------------------------
    # edit: (Mandatory if edition is required)
    # Provide a function that presents an editable interface for the data, and retur
    #     (True, new_value) if changed
    #     (False, old_value) if not changed
    # If not provided, the data will be presented as read-only
    # Note: Some widgets cannot be presented in a Node (e.g., a multiline text input
    #       You can query `fiatlight.is_rendering_in_node()` to know if you are rend
    edit: Callable[[DataType], tuple[bool, DataType]] | None = None

    # edit_collapsible:
    # Set this to False if your custom edition is small, and does not need to be col
    # If True, the gui edition will either:
    #     - show present_str + an expand button
    #     - show the custom edition + a collapse button
    edit_collapsible: bool = True

    # edit_node_compatible: (Optional: set to False if using input_text_multiline, c
    # If True, the edit function is incompatible with being presented in a node (thi
    # of the node editor, which cannot render scrollable widgets)
    # Note: instead of setting edit_node_compatible to False, you may query
    #       `fiatlight.is_rendering_in_node()` to know if you are rendering in a nod
    #       and choose alternative widgets in this case.
    edit_node_compatible: bool = True

    # ------------------------------------------------------------------------

    #                        Default value
    # ------------------------------------------------------------------------
    # default value provider (Needed only for a type without default constructor)
    # this function will be called to provide a default value if needed
    default_value_provider: Callable[[], DataType] | None = None
    # ------------------------------------------------------------------------

    #                        Events callbacks
    # ------------------------------------------------------------------------
    # on_change (Optional)
    # if provided, this function will be called when the value changes.
    # Can be used in more advanced cases,
    # for example when `present` has an internal cache that needs to be updated,
    # or other side effects.
    on_change: Callable[[DataType], None] | None = None

    # validate_value (Optional)
    # if provided, these functions will be called when the user tries to set a value
    # They should return a DataValidationResult.ok() if the value is valid,
    # or a DataValidationResult.error() with an error message.
    validate_value: list[Callable[[DataType], DataValidationResult]]

    # on_exit (Optional)
    # if provided, this function will be called when the application is closed.
```

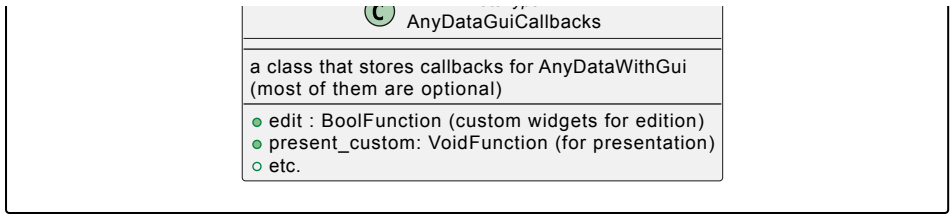
```
    # Used in more advanced cases, typically when some resources need to be released
    on_exit: VoidFunction | None = None

    # on_heartbeat: (Optional)
    # If provided, this function will be called at each heartbeat of the function nc
    # (before the value is drawn). It should return True if any change has been made
    on_heartbeat: BoolFunction | None = None

    # on_fiat_attributes_changed (Optional)
    # if provided, this function will be called when the custom attributes of the da
    # Used in more advanced cases, when the data presentation depends on custom attr
    on_fiat_attributes_changed: Callable[[FiatAttributes], None] | None = None

    # ------------------------------------------------------------------------------

    #                        Serialization and deserialization
    # ------------------------------------------------------------------------------
    # Of the GUI presentation options (not the data itself)
    #
    # save/load_gui_options_from_json (Optional)
    # Optional serialization and deserialization of the GUI presentation options
    # (i.e. anything that deals with how the data is presented in the GUI, *not the
    # If provided, these functions will be used to recreate the GUI presentation opt
    # so that the GUI looks the same when the application is restarted.
    save_gui_options_to_json: Callable[[], JsonDict] | None = None
    load_gui_options_from_json: Callable[[JsonDict], None] | None = None

    # Of the data itself
    #
    # Optional serialization and deserialization functions for DataType
    # If provided, these functions will be used to serialize and deserialize the dat
    # If not provided, "value" will be serialized as a dict of its __dict__ attribut
    # or as a json string (for int, float, str, bool, and None)
    save_to_dict: Callable[[DataType], JsonDict] | None = None
    load_from_dict: Callable[[JsonDict], DataType] | None = None
    # ------------------------------------------------------------------------------

    #                        Clipboard
    # ------------------------------------------------------------------------------
    # clipboard_copy_str (Optional)
    # if provided, this function will be called when the value is copied to the clip
    # Used in more advanced cases, when the data is not a simple string, or when pre
    clipboard_copy_str: Callable[[DataType], str] | None = None

    # clipboard_copy_possible (Optional)
    # True by default
    # If False, the user can not copy the data to the clipboard
    clipboard_copy_possible: bool = True
    # ------------------------------------------------------------------------------

    def __init__(self) -> None:
        self.validate_value = []
```

## Architecture

Below is a PlantUML diagram showing the architecture of the `fiat_core` module. See the architecture page for the full architecture diagrams.

```
from fiatlight.fiat_notebook import plantuml_magic
%plantuml_include class_diagrams/fiat_core.puml
```


fiat_core
This is the core of fiatlight.
It is a set of classes that can be used to add a GUI
to any data, function or graph of functions.
It does depend on ImGui, but not on
imgui-node-editor.
FunctionsGraph
A graph of FunctionNodes
functions_nodes
functions_nodes_links
many
many
FunctionNode
function_with_gui: FunctionWithGui
output_links: list[FunctionNodeLink]
input_links: list[FunctionNodeLink]
several
FunctionWithGui
Wraps a function with a GUI
constructor(f)
(will fill _inputs_with_gui and _outputs_with_gui,
attempting to guess the types)
invoke()
...
name: str = ""
Linked function
_f_impl: Callable[..., Any] | None = None
Members linked to the function
_inputs_with_gui: List[ParamWithGui[Any]]
_outputs_with_gui: List[OutputWithGui[Any]]
_last_exception_message: Optional[str] = None
_last_exception_traceback: Optional[str] = None
_dirty: bool = True
Behavioral Flags
invoke_async: bool = False
invoke_manually: bool = False
invoke_always_dirty: bool = False
Optional callbacks
internal_state_gui: BoolFunction | None = None
on_heartbeat: BoolFunction | None = None
FunctionNodeLink
src_function_node: FunctionNode
src_output_idx: int
dst_function_node: FunctionNode
dst_input_name: str
many
1 or many
«DataType»
ParamWithGui
name: str
data_with_gui: AnyDataWithGui[DataType]
default_value: DataType | Unspecified
«DataType»
OutputWithGui
data_with_gui: AnyDataWithGui[DataType]
«DataType»
AnyDataWithGui
a class to wrap any data with a GUI
_value: DataType | Unspecified | Error
callbacks: AnyDataGuiCallbacks[DataType]
save_to_json() / load_from_json()




# FunctionsGraph

`FunctionsGraph` is one of the core classes of FiatLight: it represents a graph of functions, where the output of one function can be linked to the input of another function.

- **Source**: see its full code online
- **Manual**: FunctionsGraph API

## Signature

Below, you will find the "signature" of the `FunctionsGraph` class, with its main attributes and methods (but not their bodies)

Its full source code is available online.

```
from fiatlight.fiat_notebook import look_at_code
%look_at_class_header fiatlight.fiat_core.FunctionsGraph
```

```python
class FunctionsGraph:
    """A graph of FunctionNodes

     `FunctionsGraph` is one of the core classes of FiatLight: it represents a graph
     where the output of one function can be linked to the input of another functior

     See its [full code](../fiat_core/functions_graph.py).

    It contains a graph of FunctionNodes modeled as a list of FunctionNode and a lis
    (which are the links between the outputs of a FunctionNode and the inputs of anc

    This class only stores the data representation of the graph, and does not deal w
    (for this, see FunctionGraphGui)

    This class is not meant to be instantiated directly. Use the factory methods ins

    Public Members
    ==============
    # the list of FunctionNode in the graph
    functions_nodes: list[FunctionNode]
    # the list of links between the FunctionNode
    functions_nodes_links: list[FunctionNodeLink]

    """
    functions_nodes: list[FunctionNode]
    functions_nodes_links: list[FunctionNodeLink]
    _secret_key: str = 'FunctionsGraph'

    class _Construction_Section:
        """
        # ================================================================================
        #                                          Construction (Empty)
        # ================================================================================
        """
        pass

    def __init__(self, secret_key: str='FunctionsGraph') -> None:
        """This class should not be instantiated directly. Use the factory methods i
        pass

    @staticmethod
    def create_empty() -> 'FunctionsGraph':
        """Create an empty FunctionsGraph"""
        pass

    class _Public_API_Add_Function_Section:
        """
        # ================================================================================
        #                                          Public API / Add functions
        #
        # --------------------------------------------------------------------------------
        # Notes:
        #   You can add either Functions or FunctionWithGui
```

```
    #     - If f is a FunctionWithGui, it will be added as is
    #     - If f is a standard function:
    #         - it will be wrapped in a FunctionWithGui
    #         - the function signature *must* mention the types of the parameter
    # ================================================================================
    """
    pass

@staticmethod
def from_function(f: Function | FunctionWithGui) -> 'FunctionsGraph':
    """Create a FunctionsGraph from a single function, either a standard functio
    pass

@staticmethod
def from_function_composition(functions: Sequence[Function | FunctionWithGui]) -
    """Create a FunctionsGraph from a list of functions that will be chained tog
    i.e. the output[0] of one function will be the input[0] of the next functior
    """
    pass

def add_function_composition(self, functions: Sequence[Function | FunctionWithGu
    """Add a list of functions that will be chained together"""
    pass

def add_function(self, f: Function | FunctionWithGui) -> FunctionNode:
    """Add a function to the graph. It will not be linked to any other function.
    pass

def add_gui_node(self, gui_function: GuiFunctionWithInputs, label: str | None=No
    pass

def add_task_node(self, task_function: GuiFunctionWithInputs, label: str | None=
    pass

def add_markdown_node(self, md_string: str, label: str='Documentation', text_wid
    pass

class _Private_API_Add_Function_Section:
    """
    # ================================================================================
    #                                          Private API / Add functions
    # ================================================================================
    """
    pass

def _add_function_with_gui(self, f_gui: FunctionWithGui) -> FunctionNode:
    pass

def _add_function(self, f: Function) -> FunctionNode:
    pass

@staticmethod
def _create_from_function_composition(functions: Sequence[Function | FunctionWit
    """Create a FunctionsGraph from a list of PureFunctions([InputType] -> Outpu
```

```
    * They should all be pure functions
    * The output[0] of one should be the input[0] of the next
    """
    pass

class _Graph_Manipulation_Section:
    """
    # ================================================================================
    #                                          Graph manipulation
    # ================================================================================
    """
    pass

def _can_add_link(self, src_function_node: FunctionNode, dst_function_node: Func
    """Check if a link can be added between two functions. (private)"""
    pass

def _add_link_from_function_nodes(self, src_function_node: FunctionNode, dst_fur
    """Add a link between two functions nodes (private)"""
    pass

def add_link(self, src_function: str | Function | FunctionWithGui, dst_function:
    """Add a link between two functions, which are identified by their *unique*

    If a graph reuses several times the same function "f",
    the unique names for this functions will be "f_1", "f_2", "f_3", etc.
    """
    pass

def merge_graph(self, other: 'FunctionsGraph') -> None:
    """Merge another FunctionsGraph into this one"""
    pass

def function_with_gui_of_name(self, name: str | None=None) -> FunctionWithGui:
    """Get the function with the given unique name"""
    pass

def _would_add_cycle(self, new_link: FunctionNodeLink) -> bool:
    """Check if adding a link would create a cycle (private)"""
    pass

def has_cycle(self) -> bool:
    """Returns True if the graph has a cycle"""
    pass

def _has_cycle_from_node(self, fn: FunctionNode, path: Set[FunctionNode] | None=
    """Check if there is a cycle starting from a given node (private)"""
    pass

def _remove_link(self, link: FunctionNodeLink) -> None:
    """Remove a link between two functions (private)"""
    pass

def _remove_function_node(self, function_node: FunctionNode) -> None:
```

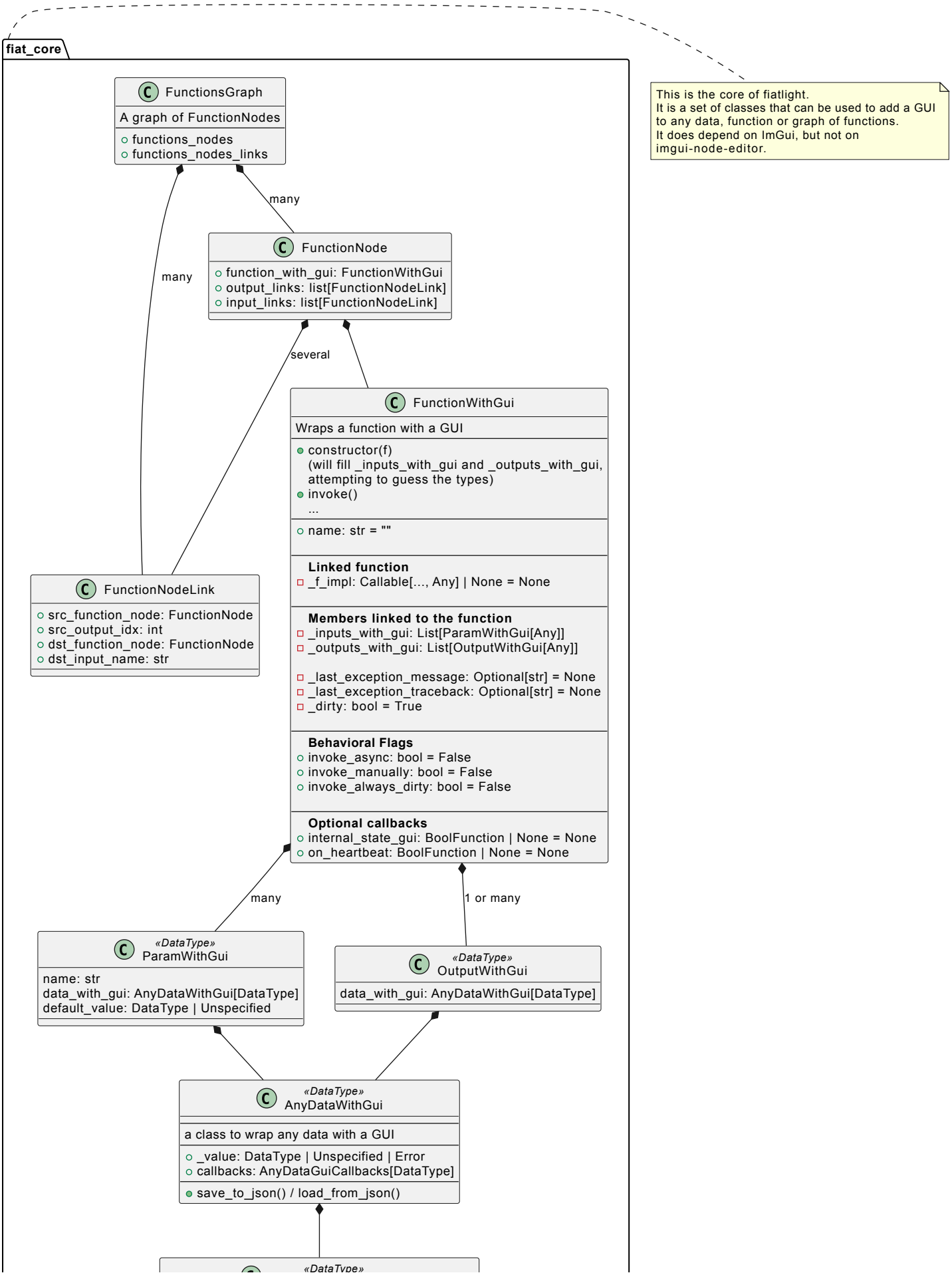
```
    """Remove a function node from the graph (private)"""
    pass

class _Utilities_Section:
    """
    # ============================================================================
    #                                        Utilities
    # ============================================================================
    """
    pass

def function_node_unique_name(self, function_node: FunctionNode) -> str:
    """Return the unique name of a function node:
    If a graph reuses several times the same function "f",
    the unique names for this functions will be "f_1", "f_2", "f_3", etc.
    """
    pass

def _function_node_with_name_or_is_function(self, name_or_function: str | Functi
    """Get the function node with the given name or function"""
    pass

def _function_node_with_unique_name(self, function_name: str) -> FunctionNode:
    """Get the function with the unique name"""
    pass

def all_function_nodes_with_unique_names(self) -> Dict[str, FunctionNode]:
    """Return a dict of all the function nodes, with their unique names as keys
    pass

def shall_display_refresh_needed_label(self) -> bool:
    """Returns True if any function node shall display a "Refresh needed" label'
    pass

class _Serialization_Section:
    """
    # ============================================================================
    #                                        Serialization
    # Note:  save_gui_options_to_json() and load_gui_options_from_json()
    #        are intentionally not implemented here
    #        See FunctionsGraphGui (which does deals with the GUI)
    # ============================================================================
    """
    pass

def save_user_inputs_to_json(self) -> JsonDict:
    """Saves the user inputs, i.e. the functions params that are editable in the
    (this excludes the params that are set by the links between the functions)"'
    pass

def load_user_inputs_from_json(self, json_data: JsonDict) -> None:
    """Restores the user inputs from a json dict"""
    pass
```

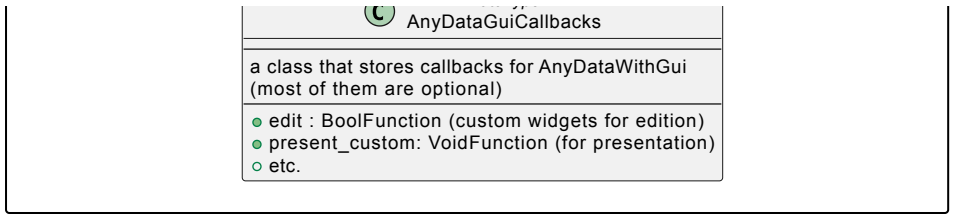
```
    def save_graph_composition_to_json(self) -> JsonDict:
        """Saves the graph composition to a json dict.
        Only used when the graph composition is editable.
        """
        pass

    def load_graph_composition_from_json(self, json_data: JsonDict, function_factory
        """Loads the graph composition from a json dict."""
        pass
```

## Architecture

Below is a PlantUML diagram showing the architecture of the `fiat_core` module. See the architecture page for the full architecture diagrams.

```
from fiatlight.fiat_notebook import plantuml_magic
%plantuml_include class_diagrams/fiat_core.puml
```


fiat_core
This is the core of fiatlight.
It is a set of classes that can be used to add a GUI
to any data, function or graph of functions.
It does depend on ImGui, but not on
imgui-node-editor.
FunctionsGraph
A graph of FunctionNodes
functions_nodes
functions_nodes_links
many
many
FunctionNode
function_with_gui: FunctionWithGui
output_links: list[FunctionNodeLink]
input_links: list[FunctionNodeLink]
several
FunctionWithGui
Wraps a function with a GUI
constructor(f)
(will fill _inputs_with_gui and _outputs_with_gui,
attempting to guess the types)
invoke()
...
name: str = ""
Linked function
_f_impl: Callable[..., Any] | None = None
Members linked to the function
_inputs_with_gui: List[ParamWithGui[Any]]
_outputs_with_gui: List[OutputWithGui[Any]]
_last_exception_message: Optional[str] = None
_last_exception_traceback: Optional[str] = None
_dirty: bool = True
Behavioral Flags
invoke_async: bool = False
invoke_manually: bool = False
invoke_always_dirty: bool = False
Optional callbacks
internal_state_gui: BoolFunction | None = None
on_heartbeat: BoolFunction | None = None
FunctionNodeLink
src_function_node: FunctionNode
src_output_idx: int
dst_function_node: FunctionNode
dst_input_name: str
many
1 or many
«DataType»
ParamWithGui
name: str
data_with_gui: AnyDataWithGui[DataType]
default_value: DataType | Unspecified
«DataType»
OutputWithGui
data_with_gui: AnyDataWithGui[DataType]
«DataType»
AnyDataWithGui
a class to wrap any data with a GUI
_value: DataType | Unspecified | Error
callbacks: AnyDataGuiCallbacks[DataType]
save_to_json() / load_from_json()




# Fiatlight Kits

Fiatlight offers several kits adapted to different domains.

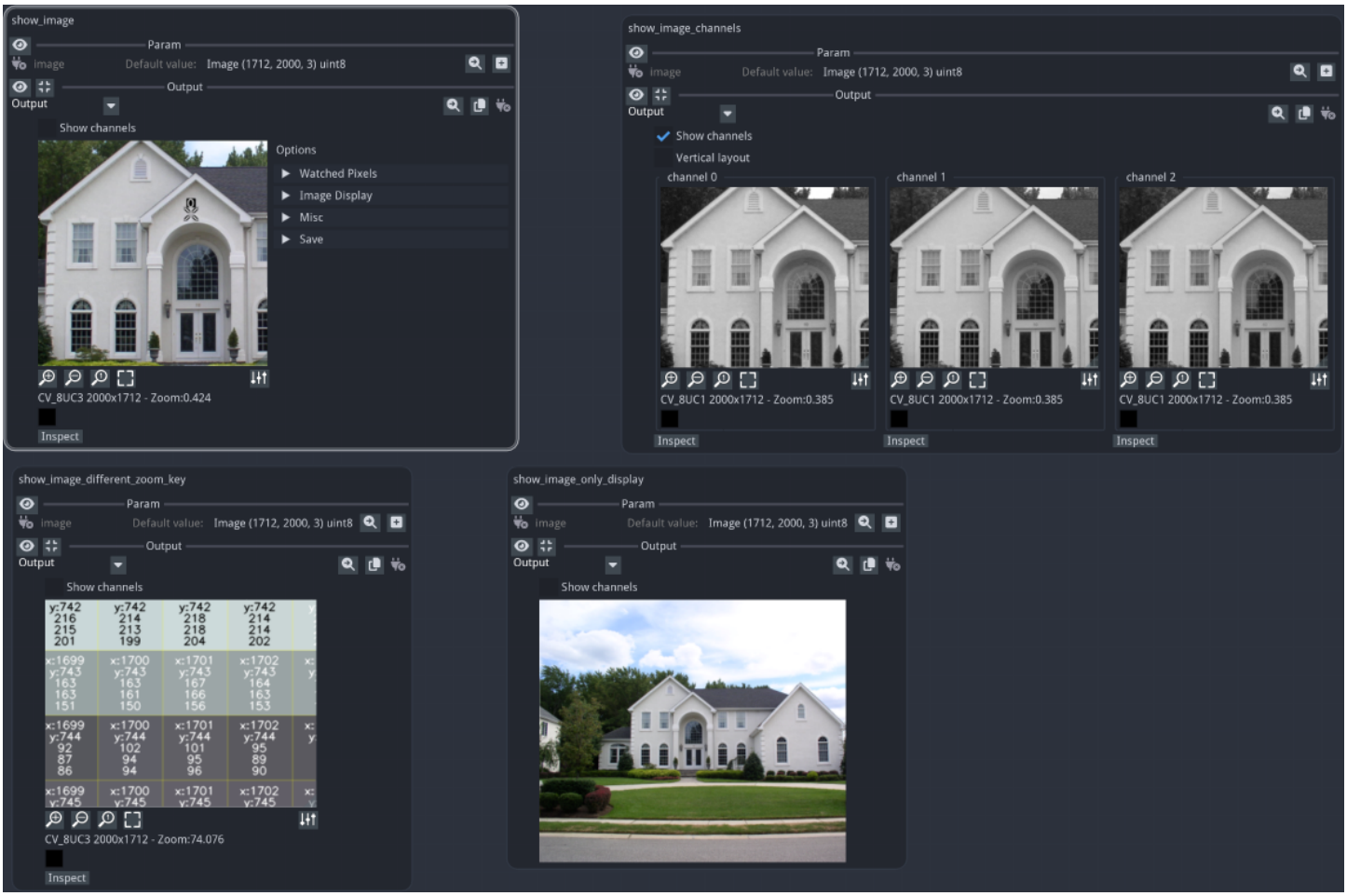
- `fiat_image`: advanced image widget
- `fiat_matplotlib`: widget to view matplotlib plots (zoomable)
- `fiat_dataframe`: widget to explore pandas dataframes
- `fiat_implot`: widget to explore 1D and 2D numpy arrays with ImPlot

## fiat_image: advanced image widget

Fiatlight provides an advanced image viewer and analyzer which enables to zoom, pan, look at pixel values and sync the zoom across images.

### Example

```
from fiatlight.fiat_kits.fiat_image import fiat_image_attrs_demo
fiat_image_attrs_demo.main()
```




- In the “show_image” output, the options panel was opened
- The “show_image_channels” output shows the image channels, and it zoom/pan is linked to “show_image”
- The “show_image_different_zoom_key” image has a different zoom key, and the zoom/pan is not linked to “show_image”. It also zoomed at a high-level, so that pixel values are displayed.
- the “show_image_only_display” image is displayed, and cannot be zoomed or panned (the widget may be resized however)

## Fiat attributes available for the ImageWithGui widget

The image widget provided with fiat_image is extremely customizable. Here is a list of all the possible customizations options:

```
%%bash
fiatlight gui ImageWithGui
```

GUI type: ImageWithGui
======================
A highly sophisticated GUI for displaying and analysing images. Zoom/Pan, show cha

Available custom attributes for fiat_image.ImageWithGui:

| Name | Type | Default | Explanation |
|---|---|---|---|
| | | | **Main attribute |
| only_display | bool | False | Only display the zoom, no pan |
| image_display_size | tuple[int, int] | (200, 0) | Initial size of height). One of |
| zoom_key | str | z | Key to zoom in same zoom key w |
| is_color_order_bgr | bool | True | Color order is uses BGR by defa |
| can_resize | bool | True | Can resize the the bottom righ |
| | | | **Channels** |
| show_channels | bool | False | Show channels |
| channel_layout_vertically | bool | False | Layout channels |
| | | | **Zoom & Pan** |
| pan_with_mouse | bool | True | Pan with mouse |
| zoom_with_mouse_wheel | bool | True | Zoom with mouse |
| | | | **Info displaye |
| show_school_paper_background | bool | True | Show school pap is unzoomed |
| show_alpha_channel_checkerboard | bool | True | Show alpha chan |
| show_grid | bool | True | Show grid with |
| draw_values_on_zoomed_pixels | bool | True | Draw values on |
| | | | **Info displaye |
| show_image_info | bool | True | Show image info |

| show_pixel_info | bool | True | Show pixel info
| | | | position under
| | | | **Control butto
| show_zoom_buttons | bool | True | Show zoom butto
| show_options_panel | bool | True | Show options pa
| show_options_button | bool | True | Show options bu
| show_inspect_button | bool | True | Show the inspec
| | | | a large version
| | | | Inspector

Available custom attributes for AnyDataWithGui Generic attributes:

| Name | Type | Default | Explanation |
|---|---|---|---|
| | | | **Generic attributes** |
| validate_value | object | None | Function to validate a paramete return DataValidationResult.ok( |
| label | str | | A label for the parameter. If e function parameter name is used |
| tooltip | str | | An optional tooltip to be displ |
| label_color | ImVec4 | ImVec4(0.000000, 0.000000, 0.000000, 1.000000) | The color of the label (will us text color if not provided) |

Code to test this GUI type:

```python
import typing
import fiatlight

@fiatlight.with_fiat_attributes(
    #  Main attributes for the image viewer
    union_param__only_display = False,
    union_param__image_display_size = (200, 0),
    union_param__zoom_key = "z",
    union_param__is_color_order_bgr = True,
    union_param__can_resize = True,
    #  Channels
    union_param__show_channels = False,
    union_param__channel_layout_vertically = False,
    #  Zoom & Pan
    union_param__pan_with_mouse = True,
```

```
    union_param__zoom_with_mouse_wheel = True,
    #  Info displayed on image
    union_param__show_school_paper_background = True,
    union_param__show_alpha_channel_checkerboard = True,
    union_param__show_grid = True,
    union_param__draw_values_on_zoomed_pixels = True,
    #  Info displayed under the image
    union_param__show_image_info = True,
    union_param__show_pixel_info = True,
    #  Control buttons under the image
    union_param__show_zoom_buttons = True,
    union_param__show_options_panel = True,
    union_param__show_options_button = True,
    union_param__show_inspect_button = True,
    #  Generic attributes
    union_param__validate_value = None,
    union_param__label = "",
    union_param__tooltip = "",
    union_param__label_color = ImVec4(0.000000, 0.000000, 0.000000, 1.000000))
def f(union_param: typing.Union[fiatlight.fiat_kits.fiat_image.image_types.ImageU8_
    return union_param

fiatlight.run(f)
```

## Image types

Fiatlight provides several synonyms for Numpy arrays that denote different types of images. Each of these types will be displayed by the image widget.

```
import fiatlight
from fiatlight.fiat_notebook import look_at_code
%look_at_python_file fiat_kits/fiat_image/image_types.py
```

```python
"""This module defines several types you can use to annotate your functions.
The image types are defined as NewType instances, which are just aliases for numpy a

All those types will be displayed in the GUI as images, using the ImmVision image vi
(https://github.com/pthom/immvision)

Notes:
    - The easiest way to display an image is to use the `Image` type, which is a uni
      or to use the `ImageU8` type, which is a union of all UInt8 image types.
    - any numpy array can be used to create an `Image`, and the viewer will try to c
"""

from typing import Any, NewType
import numpy as np
from typing import Tuple, Union

# Define shape types for clarity
ShapeHeightWidth = Tuple[int, int]
ShapeHeightWidthChannels = Tuple[int, int, int]

# Define UInt8 as a dtype for numpy arrays
UInt8 = np.dtype[np.uint8]
AnyFloat = np.dtype[np.floating[Any]]


#
# UInt8 Images
#
# ImageU8 = NewType("ImageU8", np.ndarray[ShapeHeightWidthChannels | ShapeHeightWidt
# Type definitions for UInt8 images based on channel count
ImageU8_1 = NewType("ImageU8_1", np.ndarray[ShapeHeightWidth, UInt8])
ImageU8_2 = NewType("ImageU8_2", np.ndarray[ShapeHeightWidthChannels, UInt8])
ImageU8_3 = NewType("ImageU8_3", np.ndarray[ShapeHeightWidthChannels, UInt8])
ImageU8_4 = NewType("ImageU8_4", np.ndarray[ShapeHeightWidthChannels, UInt8])
ImageU8_WithNbChannels = Union[ImageU8_1, ImageU8_2, ImageU8_3, ImageU8_4]
# Type definitions based on the roles of the channels
ImageU8_RGB = NewType("ImageU8_RGB", ImageU8_3)
ImageU8_RGBA = NewType("ImageU8_RGBA", ImageU8_4)
ImageU8_BGRA = NewType("ImageU8_BGRA", ImageU8_4)
ImageU8_BGR = NewType("ImageU8_BGR", ImageU8_3)
ImageU8_GRAY = NewType("ImageU8_GRAY", ImageU8_1)
ImageU8_WithChannelsRoles = Union[ImageU8_RGB, ImageU8_RGBA, ImageU8_BGRA, ImageU8_E

# Generic type for any 8-bit image
ImageU8 = Union[ImageU8_WithNbChannels, ImageU8_WithChannelsRoles]


#
# Float Images
#
# Type definitions for float images based on channel count
ImageFloat_1 = NewType("ImageFloat_1", np.ndarray[ShapeHeightWidth, AnyFloat])
ImageFloat_2 = NewType("ImageFloat_2", np.ndarray[ShapeHeightWidthChannels, AnyFloat
```

```
ImageFloat_3 = NewType("ImageFloat_3", np.ndarray[ShapeHeightWidthChannels, AnyFloat
ImageFloat_4 = NewType("ImageFloat_4", np.ndarray[ShapeHeightWidthChannels, AnyFloat

# Generic type for any float image
ImageFloat = Union[ImageFloat_1, ImageFloat_2, ImageFloat_3, ImageFloat_4]


#
# Generic Image Type
#
# Image is a union of all image types
Image = Union[ImageU8, ImageFloat]


# ---------------------------- Register image type factories ----------------------


def _register_image_type_factories() -> None:
    from fiatlight.fiat_togui.gui_registry import gui_factories
    from fiatlight.fiat_kits.fiat_image.image_gui import ImageWithGui

    prefix = "fiatlight.fiat_kits.fiat_image.image_types.Image"
    gui_factories().register_factory_name_start_with(prefix, ImageWithGui)
    gui_factories().register_factory_union(prefix, ImageWithGui)
```

`

## Source code for the example

```
%look_at_python_file fiat_kits/fiat_image/fiat_image_attrs_demo.py
```

```python
"""Demo how to set custom presentation attributes for the Image Widget (ImageWithGui

Notes:
    - The custom attributes can be set using the decorator fl.with_fiat_attributes
    - In these examples, we intend to set custom attributes for the output of the
      functions, i.e. the returned value.
      As a consequence, the custom attributes are set in the return__...
      arguments of the decorator.
"""

import fiatlight as fl
from fiatlight.fiat_kits.fiat_image import ImageU8_3
import cv2


# Our demo image
demo_image: ImageU8_3 = cv2.imread(fl.demo_assets_dir() + "/images/house.jpg")  # ty


# A simple function that will use the Image Widget with its default settings.
def show_image(image: ImageU8_3 = demo_image) -> ImageU8_3:
    return image


# A function whose output will initially show the channels
# Since it does not specify a zoom key,
# it will be zoomed and panned together with the image
# shown by "show_image"
@fl.with_fiat_attributes(return__show_channels=True)
def show_image_channels(image: ImageU8_3 = demo_image) -> ImageU8_3:
    return image


# A function whose output will have a different zoom key:
# it can be panned and zoomed, independently of the other images
@fl.with_fiat_attributes(return__zoom_key="other")
def show_image_different_zoom_key(image: ImageU8_3 = demo_image) -> ImageU8_3:
    return image


# A function that will use the Image Widget with custom attributes:
# - the image is displayed only (it cannot be zoomed or panned,
#   and the pixel values are not shown)
# - the image is displayed with a height of 300 pixels
#   (the width is automatically calculated)
# - the image cannot be resized
@fl.with_fiat_attributes(
    return__only_display=True,
    return__image_display_size=(0, 300),
    return__can_resize=False,
)
def show_image_only_display(image: ImageU8_3 = demo_image) -> ImageU8_3:
    return image
```

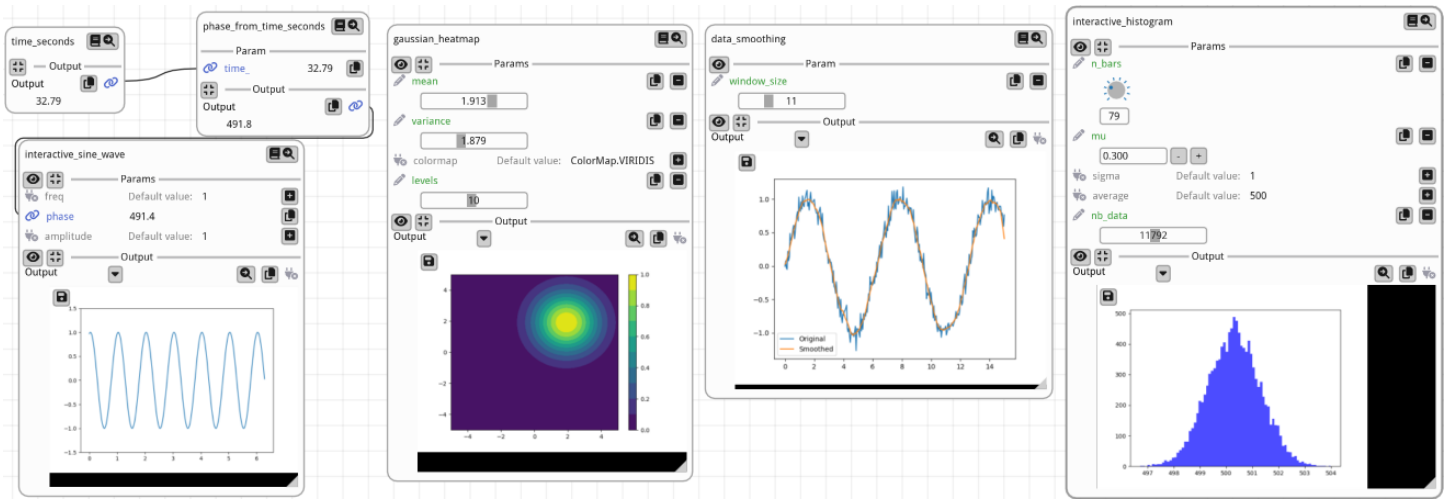
```
def main() -> None:
    graph = fl.FunctionsGraph()
    graph.add_function(show_image)
    graph.add_function(show_image_channels)
    graph.add_function(show_image_different_zoom_key)
    graph.add_function(show_image_only_display)

    fl.run(graph, app_name="fiat_image_fiat_attrs_demo")


if __name__ == "__main__":
    main()
```

# fiat_matplotlib: display matplotlib figures

Fiatlight provides `FigureWithGui`, a viewer for Matplotlib figures.

## Example

```
from fiatlight.fiat_kits.fiat_matplotlib import demo_matplotlib

demo_matplotlib.main()
```




## Fiat attributes available for the FigureWithGui widget

The FigureWithGui widget is not customizable. However, it can be zoomed by the user and this setting will be saved.

## Source code for the example

```
import fiatlight
from fiatlight.fiat_notebook import look_at_code  # noqa
%look_at_python_file fiat_kits/fiat_matplotlib/demo_matplotlib.py
```

```python
"""Interactive Matplotlib Figures with Fiatlight

This example demonstrates several types of matplotlib figures rendered within Fiatli
"""

import matplotlib.pyplot as plt
from matplotlib.figure import Figure
import numpy as np
from enum import Enum
import time
import fiatlight as fl


# Initialize the start time
_start_time = time.time()


def time_seconds() -> float:
    """Returns the time elapsed since the start of the application."""
    return time.time() - _start_time


def phase_from_time_seconds(time_: float) -> float:
    """Calculates the phase from the given time."""
    return time_ * 15.0


# Set the function to always update
time_seconds.invoke_always_dirty = True  # type: ignore


def interactive_sine_wave(freq: float = 1.0, phase: float = 0.0, amplitude: float =
    """Generates an interactive sine wave with adjustable frequency, phase, and ampl
    x = np.linspace(0, 2 * np.pi, 3000)
    y = amplitude * np.sin(2 * np.pi * freq * x + phase)
    fig, ax = plt.subplots()
    ax.plot(x, y)
    ax.set_ylim([-1.5, 1.5])  # Adjust the y-axis limits
    return fig


# Set ranges and edit types for the sine wave parameters
fl.add_fiat_attributes(
    interactive_sine_wave,
    freq__range=(0.1, 3),
    phase__range=(-np.pi, np.pi),
    amplitude__range=(0.1, 2),
    freq__edit_type="knob",
    phase__edit_type="knob",
    amplitude__edit_type="knob",
)
```

```
class ColorMap(Enum):
    VIRIDIS = "viridis"
    PLASMA = "plasma"
    INFERNO = "inferno"
    MAGMA = "magma"
    CIVIDIS = "cividis"


@fl.with_fiat_attributes(
    mean__range=(-5, 5),
    variance__range=(0.1, 5),
    levels__range=(1, 20),
)
def gaussian_heatmap(
    mean: float = 0, variance: float = 1, colormap: ColorMap = ColorMap.VIRIDIS, lev
) -> Figure:
    """Generates a Gaussian heatmap with adjustable mean, variance, colormap, and nu
    x = y = np.linspace(-5, 5, 100)
    X, Y = np.meshgrid(x, y)
    Z = np.exp(-((X - mean) ** 2 + (Y - mean) ** 2) / (2 * variance))
    fig, ax = plt.subplots()
    contour = ax.contourf(X, Y, Z, levels, cmap=colormap.value)
    fig.colorbar(contour, ax=ax)
    return fig


@fl.with_fiat_attributes(window_size__range=(1, 40))
def data_smoothing(window_size: int = 5) -> Figure:
    """Demonstrates data smoothing using a moving average filter."""
    x = np.linspace(0, 15, 300)
    y = np.sin(x) + np.random.normal(0, 0.1, 300)  # Noisy sine wave
    y_smooth = np.convolve(y, np.ones(window_size) / window_size, mode="same")
    fig, ax = plt.subplots()
    ax.plot(x, y, label="Original")
    ax.plot(x, y_smooth, label="Smoothed")
    ax.legend()
    return fig


def interactive_histogram(
    n_bars: int = 10, mu: float = 0, sigma: float = 1, average: float = 500, nb_data
) -> Figure:
    """Generates an interactive histogram with adjustable number of bars, mean, and
    data = np.random.normal(mu, sigma, int(nb_data)) + average
    bins = np.linspace(np.min(data), np.max(data), n_bars)
    fig, ax = plt.subplots()
    ax.hist(data, bins=bins, color="blue", alpha=0.7)
    return fig


# Set interactive parameters for the histogram
fl.add_fiat_attributes(
    interactive_histogram,
    n_bars__edit_type="knob",
```

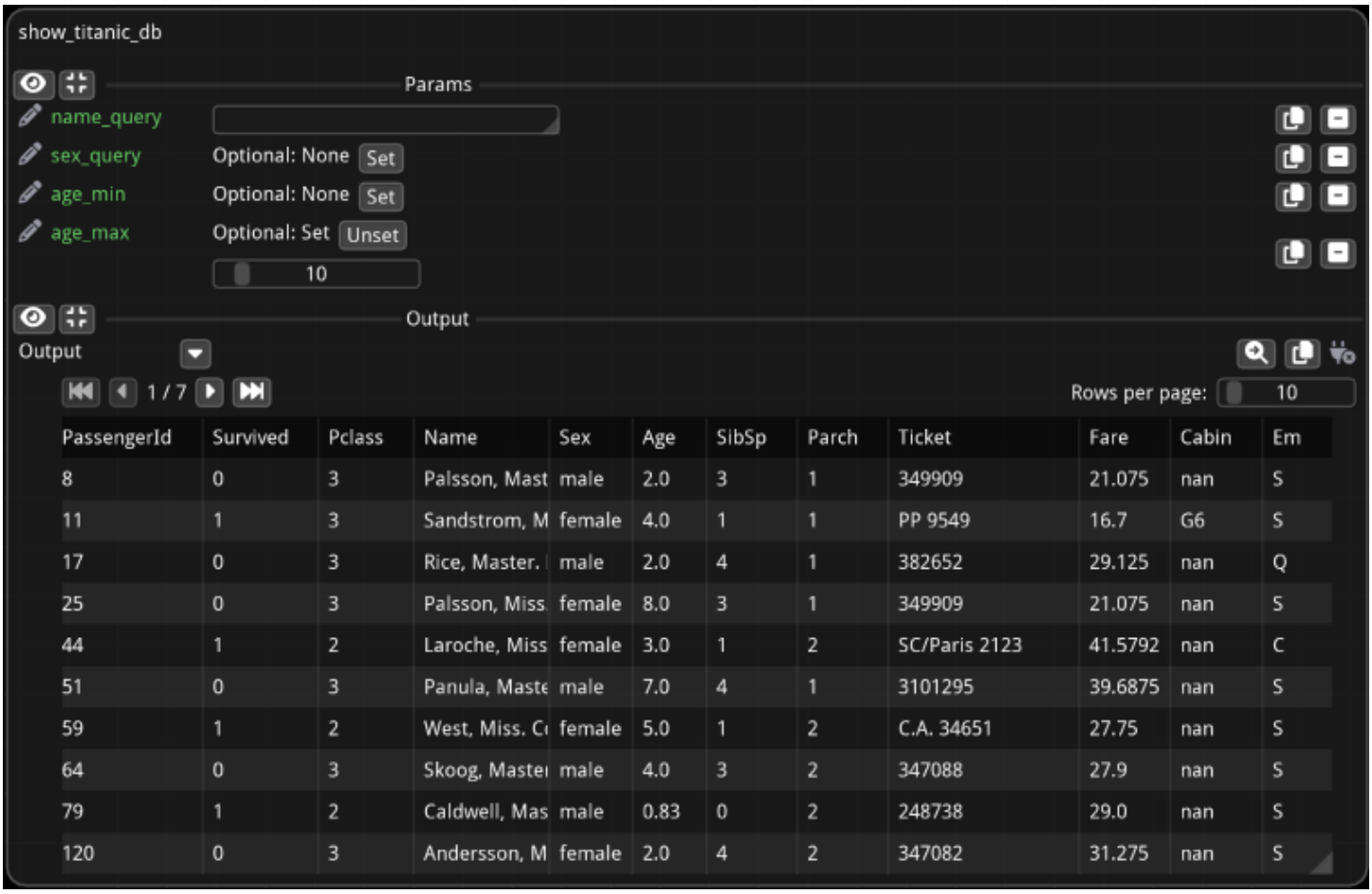
```
    n_bars__range=(1, 300),
    mu__edit_type="input",
    mu__range=(-5, 5),
    sigma__edit_type="drag",
    sigma__range=(0.1, 5),
    average__edit_type="slider_float_any_range",
    nb_data__edit_type="slider",
    nb_data__range=(100, 1_000_000),
    nb_data__slider_logarithmic=True,
    nb_data__slider_no_input=True,
)


def main() -> None:
    """Main function to run the Fiatlight application with interactive matplotlib fi
    import fiatlight

    # Create a graph to manage functions and their links
    graph = fiatlight.FunctionsGraph()
    graph.add_function(interactive_sine_wave)
    graph.add_function(gaussian_heatmap)
    graph.add_function(data_smoothing)
    graph.add_function(interactive_histogram)
    graph.add_function(time_seconds)
    graph.add_function(phase_from_time_seconds)
    graph.add_link("time_seconds", "phase_from_time_seconds", "time_")
    graph.add_link("phase_from_time_seconds", "interactive_sine_wave", "phase")
    fiatlight.run(graph, app_name="figure_with_gui_demo")


if __name__ == "__main__":
    main()
```

# fiat_dataframe: pandas DataFrame explorer

Fiatlight provides `DataFrameWithGui`, a viewer for pandas dataframes that allows to sort, and visualize the data. Composed with the advanced GUI creation capabilities of fiatlight, it can also filter data.

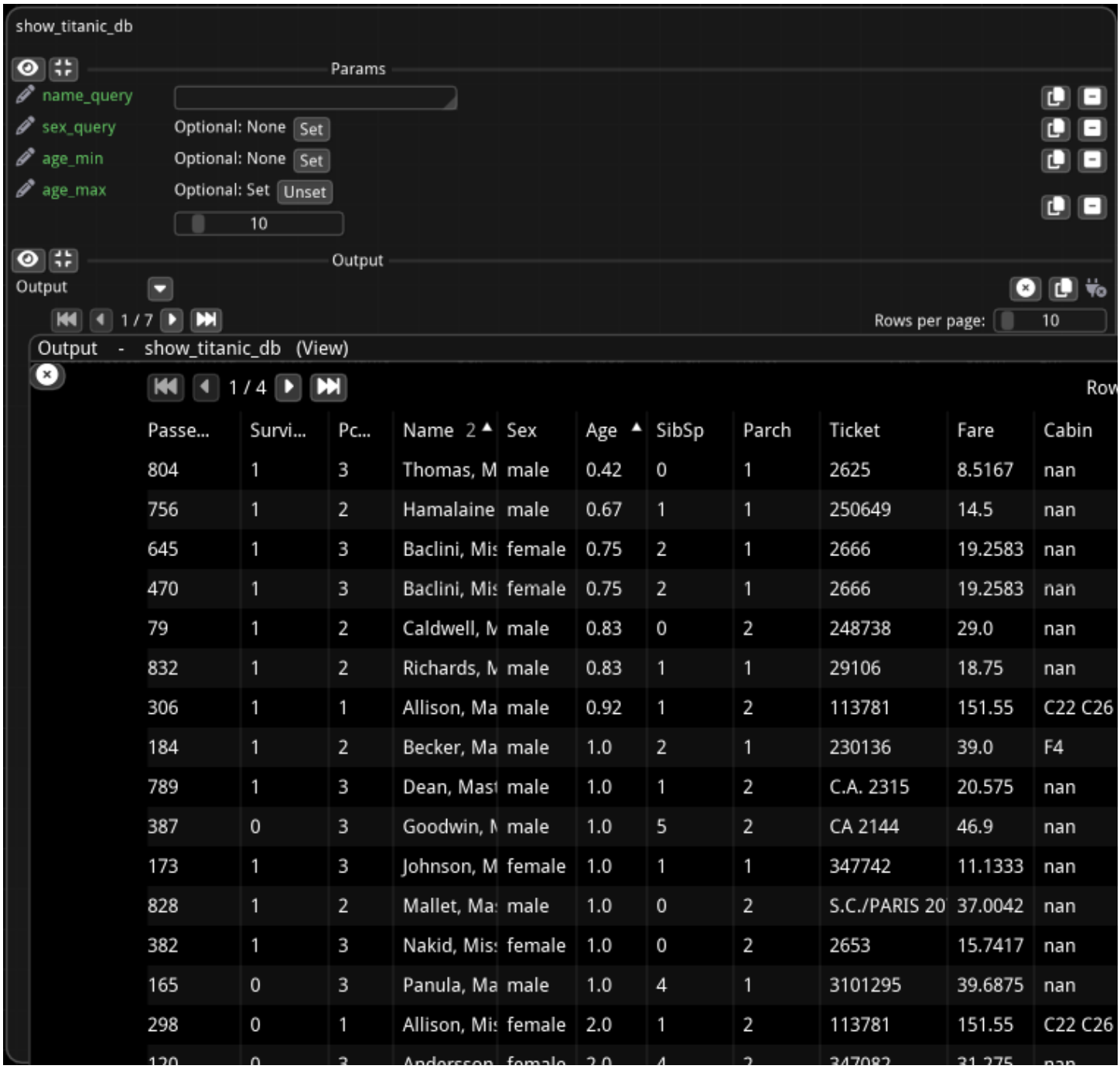
## Example

```
from fiatlight.fiat_kits.fiat_dataframe import dataframe_with_gui_demo_titanic
dataframe_with_gui_demo_titanic.main()
```




By clicking on the magnifier button on top of the dataframe, you can open it in a popup where sorting options are available. Click on one column (or shift-click on multiple columns) to sort the data.

```
dataframe_with_gui_demo_titanic.main()
```




## Fiat attributes available for DataFrameWithGui

Here is a list of all the possible customizations options:

```bash
%%bash
fiatlight gui DataFrameWithGui
```

GUI type: DataFrameWithGui
==========================

  A class to present a pandas DataFrame in the GUI, with pagination and other featu

  Available custom attributes for DataFrameWithGui:

| Name | Type | Default | Explanation |
|---|---|---|---|
| widget_size_em | tuple[float, float] | (50.0, 15.0) | Widget size in em |
| column_widths_em | dict | {} | Dictionary to spe individual column |
| rows_per_page_node | int | 10 | Number of rows to displayed in a fu |
| rows_per_page_classic | int | 20 | Number of rows to displayed in a po |
| current_page_start_idx | int | 0 | Index of the firs used for paginati |

  Available custom attributes for AnyDataWithGui Generic attributes:

| Name | Type | Default | Explanation |
|---|---|---|---|
| | | | **Generic attributes** |
| validate_value | object | None | Function to validate a paramete return DataValidationResult.ok( |
| label | str | | A label for the parameter. If e function parameter name is used |
| tooltip | str | | An optional tooltip to be displ |
| label_color | ImVec4 | ImVec4(0.000000, 0.000000, 0.000000, 1.000000) | The color of the label (will us text color if not provided) |

Code to test this GUI type:
---------------------------
```python
import typing
import fiatlight

@fiatlight.with_fiat_attributes(
    dataframe_param__widget_size_em = (50.0, 15.0),
    dataframe_param__column_widths_em = {},
```

```
    dataframe_param__rows_per_page_node = 10,
    dataframe_param__rows_per_page_classic = 20,
    dataframe_param__current_page_start_idx = 0,
    #  Generic attributes
    dataframe_param__validate_value = None,
    dataframe_param__label = "",
    dataframe_param__tooltip = "",
    dataframe_param__label_color = ImVec4(0.000000, 0.000000, 0.000000, 1.000000))
def f(dataframe_param: pandas.core.frame.DataFrame) -> pandas.core.frame.DataFrame:
    return dataframe_param

fiatlight.run(f)
```

## Source code for the example

```python
import fiatlight
from fiatlight.fiat_notebook import look_at_code  # noqa
%look_at_python_file fiat_kits/fiat_dataframe/dataframe_with_gui_demo_titanic.py
```

```python
import fiatlight as fl
import pandas as pd
from enum import Enum


def make_titanic_df() -> pd.DataFrame:
    # Here, we provide an example data frame to the user,
    # using the Titanic dataset from the Data Science Dojo repository.
    # (widely used in data science tutorials)
    url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic
    try:
        df = pd.read_csv(url)
    except Exception as e:
        print(f"Error loading sample dataset: {e}")
        df = pd.DataFrame()  # Return an empty DataFrame in case of failure
    return df


class Sex(Enum):
    Man = "male"
    Woman = "female"


@fl.with_fiat_attributes(
    # define the custom attributes for the function parameters
    age_min__range=(0, 100),
    age_max__range=(0, 100),
    # define custom attributes for the function output
    # (i.e. the presentation options for the DataFrame)
    return__widget_size_em=(55.0, 15.0),
    return__rows_per_page_node=10,
    return__rows_per_page_popup=20,
    return__column_widths_em={"Name": 5},
)
def show_titanic_db(
    name_query: str = "", sex_query: Sex | None = None, age_min: int | None = None,
) -> pd.DataFrame:
    dataframe = make_titanic_df()
    if dataframe.empty:
        return dataframe

    # filter dataframe
    if name_query:
        dataframe = dataframe[dataframe["Name"].str.contains(name_query, case=False)
    if sex_query:
        dataframe = dataframe[dataframe["Sex"] == sex_query.value]
    if age_min is not None:
        dataframe = dataframe[dataframe["Age"] >= age_min]
    if age_max is not None:
        dataframe = dataframe[dataframe["Age"] <= age_max]

    return dataframe
```

```python
def main() -> None:
    fl.run(show_titanic_db, app_name="dataframe_with_gui_demo_titanic")


if __name__ == "__main__":
    main()
```

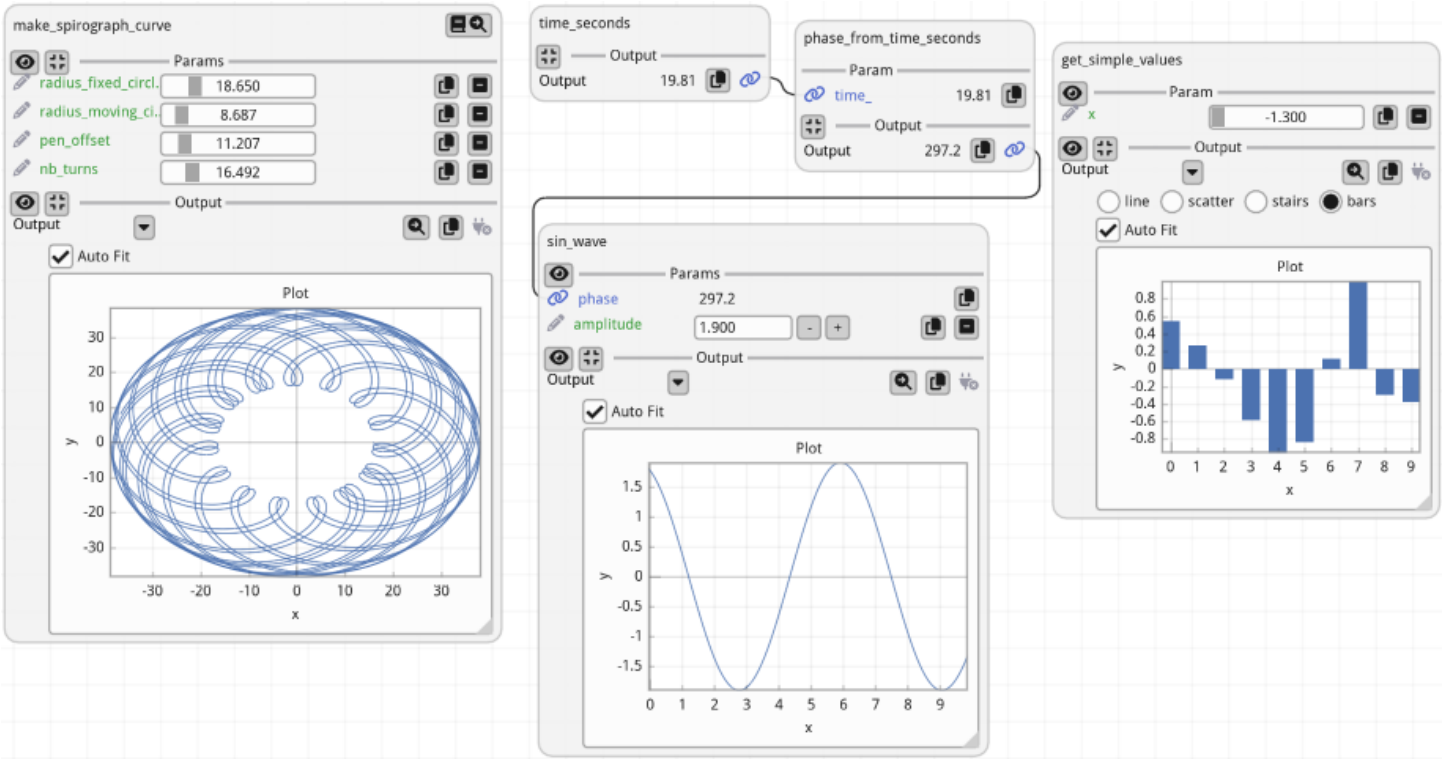
# fiat_implot: widget for 1D and 2D numpy arrays

Fiatlight provides `SimplePlotGui`, a viewer for numpy arrays that allows to plot 1D and 2D arrays with ImPlot

> - ImPlot is a very capable and fast plotting library, not limited to simple 1D and 2D plots. It is available with Fiatlight and ImGui Bundle (on which Fiatlight is based). See online demo of ImPlot for more examples.
> - It is faster than Matplotlib within Fiatlight, and well adapted for real time plots (can refresh at 120FPS +)

## Example

```python
from fiatlight.fiat_kits.fiat_implot import demo_implot

demo_implot.main()
```




# Fiat attributes available for SimplePlotGui

**Here is a list of all the type handled by SimplePlotGui:**

```
%%bash
fiatlight types FloatMatrix_Dim
```

```
+-----------------------------------------------------+-----------------------------
| Data Type                                           | Gui Type
+=====================================================+=============================
| fiatlight.fiat_kits.fiat_implot.array_types.FloatM | fiatlight.fiat_kits.fiat_imp
| atrix_Dim1                                          |   A GUI for presenting 1D or
|   synonym for a 1D ndarray of floats (NewType)      | array as a line, scatter (+ s
|                                                     | small enough)
+-----------------------------------------------------+-----------------------------
| fiatlight.fiat_kits.fiat_implot.array_types.FloatM | fiatlight.fiat_kits.fiat_imp
| atrix_Dim2                                          |   A GUI for presenting 1D or
|   synonym for a 2D ndarray of floats (NewType)      | array as a line, scatter (+ s
|                                                     | small enough)
+-----------------------------------------------------+-----------------------------
```

**Here is a list of all the possible customizations options:**

```bash
%%bash
fiatlight gui SimplePlotGui
```

GUI type: SimplePlotGui
=======================

A GUI for presenting 1D or 2D arrays with ImPlot. Can present the array as a line

Available custom attributes for SimplePlotGui:

| Name | Type | Default | Explanation |
|---|---|---|---|
| plot_type | str | line | The type of presen line, scatter, sta |
| plot_size_em | tuple[float, float] | (35.0, 20.0) | Size in em units ( height) |
| auto_fit | bool | True | Auto-scale the plo |
| small_array_threshold | int | 100 | The threshold for present scatter, b |

Available custom attributes for AnyDataWithGui Generic attributes:

| Name | Type | Default | Explanation |
|---|---|---|---|
| | | | **Generic attributes** |
| validate_value | object | None | Function to validate a paramete return DataValidationResult.ok( |
| label | str | | A label for the parameter. If e function parameter name is used |
| tooltip | str | | An optional tooltip to be displ |
| label_color | ImVec4 | ImVec4(0.000000, 0.000000, 0.000000, 1.000000) | The color of the label (will us text color if not provided) |

Code to test this GUI type:
---------------------------

```python
import typing
import fiatlight

@fiatlight.with_fiat_attributes(
    floatmatrix_param__plot_type = "line",
    floatmatrix_param__plot_size_em = (35.0, 20.0),
    floatmatrix_param__auto_fit = True,
    floatmatrix_param__small_array_threshold = 100,
    #  Generic attributes
```

```
    floatmatrix_param__validate_value = None,
    floatmatrix_param__label = "",
    floatmatrix_param__tooltip = "",
    floatmatrix_param__label_color = ImVec4(0.000000, 0.000000, 0.000000, 1.000000)
def f(floatmatrix_param: fiatlight.fiat_kits.fiat_implot.array_types.FloatMatrix) -
    return floatmatrix_param

fiatlight.run(f)
```

## Source code for the example

```python
import fiatlight
from fiatlight.fiat_notebook import look_at_code  # noqa
%look_at_python_file fiat_kits/fiat_implot/demo_implot.py
```

```python
"""Demonstrates plots generated using ImPlot (https://github.com/epezent/implot). In

This example demonstrates
- how to create a live sine wave plot with adjustable frequency, phase, and amplitud
  The frequency, phase, and amplitude can be adjusted interactively using knobs.
- how to create a spirograph-like curve using ImPlot.
"""

from fiatlight import fiat_implot
import fiatlight as fl
import numpy as np
import math
import time

_start_time = time.time()


def time_seconds() -> float:
    return time.time() - _start_time


def phase_from_time_seconds(time_: float) -> float:
    return time_ * 15.0


time_seconds.invoke_always_dirty = True  # type: ignore


def sin_wave(phase: float, amplitude: float = 1.0) -> fiat_implot.FloatMatrix_Dim2:
    x = np.arange(0, 10, 0.1)
    y = np.sin(x + phase) * amplitude
    r = np.stack((x, y))
    return r  # type: ignore


@fl.with_fiat_attributes(
    radius_fixed_circle__range=(0.0, 100.0),
    radius_moving_circle__range=(0.0, 100.0),
    pen_offset__range=(0.0, 100.0),
    nb_turns__range=(0.0, 100.0),
)
def make_spirograph_curve(
    radius_fixed_circle: float = 10.84,
    radius_moving_circle: float = 3.48,
    pen_offset: float = 6.0,
    nb_turns: float = 23.0,
) -> fiat_implot.FloatMatrix_Dim2:
    """a spirograph-like curve"""
    import numpy as np

    t = np.linspace(0, 2 * np.pi * nb_turns, int(500 * nb_turns))
    x = (radius_fixed_circle + radius_moving_circle) * np.cos(t) - pen_offset * np.c
        (radius_fixed_circle + radius_moving_circle) / radius_moving_circle * t
```

```
    )
    y = (radius_fixed_circle + radius_moving_circle) * np.sin(t) - pen_offset * np.s
        (radius_fixed_circle + radius_moving_circle) / radius_moving_circle * t
    )
    return np.array([x, y])  # type: ignore


@fl.with_fiat_attributes(
    x__range=(0.0, 10.0),
    return__plot_type="bars",
    return__auto_fit=False,
    return__plot_size_em=(20, 10),
)
def get_simple_values(x: float) -> fiat_implot.FloatMatrix_Dim1:
    r = []
    for i in range(10):
        r.append(math.cos(x**i))
    return np.array(r)  # type: ignore


def main() -> None:
    graph = fl.FunctionsGraph()
    graph.add_function(make_spirograph_curve)
    graph.add_function(get_simple_values)

    graph.add_function(time_seconds)
    graph.add_function(phase_from_time_seconds)
    graph.add_function(sin_wave)
    graph.add_link("time_seconds", "phase_from_time_seconds")
    graph.add_link("phase_from_time_seconds", "sin_wave")

    fl.run(graph, app_name="Demo ImPlot")


if __name__ == "__main__":
    main()
```

# Comparisons w. other prototyping tools

Fiatlight integrates features from various tools into a unified, flexible framework for rapid prototyping and exploration.

Similar tools dedicated to rapid prototyping, exploration and visualization include:

- **Scratch**: For visual graph creation.
- **Jupyter**: For interactive data exploration.
- **Python Streamlit** & **Dash**: For easy app creation with integrated GUI elements.

- **Ryven**: For advanced graph creation.
- **Unity Blueprints**: For visual scripting and custom widgets.
- **Comfy UI**: For AI workflow integration.

---

> Compared to the aforementioned software frameworks, Fiatlight distinguishes itself by:

## Pros

- **Automatic GUI Generation** – Introspects functions & structured data to create interfaces.
- **Live Function State Visualization** – View intermediate values at any step.
- **Error Replay & Debugging** – Reproduce issues with the exact inputs that caused them.
- **State Persistence** – Save & restore multiple application states.
- **High-Performance Rendering** – Uses **Dear ImGui** with C++ and OpenGL for speed.
- **Runs Locally in the Browser** – Can be executed entirely **in-browser via Pyodide**, requiring **no server**.
- **Seamless Transition to Full Applications** – Prototypes can evolve into full **Dear ImGui** apps, with easy migration to C++.

## Cons

- **No Server-Side Computation** – Cannot rely on a remote server for **heavy computation** (e.g., large AI models like TensorFlow).

An extensive comparison of Fiatlight with streamlit, with dash, and with ipywidgets is available in the subsections.

## Comparison with Streamlit

This page provides a detailed comparison between Fiatlight and Streamlit, highlighting the strengths and weaknesses of each framework.

## Summary

1. **Example used for the comparison**:

   *Display multiple Matplotlib figures and an animated sine wave*

2. **Performance and Responsiveness**:

   *Streamlit can refresh figures at about 2 FPS, while Fiatlight can update them at 35 FPS (it could be 120 FPS if using ImPlot instead of MatPlotlib). However, no particular effort was made in optimizing streamlit workflow.*

3. **Customization and Extensibility**:

   *Compare how each framework allows customization of widgets and extensibility.*

4. **State Management**:

   *Evaluate how user inputs and application states are managed and restored.*

5. **Algorithmic pipelines**:

   *Examine the support for chaining functions and visualizing their interactions.*

6. **User Experience**:

   *Discuss the overall user experience, including UI manipulation capabilities.*

7. **Ease of Use and Learning Curve**:

   *Assess the ease of learning and using each framework.*

8. **Deployment and Accessibility**:

   *Compare the deployment capabilities and accessibility, including online execution.*

9. **Community and Support**:

   *Look at the available community support and resources.*

10. **Integration with Data Science Tools**:

    *Analyze how well each framework integrates with popular data science libraries and tools.*

# Detailed Comparison

## 1. Example used for the comparison

This comparison is based on the following example, which includes several MatPlotlib figures, along with an animated sine wave.

### Using Fiatlight

See the code of figure_with_gui_demo.py.

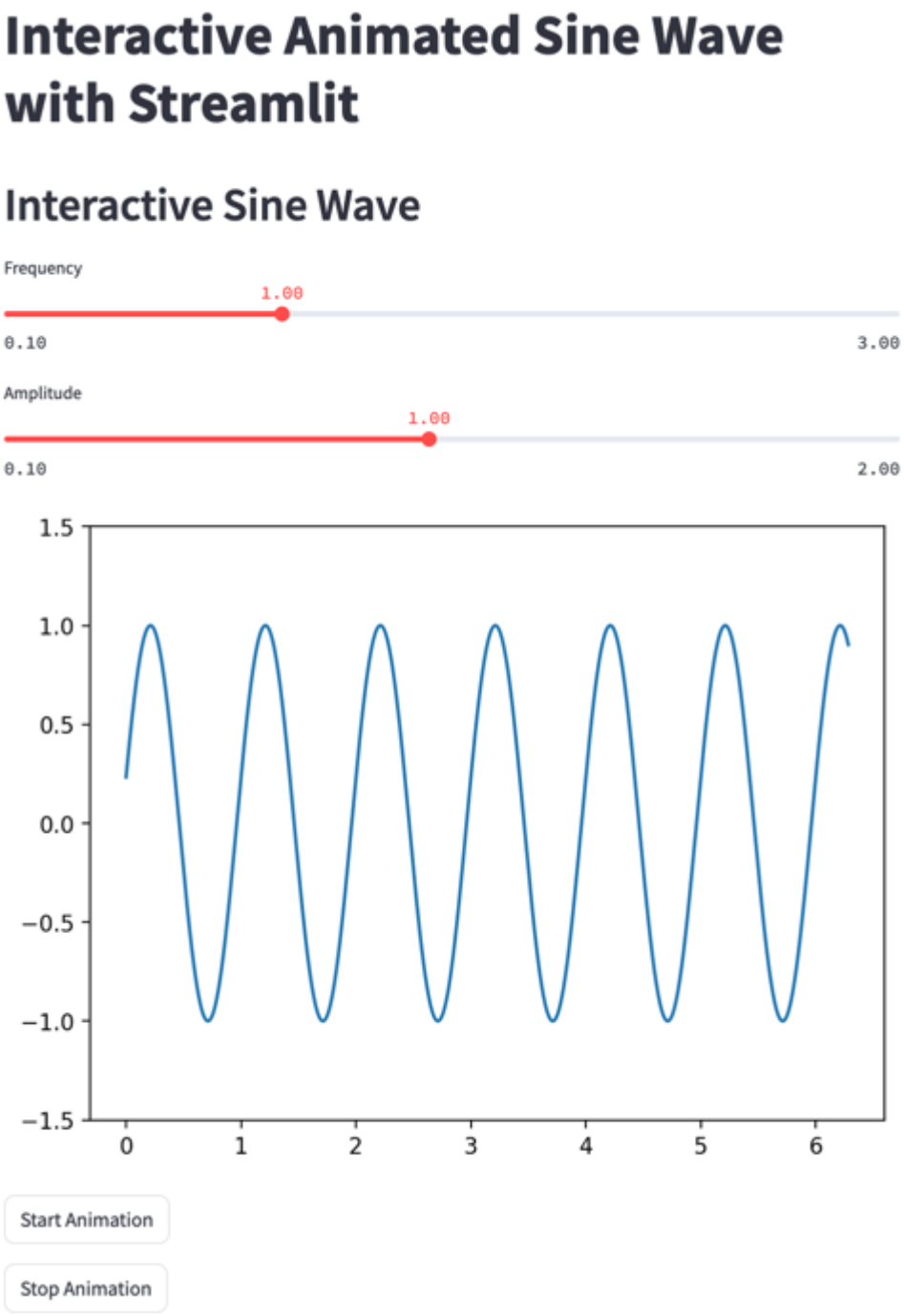
Here it is in action with Fiatlight. The sine wave is animated at 35 FPS.

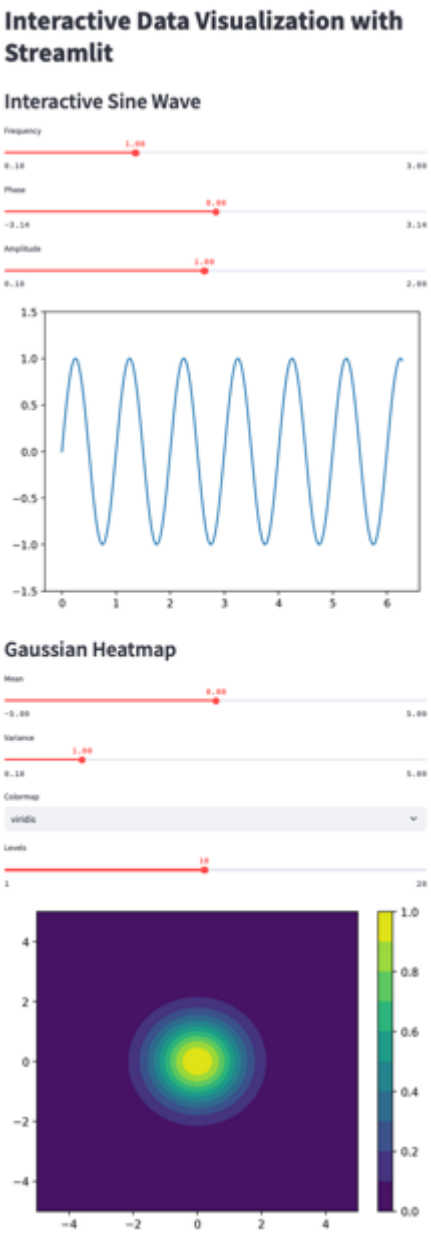
```
from fiatlight.fiat_kits.fiat_matplotlib import demo_matplotlib

figure_with_gui_demo.main()
```

### Using Streamlit

The code for Streamlit was split into two parts: figures and animated sine wave.

Here are screenshots of it in action within Streamlit. The sine wave is animated at 2 FPS.






## 2. Performance and Responsiveness

- **Fiatlight**:
  - Updates at 35 frames per second, providing real-time interactivity. If using ImPlot instead of Matplotlib, Fiatlight would reach the artificial limit of 120 FPS.
- **Streamlit**:
  - Displays at about 2 frames per second. However, no particular effort was made in optimizing streamlit workflow; and better results may be possible (maybe by switching from Matplotlib to Plotly).

## 3. Customization and Extensibility

- **Fiatlight**:
  - Allows deep customization of widgets, including advanced editing types and ranges. Users can define custom widgets and function graphs for extensive flexibility.
- **Streamlit**:
  - Provides a wide range of built-in widgets and customization options, but may not match Fiatlight's specialized functionalities.

## 4. State Management

- **Fiatlight**:
  - Automatically saves and restores user inputs, widget placements, and settings. Supports saving different configurations and restoring them later.
- **Streamlit**:
  - Requires manual handling of state management. Users need to implement custom solutions to save and restore states across sessions.

## 5. Algorithmic pipelines

- **Fiatlight**:
  - Supports function graphs, enabling chaining of functions and visualization of their inputs and outputs, simplifying complex workflows.
- **Streamlit**:
  - Does not natively support function graphs. Users need to manually code function linkages, which can be more cumbersome.

## 6. User Experience

- **Fiatlight**:
  - Offers rich user experience with the ability to resize and move figures, enhancing usability and flexibility.
- **Streamlit**:

- Provides a straightforward interface but lacks advanced UI manipulation features like resizing and moving figures.

## 7. Ease of Use and Learning Curve

- **Fiatlight**:
  - Powerful and flexible but might have a steeper learning curve due to advanced features.
- **Streamlit**:
  - User-friendly and easy to learn, allowing rapid development and prototyping with minimal code.

## 8. Deployment and Accessibility

- **Fiatlight**:
  - Fiatlight can run inside a Jupyter Notebook, but requires a local environment and lacks web-based deployment capabilities. Efforts with pyodide are underway but still in development.
- **Streamlit**:
  - Easily deployable on the web and compatible with platforms like Google Colab, making it accessible from anywhere, which is advantageous for collaboration and sharing.

## 9. Community and Support

- **Fiatlight**:
  - May not have as extensive a community or support resources as Streamlit.
- **Streamlit**:
  - Large and active community, extensive documentation, and support resources, beneficial for new users and those seeking help or examples.

## 10. Integration with Data Science Tools

- **Fiatlight**:
  - Can integrate with data science tools but may require more setup and configuration. Its use of Dear ImGui allows for high-performance graphics and interactive applications,

which can be beneficial for certain data science applications.

- **Streamlit**:
  - Well-integrated with popular data science libraries and tools, making it a go-to choice for data scientists and analysts.

## Conclusion

Both Fiatlight and Streamlit have their unique advantages.

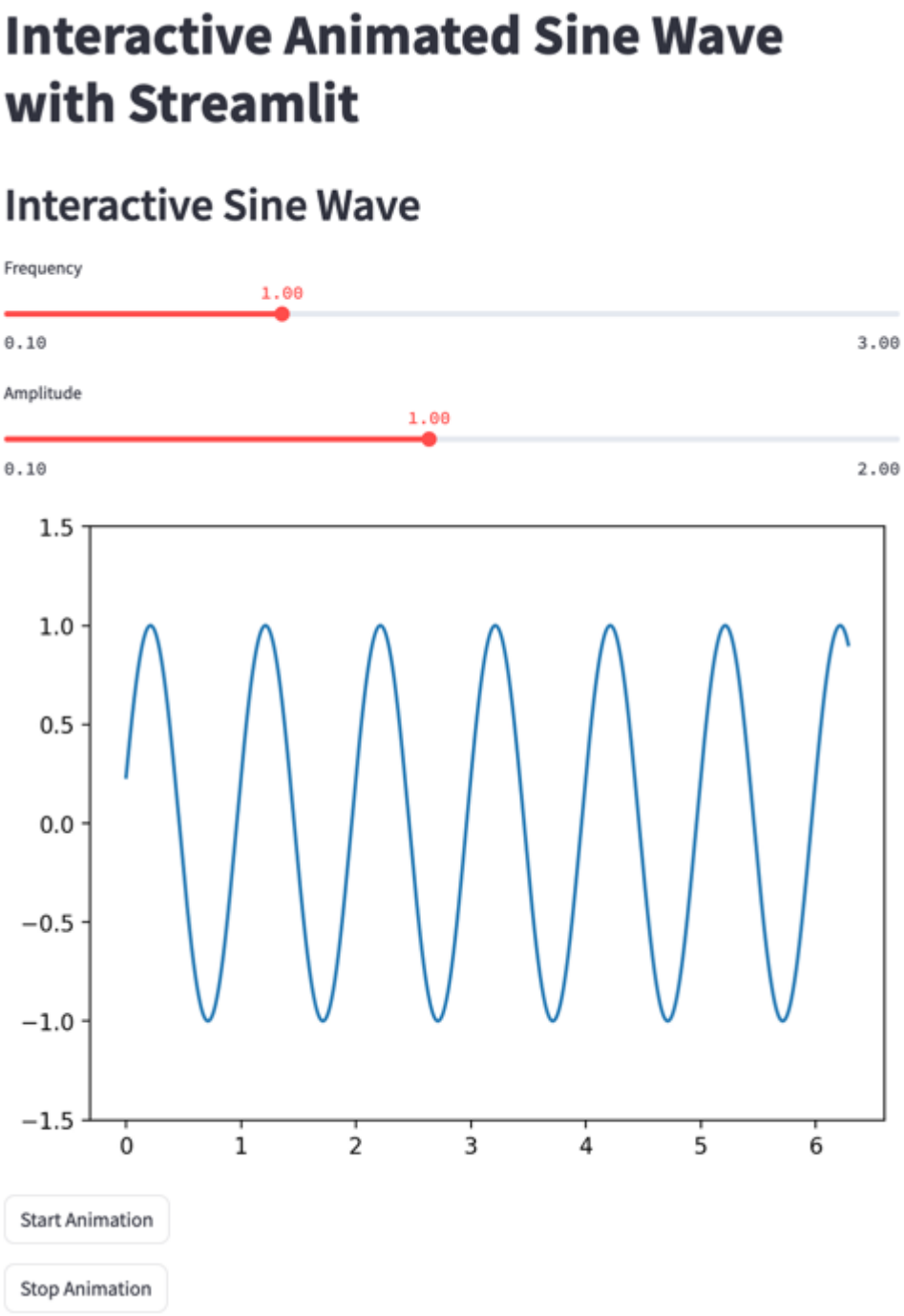
- **Fiatlight** excels in high-performance applications, offering extensive customization, advanced interactive features, and sophisticated state management that includes automatic saving and restoring of user inputs and widget placements. This makes it exceptionally well-suited for rapid prototyping, as users can quickly iterate on their designs without losing their configurations. Its support for function graphs simplifies complex workflows, making it a powerful tool for developing intricate applications.
- **Streamlit** is ideal for users who prioritize ease of use and web-based deployment. It offers a user-friendly interface that facilitates rapid development and prototyping, especially for data-driven applications and dashboards. Its seamless integration with popular data science libraries and web deployment capabilities makes it accessible from anywhere, promoting collaboration and sharing.

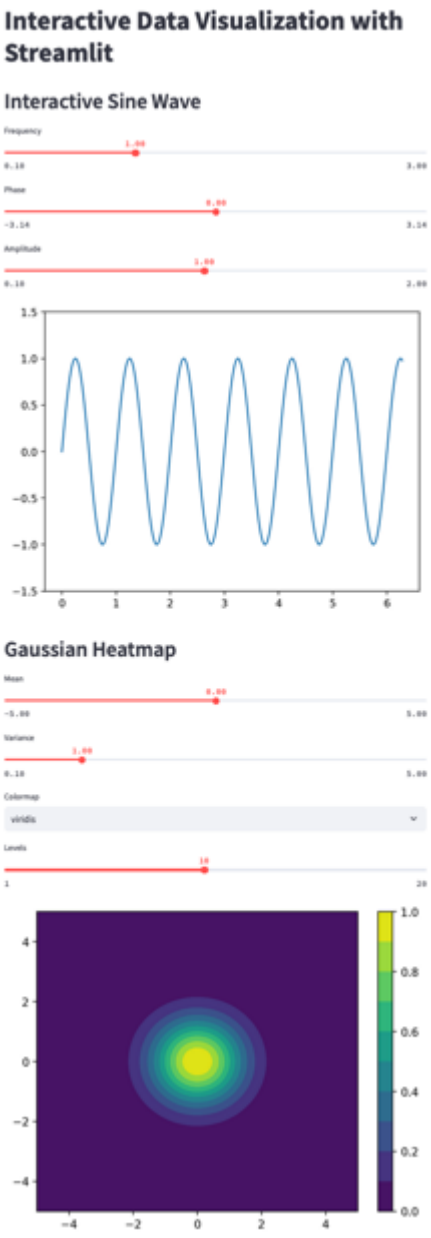
The choice between them depends on the specific needs and preferences of the user or project. Fiatlight offers a more feature-rich environment for those needing advanced GUI capabilities and state management, while Streamlit provides a simpler, more accessible solution for data visualization and web deployment.

## Using Streamlit

The code for Streamlit was split into two parts: figures and animated sine wave.

Here are screenshots of it in action within Streamlit. The sine wave is animated at 2 FPS.






## 2. Performance and Responsiveness

- **Fiatlight**:
  - Updates at 35 frames per second, providing real-time interactivity. If using ImPlot instead of Matplotlib, Fiatlight would reach the artificial limit of 120 FPS.
- **Streamlit**:
  - Displays at about 2 frames per second. However, no particular effort was made in optimizing streamlit workflow; and better results may be possible (maybe by switching from Matplotlib to Plotly).

## 3. Customization and Extensibility

- **Fiatlight**:
  - Allows deep customization of widgets, including advanced editing types and ranges. Users can define custom widgets and function graphs for extensive flexibility.
- **Streamlit**:
  - Provides a wide range of built-in widgets and customization options, but may not match Fiatlight's specialized functionalities.

## 4. State Management

- **Fiatlight**:
  - Automatically saves and restores user inputs, widget placements, and settings. Supports saving different configurations and restoring them later.
- **Streamlit**:
  - Requires manual handling of state management. Users need to implement custom solutions to save and restore states across sessions.

## 5. Algorithmic pipelines

- **Fiatlight**:
  - Supports function graphs, enabling chaining of functions and visualization of their inputs and outputs, simplifying complex workflows.
- **Streamlit**:
  - Does not natively support function graphs. Users need to manually code function linkages, which can be more cumbersome.

## 6. User Experience

- **Fiatlight**:
  - Offers rich user experience with the ability to resize and move figures, enhancing usability and flexibility.
- **Streamlit**:

- Provides a straightforward interface but lacks advanced UI manipulation features like resizing and moving figures.

## 7. Ease of Use and Learning Curve

- **Fiatlight**:
  - Powerful and flexible but might have a steeper learning curve due to advanced features.
- **Streamlit**:
  - User-friendly and easy to learn, allowing rapid development and prototyping with minimal code.

## 8. Deployment and Accessibility

- **Fiatlight**:
  - Fiatlight can run inside a Jupyter Notebook, but requires a local environment and lacks web-based deployment capabilities. Efforts with pyodide are underway but still in development.
- **Streamlit**:
  - Easily deployable on the web and compatible with platforms like Google Colab, making it accessible from anywhere, which is advantageous for collaboration and sharing.

## 9. Community and Support

- **Fiatlight**:
  - May not have as extensive a community or support resources as Streamlit.
- **Streamlit**:
  - Large and active community, extensive documentation, and support resources, beneficial for new users and those seeking help or examples.

## 10. Integration with Data Science Tools

- **Fiatlight**:
  - Can integrate with data science tools but may require more setup and configuration. Its use of Dear ImGui allows for high-performance graphics and interactive applications,

which can be beneficial for certain data science applications.

- **Streamlit**:
  - Well-integrated with popular data science libraries and tools, making it a go-to choice for data scientists and analysts.

### Conclusion

Both Fiatlight and Streamlit have their unique advantages.

- **Fiatlight** excels in high-performance applications, offering extensive customization, advanced interactive features, and sophisticated state management that includes automatic saving and restoring of user inputs and widget placements. This makes it exceptionally well-suited for rapid prototyping, as users can quickly iterate on their designs without losing their configurations. Its support for function graphs simplifies complex workflows, making it a powerful tool for developing intricate applications.
- **Streamlit** is ideal for users who prioritize ease of use and web-based deployment. It offers a user-friendly interface that facilitates rapid development and prototyping, especially for data-driven applications and dashboards. Its seamless integration with popular data science libraries and web deployment capabilities makes it accessible from anywhere, promoting collaboration and sharing.

The choice between them depends on the specific needs and preferences of the user or project. Fiatlight offers a more feature-rich environment for those needing advanced GUI capabilities and state management, while Streamlit provides a simpler, more accessible solution for data visualization and web deployment.

# Comparison with Dash

This page provides a detailed comparison between Fiatlight and Dash, highlighting the strengths and weaknesses of each framework.

## Summary

1. **Example used for the comparison**: *Display multiple Matplotlib figures and an animated sine wave*

2. **Performance and Responsiveness**: *Dash can update the graph up to 45 FPS on a local server (using Plotly), but is likely to be slower on a remote server. Fiatlight can update the graph up to 35 FPS when using Matplotlib. If using ImPlot instead of Matplotlib, Fiatlight would reach the artificial limit of 120 FPS.*
3. **Customization, Layout and Extensibility**: *Compare how each framework allows customization of widgets and extensibility.*
4. **State Management**: *Evaluate how user inputs and application states are managed and restored.*
5. **Algorithmic Pipelines**: *Examine the support for chaining functions and visualizing their interactions.*
6. **User Experience**: *Discuss the overall user experience, including UI manipulation capabilities.*
7. **Ease of Use and Learning Curve**: *Assess the ease of learning and using each framework.*
8. **Deployment and Accessibility**: *Compare the deployment capabilities and accessibility, including online execution.*
9. **Community and Support**: *Look at the available community support and resources.*
10. **Integration with Data Science Tools**: *Analyze how well each framework integrates with popular data science libraries and tools.*

## Detailed Comparison

### 1. Example used for the comparison

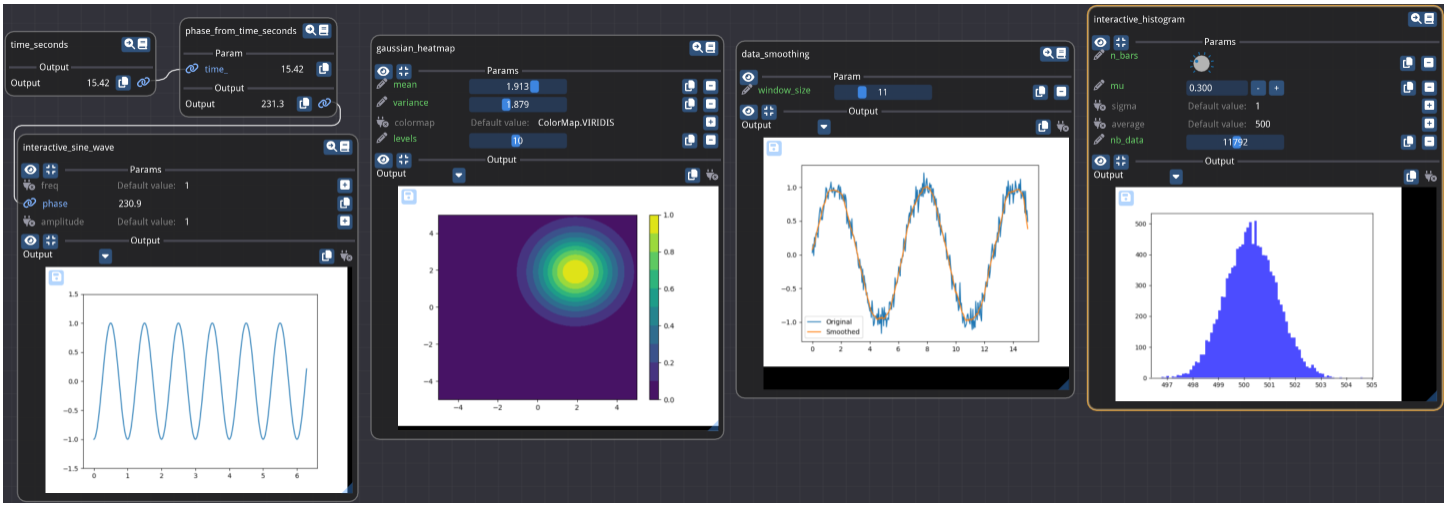
This comparison is based on the following example, which includes several Matplotlib figures, along with an animated sine wave.

#### Using Fiatlight

See the code of figure_with_gui_demo.py.

Here it is in action with Fiatlight. The sine wave is animated at 35 FPS (it could be 120 FPS if using ImPlot instead of MatPlotlib).

```
from fiatlight.fiat_kits.fiat_matplotlib import demo_matplotlib

figure_with_gui_demo.main()
```




## Using Dash

A similar application was coded for Dash. Here is its source code.

And below is a screenshot of the Dash app with multiple figures and an animated sine wave.


## 2. Performance and Responsiveness

- **Fiatlight**:
  - When using Matplotlib, Fiatlight runs at 35 FPS. When using ImPlot, it runs at the artificial limit of 120 FPS.
- **Dash**:
  - Dash can update the graph up to 45 FPS on a local server (using Plotly), but is likely to be slower on a remote server, because each timer update requires communication via a web socket.

## 3. Customization, Layout and Extensibility

- **Fiatlight**:
  - Allows deep customization of widgets, including advanced editing types and ranges. Users can define custom widgets and function graphs for extensive flexibility.
  - Supports advanced layout management, including resizing and moving figures. Arranging the functions on the screen is as easy as dragging with the mouse. And since those options are saved, they become part of the final application.
  - The code for the application occupies 135 Python lines.
- **Dash**:
  - Offers a wide range of customizable components including Knobs, but may require more manual coding to achieve highly customized interfaces.
  - The layout is achieved via standard HTML divs. Changing their size or moving them requires some adaptation on the Python side.
  - The code for the application occupies 242 Python lines.

## 4. State Management

- **Fiatlight**:
  - Automatically saves and restores user inputs, widget placements, and settings. Supports saving different configurations and restoring them later.
- **Dash**:
  - State management is manual and typically involves more code to save and restore states across sessions.

## 5. Algorithmic Pipelines

- **Fiatlight**:
  - Supports function graphs, enabling chaining of functions and visualization of their inputs and outputs, simplifying complex workflows.
- **Dash**:
  - Supports callbacks to chain functions but may be less visual and more code-intensive.

## 6. User Experience

- **Fiatlight**:
  - Offers a rich user experience with the ability to resize and move figures, enhancing usability and flexibility.
- **Dash**:
  - Provides a straightforward interface with interactive components but lacks advanced UI manipulation features like resizing and moving figures.

## 7. Ease of Use and Learning Curve

- **Fiatlight**:
  - Powerful and flexible, but it might require some initial learning since it is a novel framework. However, the immediate GUI mode is easy to grasp, making it accessible for new users.
- **Dash**:
  - User-friendly but can become complex with advanced use cases, requiring a good understanding of the Dash framework and callbacks.

## 8. Deployment and Accessibility

- **Fiatlight**:
  - Fiatlight can run inside a Jupyter Notebook, but requires a local environment and lacks web-based deployment capabilities. Efforts with pyodide are underway but still in development.
- **Dash**:
  - Easily deployable on the web, with built-in support for deploying to cloud platforms like Heroku and Azure.

## 9. Community and Support

- **Fiatlight**:
  - May not have as extensive a community or support resources as Dash.

- **Dash**:
  - Large and active community, extensive documentation, and support resources, beneficial for new users and those seeking help or examples.

## 10. Integration with Data Science Tools

- **Fiatlight**:
  - Can integrate with data science tools but may require more setup and configuration. Its use of Dear ImGui allows for high-performance graphics and interactive applications, which can be beneficial for certain data science applications.
- **Dash**:
  - Well-integrated with popular data science libraries and tools, making it a go-to choice for data scientists and analysts.

## Conclusion

Both Fiatlight and Dash have their unique advantages.

- **Fiatlight** excels in high-performance applications, offering extensive customization, advanced interactive features, and sophisticated state management that includes automatic saving and restoring of user inputs and widget placements. This makes it exceptionally well-suited for rapid prototyping, as users can quickly iterate on their designs without losing their configurations. Its support for function graphs simplifies complex workflows, making it a powerful tool for developing creative applications.
- **Dash** is ideal for users who prioritize building interactive dashboards and data visualization applications with ease of deployment. It offers a user-friendly interface that facilitates rapid development and deployment, especially for data-driven applications. Its seamless integration with popular data science libraries and robust web deployment capabilities make it accessible and powerful for building analytical web applications.

The choice between them depends on the specific needs and preferences of the user or project. Fiatlight offers a more feature-rich environment for those needing advanced GUI capabilities and state management, while Dash provides a robust solution for building and deploying data-driven dashboards and applications.

# Comparison with Jupyter Lab and ipywidgets

This page provides a detailed comparison between Fiatlight and Jupyter Lab + ipywidgets, highlighting the strengths and weaknesses of each framework.

## Summary

1. **Example used for the comparison**: *Display multiple Matplotlib figures and an animated sine wave*
2. **Performance and Responsiveness**: *Compare the performances of both frameworks on live and static figures.*
3. **Customization, Layout and Extensibility**: *Compare how each framework allows customization of widgets and extensibility.*
4. **State Management**: *Evaluate how user inputs and application states are managed and restored.*
5. **Algorithmic Pipelines**: *Examine the support for chaining functions and visualizing their interactions.*
6. **User Experience**: *Discuss the overall user experience, including UI manipulation capabilities.*
7. **Ease of Use and Learning Curve**: *Assess the ease of learning and using each framework.*
8. **Deployment and Accessibility**: *Compare the deployment capabilities and accessibility, including online execution.*
9. **Community and Support**: *Look at the available community support and resources.*
10. **Integration with Data Science Tools**: *Analyze how well each framework integrates with popular data science libraries and tools.*

## Detailed Comparison

### 1. Example used for the comparison

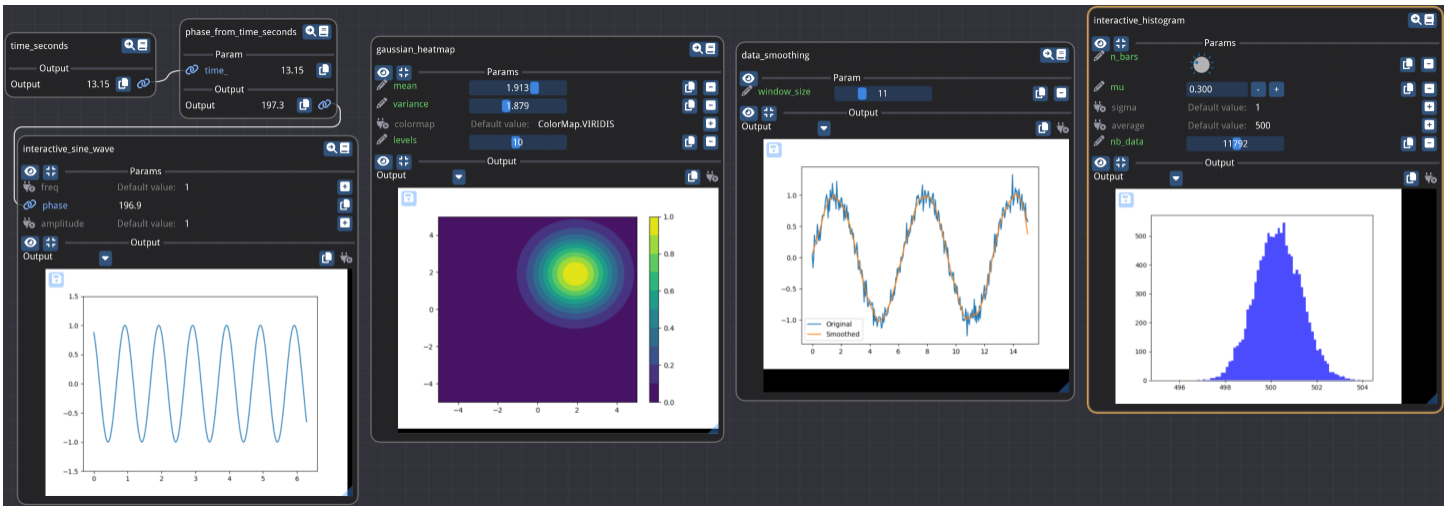
This comparison is based on the following example, which includes several Matplotlib figures, along with an animated sine wave.

### Using Fiatlight

See the code of figure_with_gui_demo.py.

Here it is in action with Fiatlight. The sine wave is animated at 35 FPS (it could be 120 FPS if using ImPlot instead of MatPlotlib).

```
from fiatlight.fiat_kits.fiat_matplotlib import demo_matplotlib

figure_with_gui_demo.main()
```




### Using ipywidgets / Jupyter Lab

A similar demo was created using Jupyter Lab and ipywidgets. It is available in this notebook.

## 2. Performance and Responsiveness

It is surprisingly difficult to create live figures in Jupyter Lab. Also, while a figure is being updated, widgets will not transmit new values to python.

An animated figure can be created by updating a figure in a loop inside a cell. The refresh rate using Matplotlib is about 1 FPS, and much higher when using `ProgressPlot`. However, the user has to wait until the cell has finished executing.

Fiatlight can update the graph up to 35 FPS when using Matplotlib. If using ImPlot instead of Matplotlib, Fiatlight would reach the artificial limit of 120 FPS. The updates are done asynchronously and all the other widgets remain active.

## 3. Customization, Layout and Extensibility

- **Fiatlight**:
  - Allows deep customization of widgets, including advanced editing types and ranges. Users can define custom widgets and function graphs for extensive flexibility.
  - Supports advanced layout management, including resizing and moving figures. Arranging the functions on the screen is as easy as dragging with the mouse. Since these options are saved, they become part of the final application.
  - The code for the application occupies 135 Python lines.
- **Jupyter / ipywidgets**:
  - Offers a variety of customizable components, including sliders, checkboxes, dropdowns, and text inputs. Users can create interactive widgets that integrate seamlessly with Jupyter notebooks.
  - The layout is limited to what is possible inside a notebook, but you can use `ipywidgets`'s `HBox`, `VBox`, and other layout widgets to organize components. However, it lacks the advanced layout management features like resizing and moving figures within the notebook interface.
  - The code for the application occupies 142 Python lines.

## 4. State Management

- **Fiatlight**:
  - Automatically saves and restores user inputs, widget placements, and settings. Supports saving different configurations and restoring them later.
- **Jupyter / ipywidgets**:
  - State management is manual and typically involves more code to save and restore states across sessions.

## 5. Algorithmic Pipelines

- **Fiatlight**:
  - Supports function graphs, enabling chaining of functions and visualization of their inputs and outputs, simplifying complex workflows.
- **Jupyter / ipywidgets**:
  - Supports sequential and interactive cell execution but lacks a native function graph feature. While users can manually code function linkages and interactions, it does not offer the same visual pipeline management as Fiatlight.

## 6. User Experience

- **Fiatlight**:
  - Offers a rich user experience with the ability to resize and move figures, enhancing usability and flexibility.
- **Jupyter / ipywidgets**:
  - offers a basic user experience for the final user. Note: the appearance of the ipywidgets is not restored when reopening a notebook: the user has to re-run the cells to get the widgets back.

## 7. Ease of Use and Learning Curve

- **Fiatlight**:
  - Powerful and flexible, but it might require some initial learning since it is a novel framework. However, the immediate GUI mode is easy to grasp, making it accessible for new users.
- **Jupyter / ipywidgets**:
  - offers a truly great experience for the developer, in terms of ease and speed of development.

## 8. Deployment and Accessibility

- **Fiatlight**:
  - Fiatlight can run inside a Jupyter Notebook, but requires a local environment and lacks web-based deployment capabilities. Efforts with pyodide are underway but still in development.

- **Jupyter / ipywidgets**:
  - deployable locally and on almost any cloud provider (Google Colab, Binder, etc.)

## 9. Community and Support

- **Fiatlight**:
  - May not have as extensive a community or support resources as more established frameworks, but it benefits from the communities of the libraries it builds upon, like Dear ImGui, Hello ImGui, and ImGui Bundle.
- **Jupyter / ipywidgets**:
  - Large and active community, extensive documentation, and support resources, beneficial for new users and those seeking help or examples. Many resources are available for troubleshooting and expanding functionality.

## 10. Integration with Data Science Tools

- **Fiatlight**:
  - Can integrate with data science tools but may require more setup and configuration. Its use of Dear ImGui allows for high-performance graphics and interactive applications, which can be beneficial for certain data science applications.
- **Jupyter / ipywidgets**:
  - Very mature integration with popular data science libraries and tools such as NumPy, pandas, scikit-learn, and more. It is widely used in the data science community, making it a go-to choice for data-driven research and analysis.

## Conclusion

Both Fiatlight and Jupyter Lab with ipywidgets have their unique advantages.

- **Fiatlight** excels in high-performance applications, offering extensive customization, advanced interactive features, and sophisticated state management that includes automatic saving and restoring of user inputs and widget placements. This makes it exceptionally well-suited for rapid prototyping, as users can quickly iterate on their designs without losing their configurations. Its support for function graphs simplifies complex workflows, making it a powerful tool for developing creative applications.

- **Jupyter Lab with ipywidgets** is ideal for users who prioritize ease of use, rapid development, and integration with data science tools. It offers a user-friendly interface that facilitates interactive data analysis and visualization. The extensive community support, along with its deployment capabilities on platforms like Google Colab and Binder, make it highly accessible and powerful for educational and research purposes.

The choice between them depends on the specific needs and preferences of the user or project. Fiatlight offers a more feature-rich environment for those needing advanced GUI capabilities and state management, while Jupyter Lab with ipywidgets provides a robust solution for interactive data science and educational applications.