この記事は最終更新日から1年以上が経過しています。

@sunameri22

Python画像処理のためのGUI入門（PySimpleGUI解説）

Python GUI 画像処理 入門 PySimpleGUI

最終更新日 2022年07月07日 投稿日 2021年06月20日

はじめに

Pythonを使って画像処理を行う際、パラメータを動的に変更しながら処理結果を確認したいということは多いかと思います（ですよね？）。こういった時にGUIベースのアプリケーションが作れると非常に便利ですが、PySimpleGUI でそれを簡単に実現することができます。

本記事では PySimpleGUI の基本的動作を理解しながら、インタラクティブな画像処理アプリケーションの作成方法を習得することを目的とします。

PySimpleGUI に関する基本的な説明から入るので、ある程度使い方が分かっている方は実践編から入って、不明点は都度戻って補完するのが良いかと思います。

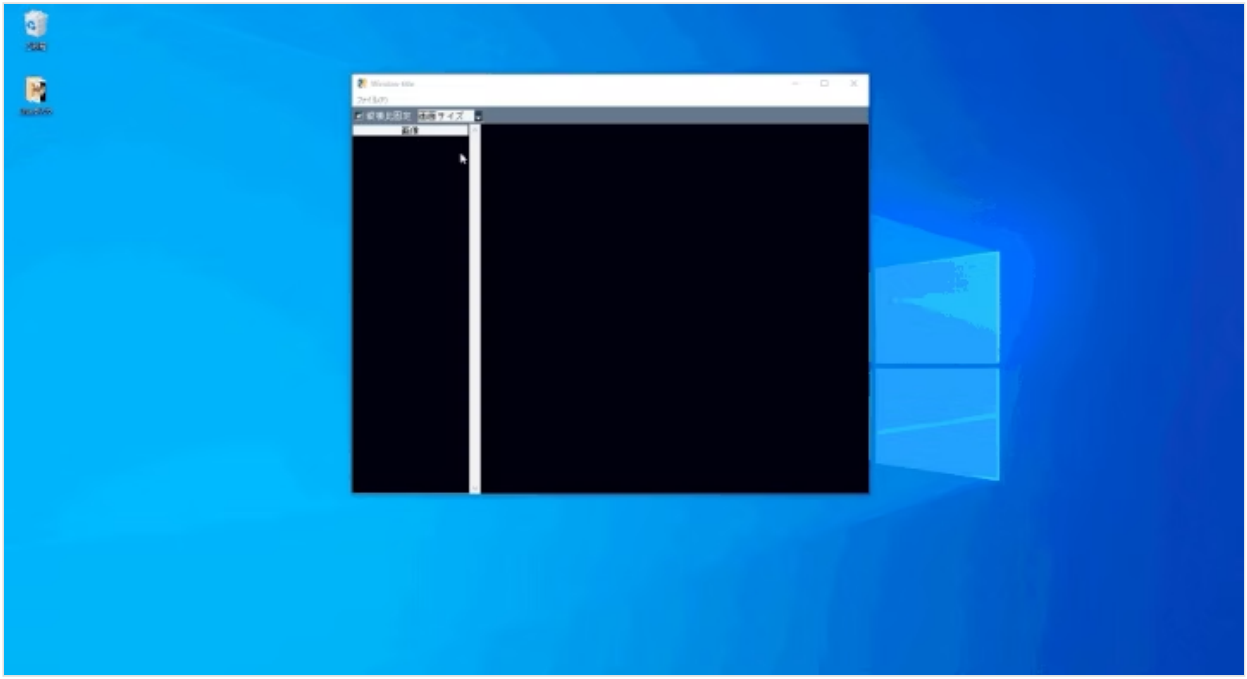
よく見そうなところ

項目	概要
基本レイアウト	コピペすればとりあえず動く雛形

項目	概要
レイアウトの引数	レイアウト調整に使う引数について
メソッド一覧	要素に適用できるメソッドを見る
bind_string 一覧	特殊な操作を受け付けたいときに
Graph 用メソッド一覧	Graphで描画処理をする
実践編	画像ビューワー作成の実践

完成物のイメージ

例として、選んだフォルダにある画像を一覧で見られる画像ビューワーを作成します。多機能ではありませんが、そのぶん好きなようにカスタマイズして使えるかと思います。



導入

pip によるインストールが可能です。ターミナルから下記コマンドを入力します。

```
pip install pysimplegui
```

インストールが完了したら、 `import PySimpleGUI` で使用できるようになります。長いのでだいたい `sg` と略されることが多いです。

```
import PySimpleGUI as sg
```

基本的な処理の流れ

最低限の構成は以下のようになります。

```
import PySimpleGUI as sg

# 1. レイアウト
layout = [
    [
        sg.Button('押してね', size=(30, 3), key='BUTTON'),
    ],
]

# 2. ウィンドウの生成
window = sg.Window(
    title='Window title',
    layout=layout
)
window.finalize()

# 3. GUI処理
while True:
    event, values = window.read(timeout=None)
    if event is None:
        break
window.close()
```

- レイアウト
- ウィンドウの生成
- GUI処理

の 3 つに分かれます。それぞれに分けて解説します。

レイアウト

```
layout = [
    [
        sg.Button('押してね', size=(60, 5), key='BUTTON'),
    ],
]
```

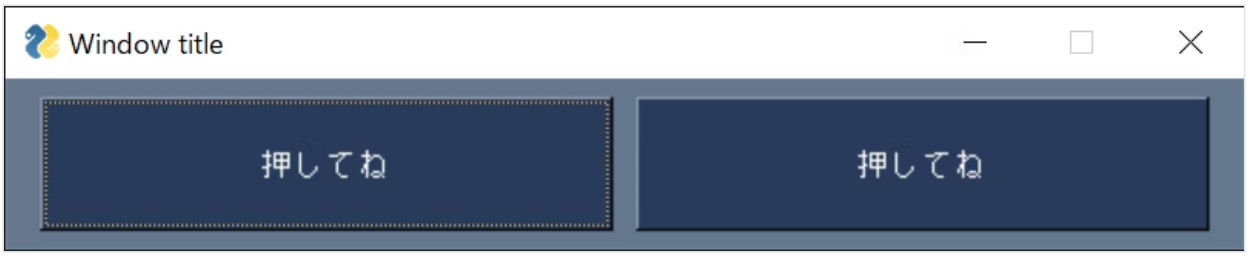
基本的にレイアウトは2次元のリストで表します。

0次元目は行を表すので要素が縦に並び、

```
layout = [
    [
        sg.Button('押してね', size=(30, 3), key='BUTTON_1'),
    ],
    [
        sg.Button('押してね', size=(30, 3), key='BUTTON_2'),
    ],
]
```

1次元目は列となるので要素は横に並びます。

```
layout = [
    [
        sg.Button('押してね', size=(30, 3), key='BUTTON_1'),
        sg.Button('押してね', size=(30, 3), key='BUTTON_2'),
    ],
]
```



ウィンドウの生成

```
window = sg.Window(
    title='Window title',
    layout=layout
)
```

変数 `window` にウィンドウを格納し、 `window.finalize()` でウィンドウを確定、表示させます。 `window.finalize()` は必須ではありませんが、個人的には入れることを推奨します。

GUI処理

```
while True:
    event, values = window.read(timeout=None)
    if event is None:
        break
window.close()
```

ここがメインの処理になります。 `window.read()` を呼び出すとウィンドウの内容を更新し、ユーザーのアクション（ボタンを押す、スライダーを動かす、など）を待ちます。ユーザーがアクションを起こしたら、起こしたアクションの内容と、ウィンドウ内の各要素が持つ値をそれぞれ `event` , `values` に格納します。

試しにこれらの値を `print` でチェックしてみましょう。該当する部分を以下のように変更し、実行してみます。

```
while True:
    event, values = window.read(timeout=None)
    # 変更部分
    print('Event: ', event)
    print('Values: ', values)
    # 変更部分終わり
    if event is None:
        break
window.close()
```

すると、起こしたアクションによって以下のような出力が得られます。

```
# 「押してね」ボタン押下
Event:  BUTTON
Values:  {}
```

```
Event:  None
Values:  None
```

「押してね」ボタンを押すと `event` に `'BUTTON'` が格納されていることが分かりますが、これはボタンをレイアウトする際に `key='BUTTON'` を指定しているためです。つまり、 `event` にはユーザーがアクションを起こした対象の `key` が格納されるということになります。レイアウト時の `key=` は省略可能ですが、ボタンやスライダーなど、ユーザーが操作することを前提とした要素について省略することは推奨しません。

▶ もう少し詳しく

×ボタンには `key` が存在せず、 `None` が格納されます。これを利用して×ボタンの押下をキャッチし、 `While` 文の外に出ます。

一方で `values` には辞書が格納されていますが、空の辞書しか返してくれません。これは `Button` 要素が保持している値がないためです。試しにテキストボックスである `Input` 要素をレイアウトに追加してみましょう。

変数 `layout` を以下のように変更します。

```
layout = [
    [
        sg.Button('押してね', key='BUTTON'),
    ],
    [
        sg.Input(key='INPUT1'),
    ],
    [
        sg.Input(key='INPUT2'),
    ],
    [
        sg.Input(key='INPUT3'),
    ],
]
```

今度はボタンを押すと `values` に値が格納されました。テキストボックスに文字を入

ます。

```
# 「押してね」ボタン押下
Event:  BUTTON
Values:  {'INPUT1': '', 'INPUT2': '', 'INPUT3': ''}

# 文字を入力してもう一度ボタン押下
Event:  BUTTON
Values:  {'INPUT1': '1 行目だよ', 'INPUT2': '2 行目だよ', 'INPUT3': '3 行目だよ'}

# ×ボタンで閉じる
Event:  None
Values:  None
```

このように値を保持する要素がある場合は、 `values` に各要素の `key` とその値の組み合わせが辞書として格納されます。特定の要素の値を取り出したい場合は `values['INPUT1']` のように書きます。

ここで要注意なのが、×ボタンを押した場合は `values` に `None` が格納されるという点です。以下のコードを例にとります。

```
while True:
    event, values = window.read(timeout=None)
    # 変更部分
    print('INPUT1: ', values['INPUT1'])
    # 変更部分終わり
    if event is None:
        break
window.close()
```

この場合は、×ボタンでウィンドウを閉じた際に

```
TypeError: 'NoneType' object is not subscriptable
```

と怒られます。×ボタンを押すと `values` に `None` が格納されるので、`None['INPUT1']` なんて無理ですよということですね。

要素

前章では `Button` 要素と `Input` のみ扱いましたが、PySimpleGUIには他にもさまざまな特徴を持った要素があります。ここでは個人的に利用頻度の高い要素を解説します。1から10まで通しで覚えるよりは、とりあえずざっと見て、ほしい所だけ都度見直す感じがいいかもしれません。というわけでザクッと表にまとめました。

補足: イベントの有効化

ここに挙げた要素のうち `Button`, `Menu` を除くすべての要素は、初期状態でイベントの発生を無効化されています。これを有効にしたい場合は、レイアウトする際に引数 `enable_events=True` を指定します。

要素一覧

要素名	概要	event	values
Button	押しボタン	ボタンクリック	-
Text	ラベル用の固定テキスト	テキストクリック	-
Input	入力可能なテキストボックス	キー入力	テキスト内容
Multiline	入力可能なテキストボックス（改行あり）	キー入力	テキスト内容
Slider	つまみで値を変更できるスライダー	つまみ位置変更	現在値
Spin	▲▼ボタンによって値の増減が可能なテキストボックス	▲▼ボタン or ボックス内クリック	現在値
Combo	選択肢から内容を選べるテキストボックス	選択肢の選択	現在値
Check	クリックで ON / OFF を切り替え	クリック	チェック有

要素名	概要	event	values
Radio	クリックで複数の選択肢から１つを選ぶラジオボタン	クリック	チェック有無
Menu	ウィンドウ上部のメニューバー	項目選択	-
Table	行の選択が可能な表（列は選べない）	要素選択	選択要素のリスト
Graph	図形の描画や画像の貼り付けができるエリア	グラフ範囲内クリック	最終クリック座標
Column	複数の要素を１つの要素にまとめる入れ物	-	-

Button

```
sg.Button('Button text', key='BUTTON')
```

押すとイベントを発生させるボタンです。

一覧に戻る

Text

```
sg.Text('Text')
```

文字列を表示します。基本的にはラベル用の固定テキストとして使うことが大半なので、 `key` を設定する必要はありません。 `Text` から `values` を取得しようとして `KeyError` で怒られるのは誰もが通る道。

一覧に戻る

Input

```
sg.Input('Default_text', disabled=False, key='INPUT')
```

入力可能なテキストボックスです。 `disabled=True` とすることでユーザーの手入力による変更を禁止することができ、表示のみ行いたい場合に有効です。

一覧に戻る

Multiline

```
sg.Multiline('Multi \ntext', disabled=False, key='MULTILINE')
```

Inputを複数行対応にしたものです。 `print` メソッドにより通常の `print` 関数によるコンソール出力のような使い方ができます（後述）。こちらも `Input` 同様に `disabled=True` とすることで手入力を禁止できます。

一覧に戻る

Slider

```
# 範囲(0, 10), 初期値0, 1刻み、縦方向のスライダー
sg.Slider((0, 10), 0, 1, orientation='v', disable_number_display=False, key='SLIDER')
```

つまみを動かすことで値を変更可能なスライダーです。下（左）端の数値、上（右）端の数値をタプルで指定します。加えて初期値と取りうる数値の刻みを指定します。0.5 刻みや 0.1 刻みなど整数以外も指定可能です。スライダーの方向を `orientation=` で指定でき、 `'v'` が上下方向、 `h` が左右方向です。デフォルトはつまみの隣に現在値が表示されますが、 `disable_number_display=True` を指定すると非表示になります。

Spin

```
# 松竹梅から選択する。初期値'梅'
sg.Spin(['松', '竹', '梅'], '梅', readonly=False, key='SPIN_1')
# range()の返り値をそのまま渡すとバグるのでリストに変換しておく
sg.Spin(list(range(100)), 0, readonly=False, key='SPIN_2')
```

テキストボックスですが、▲▼ボタンを押すことで、値の変更が可能です。値の候補はリストかタプルで指定し、中身は文字列、数値以外でも問題ありません。 `range()` 関数を使う場合は必ずリストかタプルに変換しておきましょう。 `readonly=True` とすることで▲▼ボタン以外での値の変更を禁止することができます。

一覧に戻る

Combo

```
# 松竹梅から選択する。初期値'梅'
sg.Combo(['松', '竹', '梅'], '梅', readonly=False, key='COMBO')
```

テキストボックスですが、▼ボタンを押すことで事前に設定した選択肢から値を選ぶことができます。選択肢はリストかタプルで指定します。 `readonly=True` とすることで▼ボタン以外での値の変更を禁止することができます。

一覧に戻る

Checkbox

```
sg.Checkbox('Checkbox', False, key='CHECKBOX')
```

クリックでチェックを ON / OFF できるチェックボックスです。初期状態のチェック有無を `True or False` で指定します。

Radio

```
sg.Radio('松', 'group_1', True, key='RADIO_MATSU'),
sg.Radio('竹', 'group_1', False, key='RADIO_TAKE'),
sg.Radio('梅', 'group_1', False, key='RADIO_UME'),
```

複数のボタンから ON にするものを選択するラジオボタンです。 `Checkbox` と似ていますが、大きく異なるのは複数のボタンがグループ化されることです。上の例では 3 つのボタンを `group_1` に登録しており、どれか 1 つが ON になれば残り 2 つは OFF になります。 `Checkbox` と同様に初期状態のチェック有無を `True or False` で指定します。ただしこのとき同じグループの 2 つ以上のボタンに `True` を指定しないよう注意してください。

一覧に戻る

Menu

```
sg.Menu(
    [
        [
            'ファイル(&F)',
            [
                '新規作成 (&N)::MENU_NEW::',
                '開く (&O)::MENU_OPEN::',
                '保存 (&S)::MENU_SAVE::',
                '名前を付けて保存 (&A)::MENU_SAVEAS::',
                '終了 (&X)::MENU_EXIT::',
            ],
        ],
        [
            '編集(&E)',
            [
                '元に戻す (&Z)::MENU_UNDO::',
                'やり直し (&Y)::MENU_REDO::',
                '変形 (&F)::',
                [
                    '反転 (&F)::MENU_FLIP::'
```

```
                ],
            ],
        ],
        [
            'ヘルプ(&H)',
            [
                'ユーザーマニュアル (&M)::MENU_MANUAL::',
                'バージョン情報 (&A)::MENU_VERSION::',
            ],
        ],
    ],
),
```

よくあるウィンドウ上部のメニューバーです。これがあるだけでグッとそれらしくなりますし、画面がすっきりするのでおすすめです。

レイアウトが多少ややこしいですが、

```
['メニュータイトル', ['項目1', '項目2', ...]]
```

をリストでまとめたものと理解すれば、そこまで難しくはありません。入れ子構造にすることで、メニューの項目からさらに選択肢を展開することも可能です（変形(F)の欄を参照）。

追加機能として `'&'+半角英数` によりキーボードでのアクセスが有効になります。これを項目名に組み込むことで、Alt + 該当キーでその項目が選択されるようになります。

この要素でイベントを発生させたときの特徴として、 `event` に `key=` の値ではなく**項目の名前**が格納されます。したがって `key=` の指定は必要ないのですが、後から項目名を変更した際にバグが発生してしまいます。

回避策として、項目名の末尾に `::[key代わりの文字列]` を挿入する方法があります。上の例の `Menu` 要素を表示してみると、 `::` より後ろの文字はメニューに表示されていないかと思います。一方で項目を選択した際、 `event` には `::` 以降も含めた文字列が格納されています。つまり、

```
if '::MENU_NEW::' in event:
```

のような式があれば、表示されている内容によらず末尾に `::MENU_NEW::` をもつ項目に対応することができます[1]。末尾にも `::` をつけているのは、 `key` の終わりを判定するためで、これを書かずに

```
if '::MENU_SAVE' in event:
```

と書けば `'名前を付けて保存(&A)::MENU_SAVEAS'` にも反応してしまいます（**::MENU_SAVE**ASなので）。

一覧に戻る

Table

```
sg.Table(
    [[]], # 表の中身
    ['Col 1', 'Col 2', 'Col 3'], # ヘッダー名
    col_widths=[5, 5, 5], # 列幅（列ごとに個別で指定）
    auto_size_columns=False, # col_widthを指定するなら必ずFalseにすること！
    select_mode=None, # 行の選択方法
    num_rows=None, # 表示する行数（はみ出た分はスクロールバーで表示できる）
    key='TABLE'
)
```

リストを表として表示することができます。後述の `update` メソッドを使ってリストの中身を確認するビューワーとして利用されることが多いです。他の要素と比べると複雑で覚えることが多いですが、非常に強力な要素で多くの場面で活用することができます。ここでつまづきがちなポイントを 2 つ紹介します。

中身は必ず 2 次元リストで指定する

一列だけの表であっても必ず 2 次元リストを指定する必要があります。1 次元リスト

```
    displaycolumns = element.ColumnHeadings if element.ColumnHeadings is not None else (
IndexError: list index out of range
```

```
    res = tk.call(*(args + options))
_tkinter.TclError: Invalid column index T
```

などと怒られることになります。

表の中身を空リスト `[[]]` にしたら必ず `auto_size_columns=False` とする

これをしないと

```
    width = max(column_widths[i], len(heading))  * _char_width_in_pixels(font)
KeyError: 0
```

といって怒られます。そもそも列幅の自動調整自体あまり使い勝手が良くないので、`col_width=` を自分で指定して `auto_size_columns=False` を基本とするのがいいと思います。ちなみに `auto_size_columns=False` を指定しないと `col_width=` は全く効きません。なんだそれ……。

それはさておき: 機能の話に戻ります

行はクリックすることで選択可能で、インデックスは `values` に格納されます。選択モードは `select_mode=` で指定可能で、モードに応じた操作が可能です。

select_mode=	機能
sg.TABLE_SELECT_MODE_NONE	行を選択できない
sg.TABLE_SELECT_MODE_BROWSE	1 行のみ選択可能
sg.TABLE_SELECT_MODE_EXTENDED	複数行を選択可能

ここで選択行の取得をしようとしてやりがちなのが、

```
if values['TABLE'] == 0:
    print('ROW 0 SELECTED!')
```

という書き方です。一見正しいように見えますが、 `Table` 要素の `values` は選択した行の**リスト**で返されるので、0 行目が選択されていたとしても `[0] == 0` を評価することとなり、 `False` になります[2]。

では、以下のように書けばいいのでしょうか。

```
if values['TABLE'][0] == 0:
    print('ROW 0 SELECTED!')
```

これも不十分です。このままでは行が選択されていない状態で別のイベントが発生すると、

```
IndexError: list index out of range
```

と怒られます。選択されている行がない場合は `values` に空のリストが格納されるため、インデックス 0 番を呼び出すことはできません。

これらをふまえ、下記のように書くとよいです。

```
if values['TABLE'] and values['TABLE'][0] == 0:
    print('ROW 0 SELECTED!')
```

こうすれば空のリストであっても `if values['TABLE']` の時点で `False` が確定して[3] `and` 以降がスキップされるため、エラーを回避できます。

一覧に戻る

Graph

```
# 左上(0, 0), 右下(100, 100)
sg.Graph((100, 100), (0, 100), (100, 0), key='GRAPH_2')
```

図形の描画や画像の貼り付けができるエリアです。Graphと言いながら目盛りや軸の表示もないので、どちらかというと `Canvas` と言った方がしっくりきます。要素のサイズ (x, y), 左下座標 (x, y), 右上座標 (x, y) を順番に指定します。この要素も画像処理を行う上で要となりますので、ぜひ覚えてください。

描画には `draw_line()` や `draw_image()` などといったメソッドを使用します。詳しくは後の章で解説します。

一覧に戻る

Column

```
sg.Column(
    [
        [
            sg.Button('ボタン', size=(10, 2))
        ],
        [
            sg.Button('ボタン', size=(10, 2))
        ],
    ]
),
```

複数の要素を一つの要素としてレイアウトできるようになります。レイアウトを入れ子構造にすることで、より複雑なレイアウトが可能になります。 `Table` や `Graph` のような縦に長い要素を配置しながらサイドバー的なものを作るなど、画面を「縦に割りたい」ときに効果を発揮します。

一覧に戻る

デザインの調整: サイズと配置, 色

要素のサイズは中に入れるテキストの長さなどにより自動で調整されますが、もう少し見栄えを整えたいなどはありますよね。PySimpleGUIではレイアウト時に追加のパラメータを設定することでウィンドウの配置を調整することができます。

共通のパラメータ

色

- **text_color**
  - 文字色をカラーコード（ `'#FF0000'` など）もしくは色名（ `'red'` など）で指定します。
- **background_color**
  - 背景色をカラーコード（ `'#FF0000'` など）もしくは色名（ `'red'` など）で指定します。

サイズ・位置

- **size**
  - （幅、高さ）のタプルで指定します。単位は px ではなく**半角文字数**です。
- **pad**
  - 上下左右の余白を指定します。（**左右，上下**）で指定し、 **左右** , **上下** はそれぞれ（**左，右**），（**上，下**）に置き換えることで分けて指定ができます。単位は **px** です。

その他

- **font**
  - フォントを（**フォント名，サイズ**）のタプルで指定します。フォント名は `'メイリオ'` など日本語名でOK、フォントサイズの単位は **pt** です（多分）。サイズのみ変更したい場合は、（**None，サイズ**）とします。

パラメータ早見表

要素名	text_color	background_color	size	pad	font
Button			○	○	○
Text	○	○	○	○	○
Input	○	○	○	○	○
Multiline	○	○	○	○	○
Slider	○	○	○	○	○
Spin	○	○	○	○	○
Combo	○	○	○	○	○
Checkbox	○	○	○	○	○
Radio	○	○	○	○	○
Menu	○	○			○
Table	○	○		○	○
Graph		○		○	
Column		○	○	○	

要素固有のパラメータ

Button

- **button_color**
  - （文字色，ボタン色）をタプルで指定します。

Table

- 各行の揃え方向を指定します。 `'left'`（左揃え），`'center'`（中央揃え），`'right'`（右揃え）から選べます。

- **alternating_row_color**
  - 行の縞模様を作ります。奇数行の背景が指定した色になります。
- **selected_row_colors**
  - 選択中の行の色を （**文字色，背景色**） で指定します。
- **header_text_color**
  - `text_color` と同様にヘッダーの文字色を指定します。
- **header_background_color**
  - `background_color` と同様にヘッダーの背景色を指定します。
- **header_font**
  - `font` と同様にヘッダーのフォントを指定します。
- **row_colors**
  - 特定行の色を指定します。（**行インデックス，文字色，背景色**） もしくは （**行インデックス，背景色**） をまとめたリストで指定します。ex. `row_colors=[(0, 'red', black), (1, 'blue')]`
- **col_widths**
  - 列ごとの幅をリストで指定します。 `auto_size_columns=False` を指定する必要があります。
- **num_rows**
  - 表示する行数を指定します。表示しきれない分はスクロールとなります。

Column

- **scrollable**
  - 範囲外の要素をスクロールで表示されるようにします。
- **vertical_scroll_only**
  - `scrollable=True` のとき、縦方向のスクロールのみ許可します。
- **element_justification**
  - `Column` **内の要素の**横方向の揃え方向を `'left'`（左揃え），`'center'`（中央揃え），`'right'`（右揃え）から指定します。
- **vertical_alignment**
  - `Column` **自体の**縦方向の揃え方向を `'top'`（上揃え），`'center'`（中央揃

- **expand_x**
  - 横方向の余白いっぱいに要素サイズを拡張します。
- **expand_y**
  - 縦方向の余白いっぱいに要素サイズを拡張します。

メソッドの活用

これまではユーザー操作をプログラムに反映させる方法について解説してきましたが、今度は逆にプログラムからインターフェースを変化させる方法について説明していきます。

要素の取得方法

```
# 変数inputにkey='INPUT'をもつ要素オブジェクトを格納
input = window['INPUT']
# 対象の要素にupdate()メソッドを実行
input.update()
# 変数を介さず直接実行してもOK
window['INPUT'].update()
```

メソッドを使うためにはまず、要素のオブジェクトを取得する必要があります。
`window` には辞書型のように要素オブジェクトが格納されており、レイアウト時に `key=` で指定した文字列をキーとして要素オブジェクトを取得することができます。

共通メソッドとGraph固有メソッド

すべての要素に共通なメソッドには以下の 3 つがあります。

メソッド	概要
update()	要素の持つ値を更新する

メソッド	概要
get_size()	要素のサイズを取得する

またスクロール可能な `Table`，`Column` 要素にはスクロール制御用の、様々な図形を描画できる `Graph` 要素に描画用のメソッドがあるため、そちらもあわせて解説します。

update()

要素の値や外観を更新します。元の値はクリアされるので、現在の内容に追加したい場合は工夫が必要です。以下に例を示します。

```
# 元のテキスト末尾に「！」を追加
window['INPUT'].update(values['INPUT'] + '！')
```

`values` で現在値を取得し、その後ろに追加したい文字列 `'！'` を追加して更新することで、現在値の末尾に `'！'` が追加されたように見えます。

要素ごとに更新される値は以下の通りです。

要素名	変化する値
Button	ボタンテキスト
Text	テキスト
Input	テキスト
Multiline	テキスト
Slider	現在値
Spin	現在値
Checkbox	チェック有無（True / False）

要素名	変化する値
Menu	新しいメニュー構造
Table	テーブル構造
Graph	背景色
Column	=False で非表示

その他のパラメータの更新

表に示したのはあくまで第一引数のみで、その他の引数を指定することで他のパラメータについても変更が可能です。詳細は PySimpleGUI の Call reference で確認できます。

Call reference - PySimpleGUI
https://pysimplegui.readthedocs.io

ここでは例として使用頻度の高い `Table` の `select_rows=` を挙げます。

```
window['TABLE'].update(select_rows=[])
```

`select_rows=` に選択したい行のインデックスの**リスト**を与えると、その行が選択された状態になります。空のリストを与えるとすべての選択が解除されます。引数にリストではなく `int` を渡してしまうと、

```
    rows_to_select = [i + 1 for i in select_rows]
TypeError: 'int' object is not iterable
```

と怒られてしまいます。

表示されていない行を選択するのもNGです。3 行しか表示していない `Table` に `select_rows=[10]` などとやってしまうと、

```
    self.tk.call(self._w, "selection", selop, items)
_tkinter.TclError: Item 11 not found
```

といって怒られることになります。空のテーブルに対して `select_rows=[0]` とするのも同様にNGです。

bind()

```
# key='GRAPH'をもつ要素にドラッグ操作を追加
# ドラッグを検知すると event='GRAPH__DRAG' となる
window['GRAPH'].bind('<Button1-Motion>', '__DRAG')
```

任意のアクションをイベントとして捕捉することができます。画像処理においては「画像上にあるカーソルの座標を取得」、「表示した画像から ドラッグ & ドロップで矩形選択」といった複雑なアクションをしたいことも多く、特に `Graph` との組み合わせが有効です。

`bind()` は一度実行すればウィンドウを閉じるまで有効ですが、必ず `window.finalize()` してから行います。 `finalize()` 前にバインドしようとすると、

```
ERROR  Unable to complete operation on element with key ***
```

と怒られます。

`bind()` は 2 つの引数 `bind_string` , `key_modifier` をもちます。

bind_string

捕捉したいイベントを格納する `bind_string` は**修飾子**、**種類**、**詳細**を `-` （ハイフン） でつなげた文字列を `<>` で囲んで記述します。例えば `'<Control-Shift-S>'` は Ctrl + Shift + s キー、 `'<Button1-Motion>'` はマウスドラッグです。**修飾子**は何個でも連結することができますが、**詳細**は 1 個しか指定できません。

修飾子（modifier）

文字列	概要
Control	Ctrl キーを押しながら
Shift	Shift キーを押しながら
Alt	Alt キーを押しながら
Button1, B1	マウス左ボタンを押しながら
Button2, B2	マウスホイールボタンを押しながら
Button3, B3	マウス右ボタンを押しながら
Double	操作の 2 回繰り返し
Triple	操作の 3 回繰り返し

種類（type）

文字列	概要
KeyPress, Key, 省略[4]	キーボードを押す
KeyRelease	キーボードを離す
Return	Enterキーを押す
ButtonPress, Button	マウスボタンを押す
ButtonRelease	マウスボタンを離す

文字列	概要
Motion	要素の上でカーソルを動かす
Enter	要素の内側にカーソルを入れる
Leave	要素の外側にカーソルを出す

詳細（detail）

文字列	概要
1	マウス左クリック
2	マウスホイールクリック
3	マウス右クリック
a, b, c, … , z, A, B, C, … , Z その他記号	対応するキー

キー入力については**大文字小文字が区別されます。** `'<Control-Shift-s>'` などと書いてしまうと全然反応しないので注意が必要です（Shift + s の組み合わせが大文字 `S` として認識されるため）。Caps Lock がかかると文字の大小は逆転するので、厳密には `s` と `S` の両方を `bind()` すべきなのですが、個人的にはそこまでしなくてもいいかなとも思います。

key_modifier

`bind_string=` で指定したユーザー操作を捕捉したときに、他のイベントと区別するため末尾に修飾子をつけることができます。

```
window['GRAPH'].bind('<Button1-Motion>', '__DRAG')
```

のように書いた場合、この操作を捕捉すると従来のキー `'GRAPH'` に `'__DRAG'` を

Table, Column 用メソッド

set_vscroll_position()

```
# 一番上にスクロール
window['TABLE'].set_vscroll_position(0)
# 中央にスクロール
window['TABLE'].set_vscroll_position(0.5)
# 一番下にスクロール
window['TABLE'].set_vscroll_position(1)

# 選択中の位置に合わせてスクロール
# 0 除算回避のため len(list)-1 とはしない
window['TABLE'].update(list)
window['TABLE'].set_vscroll_position(values['TABLE'][0]/len(list))
```

スクロール位置を 0 ~ 1 の範囲で指定することができます。 `Table.update()` で内容を更新したり選択行を変更したとき、その行が画面外にあっても自動でスクロールはしてくれません。なのでこのメソッドを使い必要な位置にスクロールしてやる必要があります。

Graph 用メソッド

`Graph` はメソッドを利用して図形や文字、画像の描画が可能です。主な描画関連のメソッドを表にまとめました。

メソッド一覧

メソッド	概要
draw_point()	点を描画する
draw_line()	線を描画する
draw_rectangle()	矩形を描画する

メソッド	概要
draw_circle()	円を描画する
draw_text()	テキストを描画する
draw_image()	画像を描画する
move_figure()	描画済みの図を移動する
move()	描画全体を移動する
bring_figure_to_front()	描画済みの図を最前面に移動する
send_figure_to_back()	描画済みの図を最背面に移動する
delete_figure()	図を削除する
erase()	描画全体を削除する
change_coodinates()	左下、右上座標を変更する

draw_point()

```
# 座標(10, 20)に直径5, 赤色の点を描画
window['GRAPH'].draw_point((10, 20), size=5, color='#FF0000')
```

一覧に戻る

draw_line()

```
# 座標(10, 20)と(110, 120)を結ぶ幅1, 緑色の線を描画
window['GRAPH'].draw_line((10, 20), (110, 120), color='#00FF00', width=1)
```

一覧に戻る

draw_rectangle()

```
# 座標(10, 20)と(110, 120)を対角線とする矩形を描画。塗りつぶしは青、線は黄色、線幅は1
window['GRAPH'].draw_rectangle(
    (10, 20),
    (110, 120),
    fill_color='#0000FF',
    line_color='#FFFF00',
    line_width=1
)
```

一覧に戻る

draw_polygon()

```
# pointsに指定した点を結ぶ三角形を描画。塗りつぶしなし、線は赤、線幅は1
points=[
    (10, 20),
    (110, 120),
    (210, 100),
]
window['GRAPH'].draw_polygon(
    points,
    fill_color=None,
    line_color='#FF0000',
    line_width=1
)
```

一覧に戻る

draw_circle()

```
# 座標(10, 20)に半径30の円を描画。塗りつぶしは青、線なし
window['GRAPH'].draw_circle(
    (10, 20),
    30,
    fill_color='#0000FF',
    line_color=None,
    line_width=None
)
```

一覧に戻る

draw_text()

```
# 文字中央を基準として座標(110, 20)に白で'Text'と表示
# フォントはメイリオ 12pt、文字の回転はなし
window['GRAPH'].draw_text(
    'Text',
    (110, 20),
    color='#FFFFFF',
    font=('メイリオ', 12),
    angle=0,
    text_location=sg.TEXT_LOCATION_CENTER
)
```

一覧に戻る

draw_image()

```
import cv2
# OpenCVで取り込んだ画像imgを変換して、画像左上を基準として座標(10, 20)に表示
img = cv2.imread('image.png')
img_bytes = cv2.imencode('.png', img)[1].tobytes()
window['GRAPH'].draw_image(data=img_bytes, location=(10, 20))
```

一覧に戻る

move_figure()

```
# draw_point()で描画した点を横に20移動、縦に-10移動
id_ = window['GRAPH'].draw_point((10, 20), size=5, color='#FF0000')
window['GRAPH'].move_figure(id_, 20, -10)
```

一覧に戻る

```
# 描画全体を横に20移動、縦に-10移動
window['GRAPH'].move(20, -10)
```

一覧に戻る

bring_figure_to_front()

```
# 矩形に隠れてしまった点を前面に移動
point = window['GRAPH'].draw_point((50, 50), size=5, color='#FF0000')
rect = window['GRAPH'].draw_rectangle(
    (0, 0),
    (100, 100),
    fill_color='#0000FF',
)
window['GRAPH'].bring_figure_to_front(point)
```

一覧に戻る

send_figure_to_back()

```
# 点を隠してしまった矩形を背面に移動
point = window['GRAPH'].draw_point((50, 50), size=5, color='#FF0000')
rect = window['GRAPH'].draw_rectangle(
    (0, 0),
    (100, 100),
    fill_color='#0000FF',
)
window['GRAPH'].send_figure_to_back(rect)
```

一覧に戻る

delete_figure()

```
# 描画した点を削除
id_ = window['GRAPH'].draw_point((10, 20), size=5, color='#FF0000')
```

一覧に戻る

erase()

```
# 描画をすべて削除
window['GRAPH'].erase()
```

一覧に戻る

change_coodinates()

```
# グラフ内の座標を左上(0, 0), 右下(640, 480)に変更
window['GRAPH'].change_coordinates((0, 480), (640, 0))
```

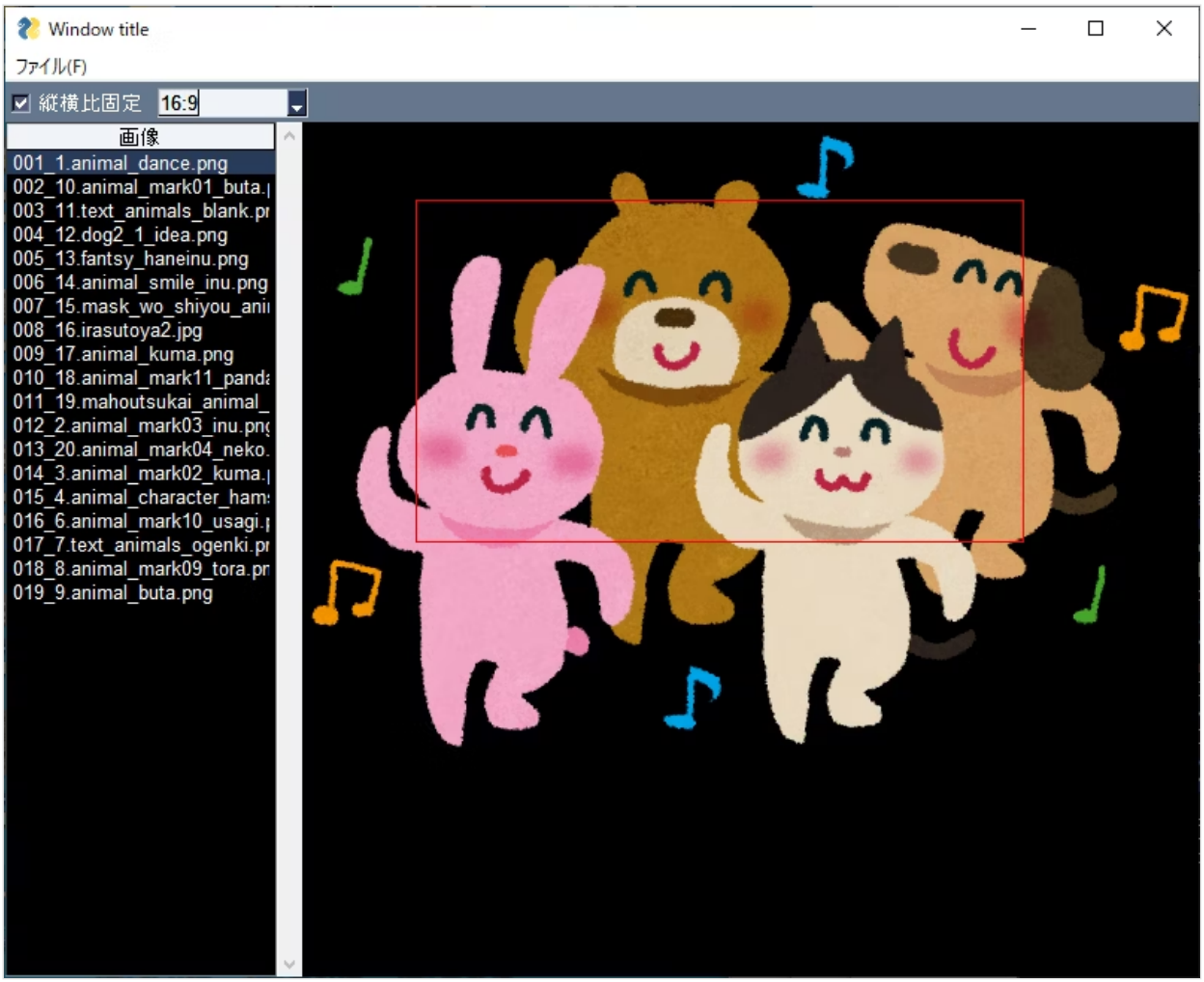
一覧に戻る

実践編

PySimpleGUIを用いた実装例として、画像ビューワーを作成してみます。

画像ビューワ概要

動作イメージは下図の通りです。



ファイル ＞ 開く と選択するとフォルダ選択ダイアログが表示されます。フォルダを選択すると、その中にある画像ファイルが画面左側のリストに一覧表示されます。一覧から表示する画像を選び、表示範囲を選ぶことで拡大ができます。表示範囲は縦横比の指定が可能です。

操作一覧

操作	アクション
リストをクリック	表示画像を選択する
画像上でマウススクロール	表示画像の順送り / 逆送り
画像上でドラッグ	表示範囲の選択

操作	アクション
画像上でダブルクリック	表示範囲の選択解除

コード

以下のコードをコピペすれば動作します。

▶ コード全文を表示

ポイント

以下の処理に分けて解説します。

- cv2.imread() の改良
- Graph に画像表示用メソッドを追加
- Column による表示切り替え
- レイアウト
- Graph 要素のセットアップ
- 描画エリアサイズ変更検知
- ファイル一覧を取得し、Table に一覧表示
- Table の操作
- 画像の矩形選択
- 選択範囲の確定
- 選択範囲のリセット
- 画像の更新
- 画像表示
- 選択範囲表示

cv2.imread() の改良

```
# 画像読込
```

```
    try:
        n = np.fromfile(filename, dtype)
        img = cv2.imdecode(n, flags)
        return img
    except Exception as e:
        print(e)
        return None
```

いきなり PySimpleGUI じゃないですが・・・。

OpenCV の画像読み込み関数 `cv2.imread()` は日本語パスに対応していません。今回はユーザー指定のフォルダからファイルを読み込むので、日本語パスに対応する必要があります。下記の記事より `imread()` 関数を拝借しました。

一覧に戻る

Graph に画像表示用メソッドを追加

```
# 画像を表示（sg.Graph インスタンスメソッド）
def draw_image_plus(self, img, location=(0,0)):
    if type(img) == np.ndarray:
        img = cv2.imencode('.png', img)[1].tobytes() # cv2.imencode('.png', img)[0] はエ
    id_ = self.draw_image(data=img, location=location)
    return id_
sg.Graph.draw_image_plus = draw_image_plus
```

`Graph` には画像を描画する `draw_image()` というメソッドがありますが、`imread()` で取り込んだ画像をそのまま表示することはできません。`cv2.imencode()` 関数により画像を `image` オブジェクト化する必要があります。今回は無劣化で表示したいので png 形式でエンコードします。その後 `tobyte()` でバイナリ化すると `Graph` で表示できるようになります。

この処理を画像表示のたびに書くのは非効率なので関数化します。さらにその関数を `Graph` 要素のメソッドとして組み込むことで、他の描画用メソッドと同じような書

一覧に戻る

Column による表示切り替え

```
sg.Column(
    [
        [
            sg.Input('', size=(5, 1), pad=(0, 0), key='ASPECT_X'),
            sg.Text(' : ', pad=(0, 0)),
            sg.Input('', size=(5, 1), pad=(0, 0), key='ASPECT_Y'),
        ]
    ],
    visible=False,
    key='COLUMN_ASPECT',
)
```

横一列に要素が並んでいるだけで、一見すると `Column` は必要ないように見えます。しかしこうすることで、

```
# 表示
window['COLUMN_ASPECT'].update(True)
# 非表示
window['COLUMN_ASPECT'].update(False)
```

のように表示を切り替えることができます。ただし非表示→表示とすると**なぜか行の右端に出現する**ので、右側に別の要素を配置している場合はレイアウトが崩壊します[5]。

一覧に戻る

レイアウト

```
# Graph
canvas = sg.Graph(
    (1920, 1920), # 大きめに作って画面外にはみ出させる
    (0, 1920), # 表示サイズに合わせる
```

```
    pad=(0, 0),
    key='CANVAS',
)

# Window
window = sg.Window(
    title='Window title',
    layout=layout,
    resizable=True,
    size=(800, 600),
    margins=(0, 0),
)
```

あらゆる要素に `pad=0` を指定してカツカツのレイアウトにしています。 `Window` も `margins=(0, 0)` を指定することで窓枠ギリギリまで要素を置くことができます。

一覧に戻る

Graph 要素のセットアップ

```
# 定義
canvas = sg.Graph(
    (1920, 1920), # 大きめに作って画面外にはみ出させる
    (0, 1920), # 表示サイズに合わせる
    (1920, 0),
    background_color='#000000',
    pad=(0, 0),
    key='CANVAS',
)

# イベントバインド
canvas.bind('<ButtonPress-1>', '__LEFT_PRESS') # テーブル選択
canvas.bind('<MouseWheel>', '__SCROLL') # 表示画像のスクロール変更
canvas.bind('<Button1-Motion>', '__DRAG') # ドラッグで範囲選択
canvas.bind('<Button1-ButtonPress-3>', '__DRAG_CANCEL') # ドラッグ中止（ドラッグ中に右ク
canvas.bind('<ButtonRelease-1>', '__LEFT_RELEASE') # ドラッグ範囲確定
canvas.bind('<Double-ButtonPress-1>', '__DOUBLE_LEFT') # 選択範囲解除

# メンバ変数定義
canvas.drag_from = None # ドラッグ開始位置
```

```
canvas.selection = None # 選択範囲
canvas.selection_figure = None # 選択範囲の描画ID
```

Graph 要素は画像処理 GUI で特に使用頻度が高いので使い勝手を上げます。メソッド呼び出すごとに window['CANVAS'] は面倒なので、短めの変数に要素を格納しておきます。あとは必要なイベントをバインドして、カーソル位置や選択範囲などのパラメータを定義しておきます。 Graph 操作関連のパラメータはメンバ変数として定義しておいたほうが拡張性がある[6]のでおすすめです。

一覧に戻る

描画エリアサイズ変更検知

```
# 描画エリアサイズ監視用にタイムアウトを指定
event, values = window.read(timeout=100, timeout_key='TIMEOUT')

# 描画エリアサイズ変更の検出
    if event == 'TIMEOUT':
        if previous_canvas_size != canvas.get_size():
            event = 'CANVAS_RESIZE'
        previous_canvas_size = canvas.get_size()
        if event == 'TIMEOUT':
            continue
```

ウィンドウサイズに合わせて画像の大きさを調節するため、描画エリアサイズの変更を捕捉します。正攻法ではイベントとして登録できないので、リアルタイムにウィンドウサイズを監視して、変化があった場合は手動でイベントを起こします。リアルタイム処理を行う場合は window.read() に timeout= を指定します。単位はミリ秒です。あわせて timeout_key= を指定すると、 timeout= 時間内に何もイベントがおきなければ指定したキーでイベントを発生させます。 canvas.get_size() で描画エリアのサイズを検出し前回の結果と比較、変化があれば event='CANVAS_RESIZE' として手作りイベントを発生させます。

一覧に戻る

```
import tkinter as tk

# フォルダを開く
if '::MENU_OPEN_FOLDER::' in event:
    source_dir = tk.filedialog.askdirectory().replace('[', '[[]').replace(']', '[]]').re
    if source_dir:
        trim_areas = {}
    # ファイル一覧を取得
        fullpath_list = glob.glob('{}/*.png'.format(source_dir)) \
            + glob.glob('{}/*.jpg'.format(source_dir)) \
            + glob.glob('{}/*.jpeg'.format(source_dir)) \
            + glob.glob('{}/*.bmp'.format(source_dir)) \
            + glob.glob('{}/*.gif'.format(source_dir))
        fullpath_list.sort()
        fullpath_list = [s.replace('\\', '/') for s in fullpath_list]
        if fullpath_list:
            select_rows=[0]
        else:
            select_rows=[]
    # ファイル一覧更新
        window['TABLE_SOURCE'].update([[s.split('/')[-1]] for s in fullpath_list], sele
```

フォルダを指定して中にある画像ファイルを一覧取得します。フォルダ選択ダイアログには tkinter を使用します。PySimpleGUI にも `FolderBrowse()` というフォルダ選択用の要素がありますが、ボタンの形に縛られたり、値の更新に `window.read()` が必要だったりと制約が多いです。その分組み込みが楽だったりはあるので良し悪しですが、迷ったら `tkinter` で良いと思います。

中にあるファイルの一覧取得には `glob()` を使います。指定した文字列に合致するファイル（とフォルダ）をリストで返します。 `*` はワイルドカードで任意の文字列にマッチします。ちなみに `?` は任意の 1 文字にマッチします。

また `[` , `]` は「`[]` 中の 1 文字にマッチ」を表す文字列なので、 `[[]` , `[]]` に置換します。 `replace('[', '[[]')` , `replace(']', '[]]')` と置換を繰り返すと `[` から `[[]` に置換されたものが `[[[]]` に再置換されるので `[[]` に戻す必要があります。スマートにやりたい方は正規表現を使いましょう。

`Table.update()` メソッドには**二次元リスト**を渡します。内包表記を使って、1次元

一覧に戻る

Table の操作

```
# 選択スクロール
    if event == 'CANVAS__SCROLL' and values['TABLE_SOURCE']:
        row = values['TABLE_SOURCE'][0]
        item_len = len(fullpath_list)
        if canvas.user_bind_event.delta > 0 and row > 0:
            row -= 1
        elif canvas.user_bind_event.delta < 0 and row < item_len - 1:
            row += 1
        window['TABLE_SOURCE'].update(
            [[s.split('/')[-1]] for s in fullpath_list],
            select_rows=[row],
        )
```

マウスホイールの動きに合わせて選択画像を切り替えられるようにします。
`select_rows=` に負数や行数以上の数が入らないように条件を指定します。最初に `values['TABLE_SOURCE'][0]` 選択行を指定していますが、選択していない状態だとエラーが発生するため、 `if values['TABLE_SOURCE']` で回避します[3]。マウスホイールの動きに合わせて `user_bind_event.delta` に値が代入されます。上回転が +, 下回転が - ですので、それに合わせて行番号を増減させます。

一覧に戻る

画像の矩形選択

```
canvas.bind('<ButtonPress-1>', '__LEFT_PRESS') # 範囲選択開始
canvas.bind('<Button1-Motion>', '__DRAG') # ドラッグで範囲選択
canvas.bind('<Button1-ButtonPress-3>', '__DRAG_CANCEL') # ドラッグ中止（ドラッグ中に右ク
canvas.bind('<ButtonRelease-1>', '__LEFT_RELEASE') # ドラッグ範囲確定

# 矩形選択開始
    if event == 'CANVAS__LEFT_PRESS':
        canvas.drag_from = np.array((canvas.user_bind_event.x, canvas.user_bind_event.y)
        canvas.current = np.array((canvas.user_bind_event.x, canvas.user_bind_event.y))
```

```
            canvas.current = np.array((canvas.user_bind_event.x, canvas.user_bind_event.y))
            canvas.selection = np.array((canvas.drag_from, canvas.current))
            canvas.selection = np.array((canvas.selection.min(axis=0), canvas.selection.max(
        # アスペクト比の適用
            if aspect is not None:
                selection_size = (canvas.selection[1] - canvas.selection[0])
                aspected = (aspect[0]/aspect[1]*selection_size[1], aspect[1]/aspect[0]*selec
                canvas.selection = np.vstack([canvas.selection, [aspected]]) # アス比適応時
            canvas.selection = np.array((canvas.selection.min(axis=0), canvas.selection.max(
    # 矩形選択キャンセル
        if event == 'CANVAS__DRAG_CANCEL':
            canvas.selection = None
            canvas.drag_from = None
```

画像上をドラッグすることで範囲を指定できるようにします。

左クリックでドラッグの起点座標 `canvas.drag_from` を設定します。カーソルの座標は `canvas.user_bind_event.x` , `canvas.user_bind_event.y` で取得可能です。ただし、取得できる座標は**左上基準のピクセル単位**で固定です。Graphをレイアウトしたときに指定した座標系は関係ないので注意してください[7]。

ドラッグ中はカーソル現在地 `canvas.current` を更新し続け、 `canvas.drag_from` と `canvas.current` の組み合わせで選択範囲 `canvas.selection` を定義します。このとき2 点の位置関係が不明なので、

```
canvas.selection = np.array((canvas.drag_from, canvas.current))
canvas.selection = np.array((canvas.selection.min(axis=0), canvas.selection.max(axis=0)
```

として x, y それぞれの最小値（左上）と最大値（右下）に振り分けます。この辺りの計算を楽にするために、座標を `numpy.array` で定義しています。アスペクト比の適用については割愛します。そんなもんなんだな～位に思っておいてください。

ドラッグ中に右クリックを押すと、 `canvas.selection` と `canvas.drag_from` をリセットして範囲選択を取り消します。

一覧に戻る

```
    # 矩形選択完了
        current_is_key = current_fullpath in list(trim_areas.keys()) # 記録済みの選択範囲が
        if event == 'CANVAS__LEFT_RELEASE' and canvas.selection is not None:
        # 面積0の選択範囲はスキップ
            if (canvas.selection[1] - canvas.selection[0]).min() >= 1:
                canvas.selection = (canvas.selection.astype(float)*image_scale).astype(int)
            # 記録済みの選択範囲がある場合はオフセットする
                if current_is_key:
                    canvas.selection += trim_areas[current_fullpath][0]
            # 選択範囲の記録
                trim_areas[current_fullpath] = canvas.selection
        # 範囲を記録したらリセット
            canvas.selection = None
            canvas.drag_from = None
        current_is_key = current_fullpath in list(trim_areas.keys())
```

ドラッグした状態からマウスを離すと選択範囲を確定し、変数 `trim_areas` に登録します。 `trim_areas` は画像のファイルパスと選択範囲を組み合わせた辞書型の変数です。

まず選択範囲の大きさを確認し、高さ / 幅が 0 であれば登録しません。矩形選択が可能な形状であれば、選択範囲を画像の表示倍率 `image_scale` で拡大 / 縮小します。

すでに一度範囲の登録を終えていて拡大表示されている場合は、 `Graph` 左上 ≠ 元画像原点となるのでそのまま登録できません。

```
    # 記録済みの選択範囲がある場合はオフセットする
    if current_is_key:
        canvas.selection += trim_areas[current_fullpath][0]
```

登録済みの選択範囲を参照し、左上座標のぶんだけ全体をオフセットします。

一覧に戻る

選択範囲のリセット

```
        trim_areas.pop(current_fullpath)
        current_is_key = False
```

ダブルクリックをトリガーにして選択範囲をリセットします。 `pop()` メソッドを使って現在の選択範囲を削除します。

一覧に戻る

画像の更新

```
# 画像更新
if event in ('TABLE_SOURCE', 'CANVAS__LEFT_RELEASE', 'CANVAS__DOUBLE_LEFT'):
    filename = fullpath_list[values['TABLE_SOURCE'][0]]
    img = imread(filename)
# 登録済みの選択範囲があればトリミングする
    if current_is_key:
        rect = trim_areas[current_fullpath]
        img_trim = img[rect[0, 1]:rect[1, 1], rect[0, 0]:rect[1, 0]]
    else:
        img_trim = img.copy()
    img_update = True
```

画像の選択や選択範囲の登録・解除に関わる操作があった場合は画像を更新します。先に解説した `imread()` 関数で画像を表示します。

`trim_areas` に選択範囲が登録されている場合は、その範囲に応じて画像をトリミングします。 `img[上:下, 左:右]` とスライスを指定すると、その範囲のみの画像が得られます。

後ろの描画処理に更新したことを伝えるため `img_update` を `True` にします。

一覧に戻る

画像表示

```
# 画像表示（画像が更新された場合か、ウィンドウがリサイズされた場合）
```

```
        canvas_size = np.array(canvas.get_size())
    # キャンバス比で長い方の割合をsceleとする
        image_scale = (img_size / canvas_size).max()
    # キャンバスに対して長い方を基準に縮小するので、画像が画面外にはみ出ない
        img_resize = cv2.resize(img_trim, tuple((img_size/image_scale).astype(int)))
    # 画像端座標を取得
        img_area_limit = ((np.array(img_resize.shape[1::-1])-1))
    # キャンバスリセット→画像表示
        canvas.erase()
        canvas.draw_image_plus(img_resize)
        img_update = False
```

画像が更新された、もしくはウィンドウサイズが変更された場合は画像を表示しなおします。画像をそのまま表示すると、画面に対して小さすぎたり、逆に画面からはみ出てしまったりするので、ウィンドウサイズに合わせて画像をリサイズして表示します。

画像のサイズ取得には `shape` を使います。`imread()` 関数で画像は配列 `numpy.ndarray` となり、`shape` にアクセスすると（縦画素数，横画素数，チャンネル数）のタプルを得ることができます[8]。ここで得られた画像サイズ `img_size` とキャンバスサイズ `canvas_size` の比を縦横それぞれで計算し、その最大値をリサイズの倍率にします。

得られた倍率から、`cv2.resize()` を使って画像をキャンバスサイズに合わせてリサイズします。この関数には画像と変更後のサイズ（幅，高さ）を渡しますが、`numpy.ndarray` を渡すと、

```
SystemError: new style getargs format but argument is not a tuple
```

というエラーが発生してしまいます。必ずリストなりタプルに変換して渡しましょう。リサイズが終われば、そのサイズを選択可能範囲 `img_area_limit` として登録します。こうすることで画像でない範囲を選択してエラーが起きるのを防ぐことができます。

その後はリサイズした画像を自作のメソッド `draw_image_plus()` に渡して画像を表示させます。このときキャンバスに残った画像を削除しないと、表示する画像がどん

完了したら `image_update` を `False` に戻します。

一覧に戻る

選択範囲表示

```
# 選択範囲表示
if canvas.selection_figure is not None:
    canvas.delete_figure(canvas.selection_figure)
if canvas.selection is not None:
    canvas.selection_figure = canvas.draw_rectangle(
        list(canvas.selection[0]),
        list(canvas.selection[1]),
        line_color='#FF0000',
        line_width=1
    )
```

ドラッグ中の選択範囲を表示します。後から描画した要素が上に表示されるので、画像の描画をしてから矩形を描画します。画像の描画時と同様に描画済みの図形をリセットする必要があるので、 `delete_figure()` メソッドを使って描画済みの図形を削除してから描画します。

一覧に戻る

おわりに

お疲れさまでした。自分自身このライブラリを使う中であちこち参照して回るのが億劫になり、全体を網羅的にまとめたい！と思ったら相当なボリュームになってしまいました・・・。

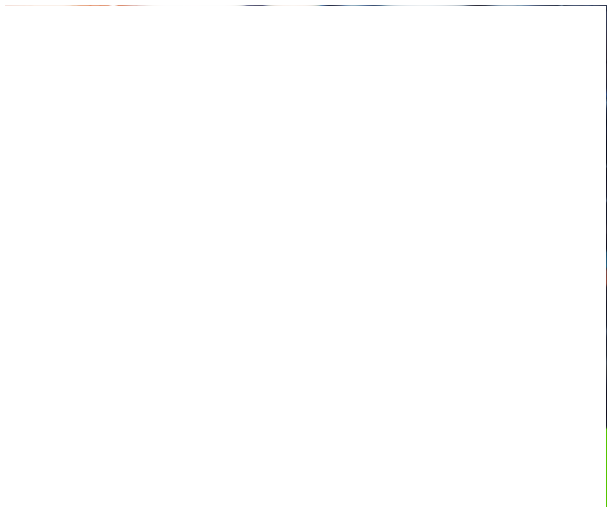
とはいえ PySimpleGUI を使えばかんたんなGUIをサクッと作ることから、マルチウィンドウを駆使した複雑な構成まで可能になっています。公式の解説も英語ではありますが、中々親切です。

参考

- PySimpleGUI
  - https://pysimplegui.readthedocs.io/en/latest/
- Tkinterを使うのであればPySimpleGUIを使ってみたらという話
  - https://qiita.com/dario_okazaki/items/656de21cab5c81cabe59
- PysimpleGUIを用いたOpenCV画像処理表示
  - https://qiita.com/Gyutan/items/b4781ee1baa4966699ef
- Tkinter の bind とイベントシーケンス
  - http://www.rouge.gr.jp/~fuku/tips/python-tkinter/bind.shtml
- Python OpenCV の cv2.imread 及び cv2.imwrite で日本語を含むファイルパスを取り扱う際の問題への対処について
  - https://qiita.com/SKYS/items/cbde3775e2143cad7455

---

1. `'[文字列B]' in '[文字列A]'` は `'[文字列B]'` が `'[文字列A]'` に含まれる場合に `True` になります。 ↩
2. これは `TABLE_SELECT_MODE_BROWSE` を選んで複数行の選択を禁止したとしても変わりません。 ↩
3. `if` 文に空のリストを渡すと `False` 扱いとなります。 ↩ ↩2
4. 種類を書かず詳細をそのまま書くということです。 `<h>`, `<Control-c>` など。 ↩
5. 対策知ってる方いたら教えてください。 ↩
6. 2 画面表示などでも個別のメンバ変数として一意な名前付けができる、など。 ↩
7. values で取得できる座標だけはレイアウト時の座標系に従います。 ↩
8. モノクロ画像の場合は（縦画素数，横画素数）になります。 ↩



関連記事 Recommended by

PySimpleGUIでグラフを描く

by dario_okazaki

Pythonでも簡単にGUIは作れる

by konitech913

SVGとD3.jsの入門まとめ

by simonritchie

PySimpleGUIでVBAの代わりになるUIをつくってみる（ファイルダイアログ、リスト、ログの出...

by dario_okazaki

＜ITエンジニア採用募集中＞日立のデジタル人財ストーリーはこちら

PR 株式会社日立製作所

リモートでプロジェクト遂行！テクノプロ・デザイン社での働き方をインタビュー

PR テクノプロ・デザイン社

コメント

この記事にコメントはありません。

あなたもコメントしてみませんか :)

新規登録

すでにアカウントを持っている方はログイン

72 104