

Pythonでつくる宣言的UIラッパーフレームワーク

既存GUIフレームワークの調査を添えて

2021/10/16 14:50 - 15:20 @ PyCon JP 21

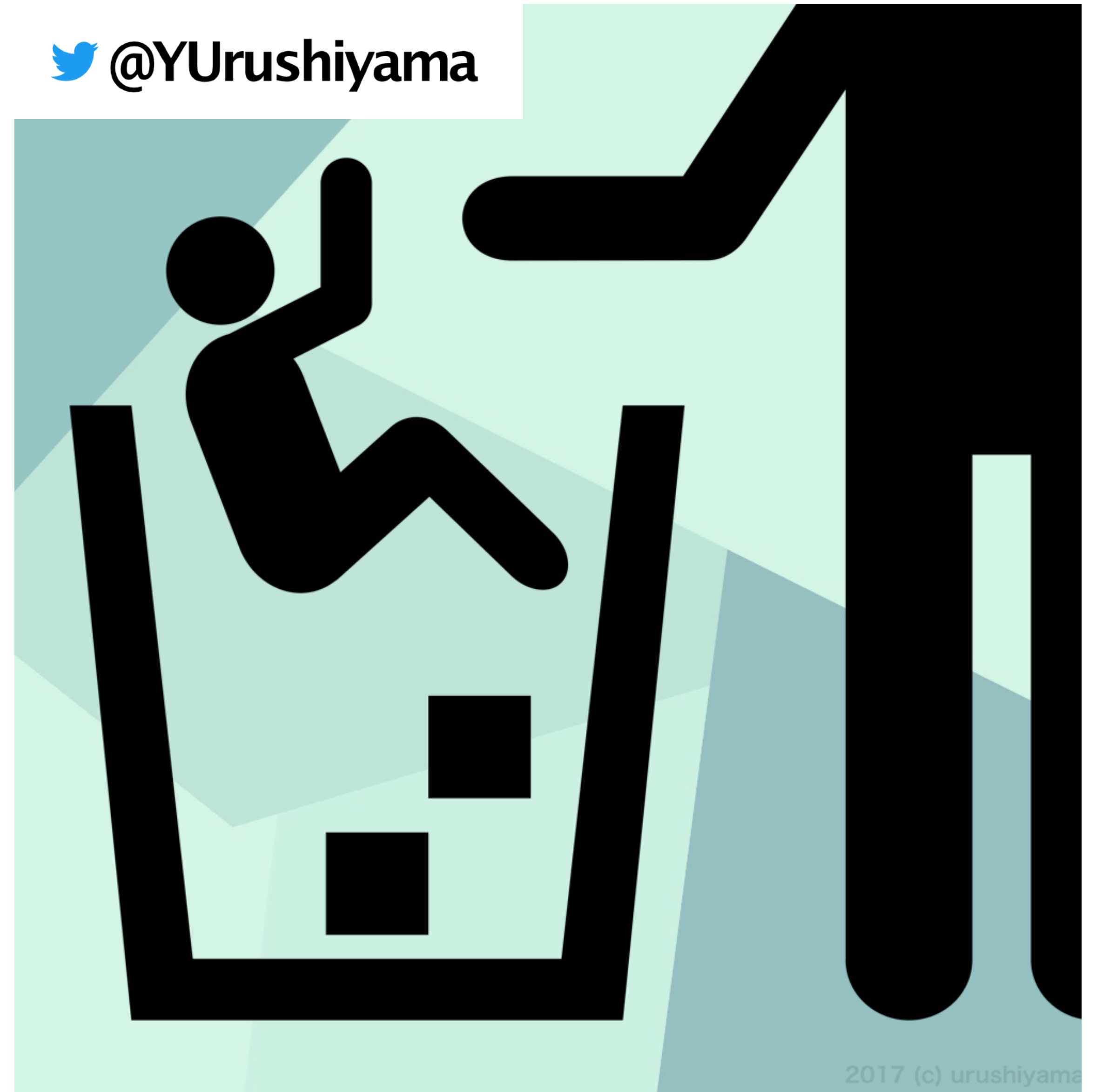
Yuta Urushiyama

自己紹介

漆山 裕太 (Yuta Urushiyama)

- ・ 2021卒
 - とある農業と会計のIT企業
- ・ Pythonとの関わり
 - ロパク動画作成ツール
 - PyPIでライブラリ公開
 - DS/社内業務システム (OJT)
 - 社内資産のAPIサーバ化

 @YUrushiyama



Python × GUI

経緯

口パク動画作成ツールにGUIをつけたい！

- CLIは作成済み

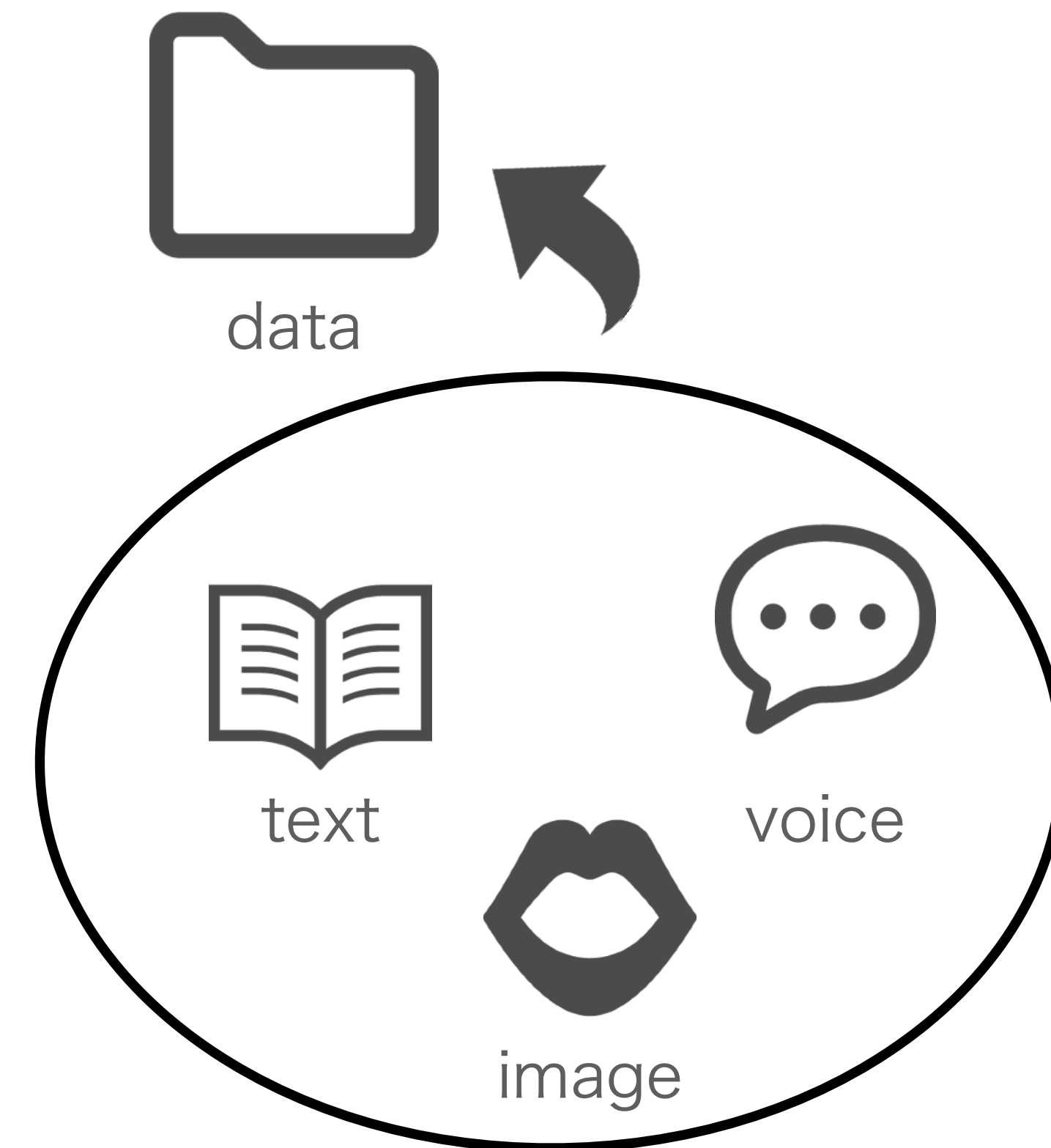
```
$ python convert.py -i data/ -o target/
```

- ツールの特性上マウスだけで操作できると楽ちん

口ジックはもうできてるし
楽勝でしょ



バカ



経緯

あれ？PythonでGUIつくるの（他言語と比べて）面倒くさい？

- Tkinterは命令的
 - コードから見た目を想起しづらい
- PySimpleGUIはスッキリしているけど…
 - PythonでLGPLv3なのがちょっと気になる
- Kivyはどうしてコードとビューの宣言がわかれてるの？

経緯

React "Native"世代^{^1}としての不満 -> 創作意欲

Reactみたいな宣言的UIフレームワークないかなあ～



Pythonでも宣言的UIフレームワークつくれるのかな～



よし、いっちょ作ってみますか！

^{^1}: Reactは2013年にオープンソース化したので、高校時代にプログラミングに興味を持った20代はGUI=宣言的UIという思考になりやすい？

トピック

- これから話すこと
 - Pythonで使えるGUIフレームワークの宣言的UI視点での整理
 - PythonでReactの仮想DOMを模倣した話
- 話さないこと
 - データバインディング
 - イベントハンドリング

宣言的UIとは

「宣言的UI」とは？

歴史的経緯 | 命令的UI

- ・ 2000年代までのGUIは命令的UIライブラリが主流
 - どの階層、どの位置に部品を置くかをコードで逐一制御
 - 4:3ないし16:9の画面の前でキーボードとマウスを操作する業務用パッケージソフトを作るのには適していた
 - ▶ WFなら画面設計は1度きり()
 - ▶ ウィンドウ縮小もサポートする最小解像度まででOK

「宣言的UIフレームワーク」とは？

歴史的経緯 | 命令的UI

- ・ 2000年代までのGUIは命令的UIライブラリが主流

- どの階層、どの位置に部品を置くかをコードで記述

- 4:3ないし16:9の画面の前でキーボード入力

- 業務用パッケージソフトを制作

- ▶ WFなら画面設計

- ▶ ウィン

そんな時代はもう過去のものです。

そう、○Phoneならね。



「宣言的UIフレームワーク」とは？

歴史的経緯 | Release Early, Release Often

- ・ モバイル端末の画面サイズは多様で統制できない
 - 画面設計の見直し頻度UP
- ・ 「個人が持ち運べるインターネット」としての巨大で新しいマーケット
 - スクラップ&ビルドやアジャイル開発の増加
 - 納品と保守 ► 継続的なアップデート

「宣言的UIフレームワーク」とは？

歴史的経緯 | Release Early, Release Often

- モバイルアプリ
- 異なる要件で変動するUIと状態を分割して管理したい



命令的から宣言的なUIへ

- 「個人が持つさまざまなデバイス」を一つのUIに入力して新しいデバイスで表示する

- スクリーンやUIの状態を自分で管理したくない



ライブラリからフレームワークへ

- 網

「宣言的UIフレームワーク」とは？

ソリューション | UIと状態を分離し、共通の仕掛けで結合させる

- UIと状態の分離
 - 仮想DOM（詳細は後ほど）
 - ▶ 今回Pythonで実装した部分
- UIと状態の結合
 - データバインディング（ReactiveXやPub/Subなど）
 - ▶ 余力がなく実装できていない🌀

GUIフレームワークの整理

Pythonで使えるGUIフレームワーク

※フレームワーク/ライブラリの区分はごちゃまぜ

tkinter --- Tcl/Tk の Python インタフェース

wxPython

Flexx

Edifice

Kivy

ENAML

PyQt / PySide

etc.

Pythonで使えるGUIフレームワーク

命令的UI

宣言的UI

tkinter --- Tcl/Tk の Python インタフェース

wxPython

Flexx

Edifice

Kivy

with .kv

ENAML

with QML

PyQt / PySide

統合型

分離型

etc.

Tkinter



命令的UI

若干くせがあるが自由度の高い標準ライブラリ

- 内部でTkのコマンドを呼び出すので非常に手続き的

```
> pack .widget -side left
```

- そのぶん書き方の自由度が高い
- Tcl/Tkの実行環境の準備やpython実行環境との相性などお手軽に見えて若干お手軽でない

```
class HelloApp(tk.Frame):
    def __init__(self):
        self.master = tk.Tk()
        super().__init__(self.master, width=300, height=300)
        self.master.title("Hello Tkinter")

        # 双方向データバインディング
        self.m_text = tk.StringVar()

        # ラベル
        self.label = tk.Label(
            self, textvariable=self.m_text
        ).pack(side="top")

        # 1行テキストボックス
        self.textbox = tk.Entry(
            self, textvariable=self.m_text
        ).pack(side="left")

        # ボタン
        self.button = tk.Button(
            self, text="Clear", command=lambda: self.m_text.set("")
        ).pack(side="left")

        self.pack()
```

wxPython (Phoenix)



命令的UI

Windowsライク？な命令的UIライブラリ

- Tkinterに似た手続き的記述
- WindowsのGUIツールキットに近い文法らしい
 - 個人的にはメソッドがPascalCaseなのが好みではない
 - 私は🍏派なのでよく分かりません
- 書き方を間違えると容易にMEMORY_BAD_ACCESSになる

```
class HelloFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, 'Hello wxPython', size=(300, 300))
        pnl = wx.Panel(self)

        # イベント駆動によるデータ更新
        self.m_text = ""

        # ラベル
        self.st = wx.StaticText(pnl, label=self.m_text)

        # 1行テキストボックス
        self.tc = wx.TextCtrl(pnl)
        self.tc.Bind(wx.EVT_TEXT, self.on_type)

        # ボタン
        self.bt = wx.Button(pnl, wx.ID_CLEAR, label="Clear")
        self.bt.Bind(wx.EVT_BUTTON, self.on_clear)

        sizer = wx.GridBagSizer()
        sizer.Add(self.st, (0, 0), (1, 3), flag=wx.EXPAND)
        sizer.Add(self.tc, (1, 0), (1, 2), flag=wx.EXPAND)
        sizer.Add(self.bt, (1, 2), (1, 1), flag=wx.EXPAND)
        sizer.AddGrowCol(1)
        pnl.SetSizer(sizer)
```

PyQt / Qt for Python (旧PySide)

命令的UI

ライセンスの異なるQtのPythonインタフェース



- 強力なSignal / Slot
 - イベントの発火とロジックをスレッドを超えて分離できる
- ややこしい選択肢とライセンス
 - PyQt:
GPLv3 / 商用
 - Qt for Python:
GPLv3 / LGPLv3 / 商用

```
class HelloWorldWidget(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        # スロットによるイベント駆動
        self.m_text = ""

        self.label = QLabel(self.m_text)
        self.label.setAlignment(Qt.AlignCenter)

        self.text_entry = QLineEdit()
        self.text_entry.textChanged.connect(self.on_type)

        self.button = QPushButton("Clear")
        self.button.clicked.connect(self.on_clear)

        layout = QGridLayout()
        layout.addWidget(self.label, 0, 0)
        layout.addWidget(self.text_entry, 1, 0)
        layout.addWidget(self.button, 1, 1)
        self.setLayout(layout)

    @Slot()
    def on_type(self):
        self.label.text = self.text_entry.text

    @Slot()
    def on_clear(self):
        self.label.text = ""
        self.text_entry.text = ""
```

PyQt / PySide with QML

宣言的UI - 分離型

Qtを宣言的に扱う

- QMLというマークアップを用いてビュー構造を宣言的に記述できる
 - 簡単なロジックならQML内でJavaScriptを書けてしまう

```
# --- Python ---
if __name__ == "__main__":
    app = QApplication(sys.argv)
    engine = QQmlApplicationEngine()
    url = QUrl.fromLocalFile("view.qml")
    engine.load(url)
    if not engine.rootObjects():
        sys.exit(-1)

    sys.exit(app.exec())
```

```
# --- QML ---
import QtQuick
import QtQuick.Controls
import QtQuick.Layouts

ApplicationWindow {
    title: qsTr("Hello QML")
    width: 300
    height: 300
    visible: true

    ColumnLayout {
        Text {
            id: text
            text: ""
        }

        RowLayout {
            Layout.alignment: Qt.AlignHCenter | Qt.AlignBottom
            TextField {
                id: textfield
                text: ""
                Layout.alignment: Qt.AlignHCenter | Qt.AlignTop
                Layout.margins: 5
                onTextChanged: {
                    text.text = textfield.text
                }
            }

            Button {
                text: qsTr("Clear")
                onClicked: {
                    text.text = ""
                }
            }
        }
    }
}
```



モバイルにも対応するフレキシブルなフレームワーク

- 扱いやすいMITライセンス
- クラスとbuildメソッドの戻り値でビューを組み立てる

```
# --- 命令的UI ---  
from kivy.app import App  
from kivy.uix.button import Button  
  
class TestApp(App):  
    def build(self):  
        return Button(text='Hello World')  
  
TestApp().run()
```


Kivy + Kv language



宣言的UI - 分離型

モバイルにも対応するフレキシブルなフレームワーク

- ロジックは基本的にPython側に書く
 - presentationとlogicの分離
- UIと状態はクラス名で結合

```
# --- 宣言的UI logic ---
class Controller(FloatLayout):
    label_wid = ObjectProperty()
    info = StringProperty()

    def do_action(self):
        self.label_wid.text = 'My label after button press'
        self.info = 'New info text'

class ControllerApp(App):
    def build(self):
        return Controller(info='Hello world')

if __name__ == '__main__':
    ControllerApp().run()
```

```
<Controller>:
    label_wid: my_custom_label

    BoxLayout:
        orientation: 'vertical'
        padding: 20

        Button:
            text: 'My controller info is: ' + root.info
            on_press: root.do_action()

        Label:
            id: my_custom_label
            text: 'My label before button press'
```



QtなUIをPythonicな構文で記述できるフレームワーク

- Pythonに似た構文でUIを記述
 - 動的にUIと状態を結合できる
- Pythonicな構文であるだけに
enamlファイルでなく
pyファイルに書きたくなる

```
# person_view.enaml
from enaml.widgets.api import (
    Window, Form, Label, Field
)

enamldef PersonView(Window):
    attr person
    title = 'Person View'
    Form:
        Label:
            text = 'First Name'
        Field:
            text := person.first_name
        Label:
            text = 'Last Name'
        Field:
            text := person.last_name
```



QtなUIをReactっぽく記述できるフレームワーク

- ReactをPythonにポートするとこんな感じ、を実現している
 - 引数と戻り値で木構造を形成
- 子要素を引数で渡すため、子要素の変更のためには関数チックな条件分岐を書く
 - 内包表記, (v1) if (c) else (v2)

```
import edifice as ed
from edifice import Label, TextInput, View

class MyApp(ed.Component):
    def render(self):
        return View(layout="row")(
            Label("Measurement in
meters:"),
            TextInput(""),
            Label("Measurement in feet:"),
        )

if __name__ == "__main__":
    ed.App(MyApp()).start()
```




Pythonコード上でWebベースのUIを記述できるフレームワーク

- withステートメントを活用したビューのビルド
 - 自作後に既存のものを調べたらめちゃくちゃ似ていた😅
- Webベースなので動かす選択肢が多い
- Webベースなのでサーバ (Python) 側とクライアント (JavaScript) 側をそれぞれ考える必要あり

```
from flexx import flx

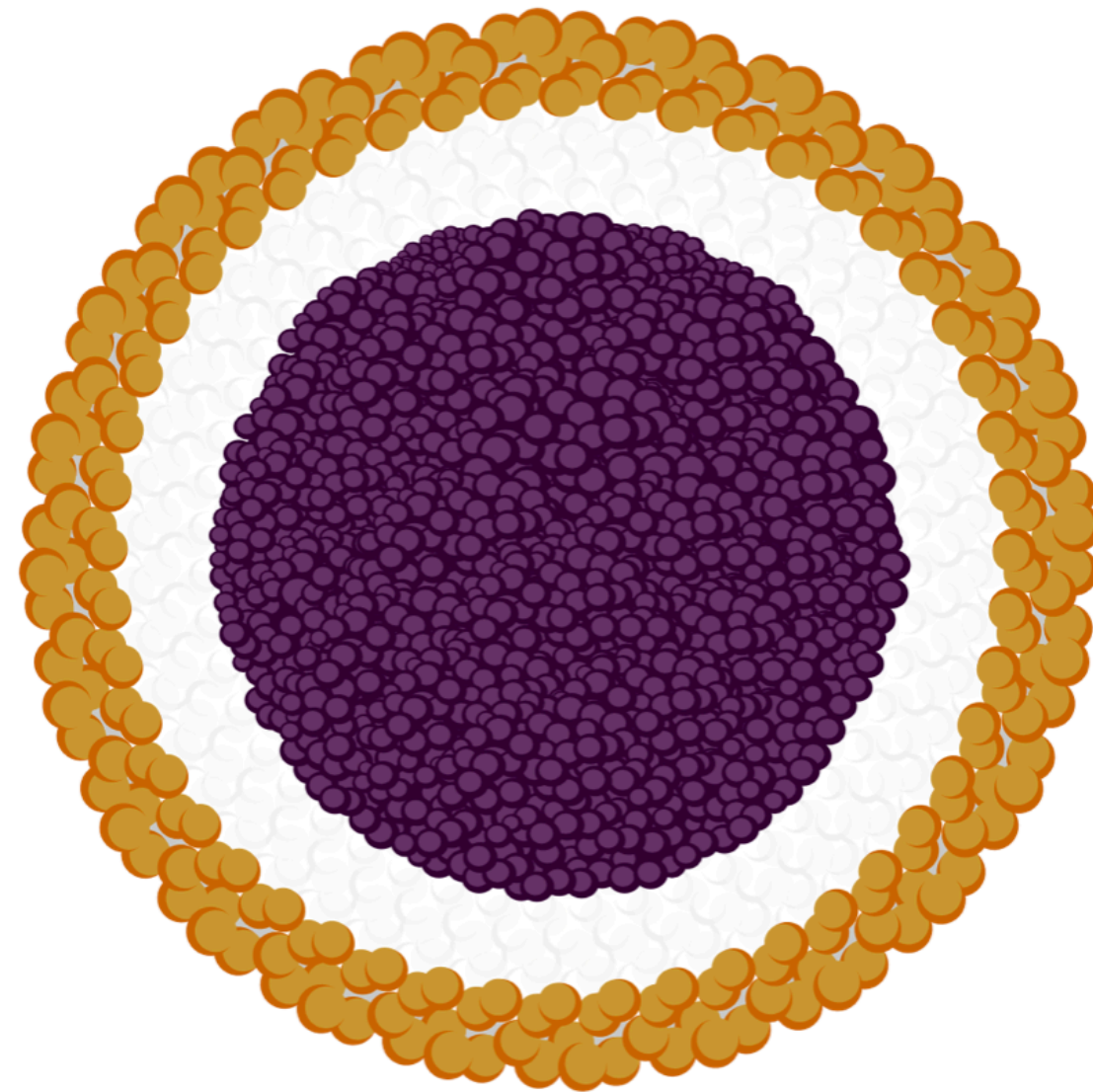
class Example(flz.Widget):
    def init(self):
        with flx.HSplit():
            flx.Button(text='foo')
            with flx.VBox():
                flx.Widget(style='background:red;', flex=1)
                flx.Widget(style='background:blue;', flex=1)

if __name__ == "__main__":
    app = flx.App(Example)
    app.launch('app')
    flx.run()
```

https://flexx.readthedocs.io/en/stable/guide/widget_basics.html

<https://flexx.readthedocs.io/en/stable/guide/running.html>

ようやく本題



Declarative UI Wrapper Framework for Python

The logo is inspired by Chinese sweets Jin Deui, which is sesame-coated and fried rice cake wrapping lotus, black bean, or red bean paste.

Register your account

You have to make an account to use this service. Please enter the form below and submit to register your account.

Input your name here

First Name

Last Name

DeUI

Home

Tutorial

Docs

Search

Search

...

with Body(clazz=["d-flex", ...]):

with Header(clazz="mb-auto"):

NavigationBar() # Component

with Main(clazz=["container", ...]):

with Division(clazz=["row", ...]):

with Division(clazz=["col-sm-4"]):

Image(clazz=["img-fluid"], src="/static/DeUI_logo.png")

with Division(clazz=["col-sm-6"]):

with Paragraph(clazz="h3"):

Text(value="Declarative UI Wrapper Framework for Python")

with Paragraph():

with Small(clazz="text-muted"):

with Joined():

Text(value="The logo is inspired by ...")

with Division(clazz=["row", "align-items-center"]):

with Heading(level=1):

Text(value="Register your account")

with Paragraph():

Text(value="You have to make an account ...")

Text(value="Please enter the form below ...")

...

Register

Register your account

You have to make an account to use this service. Please enter the form below and submit to register your account.

Input your name

First Name

Last Name

Pythonによる仮想DOMの実装

仮想DOM

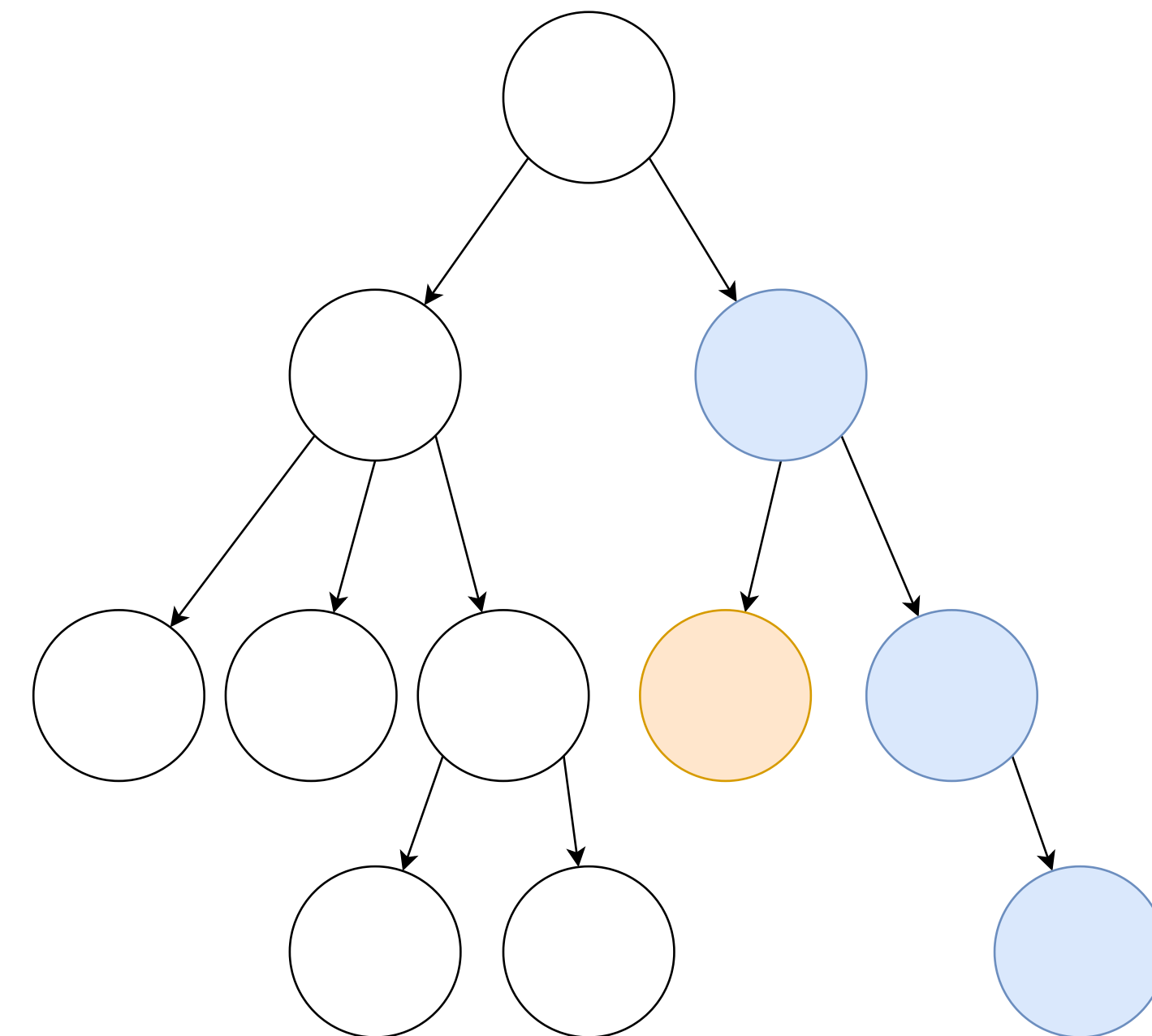
「仮想」的な「Document Object Model」

- 「Document Object Model」
 - タグ付けされた要素を木構造でモデル化すること/されたもの
- 「仮想」
 - 実際のDOM（≒画面）とは独立している、という意味
 - ▶ SortedやComponentなどの論理的要素を含められる

なぜ仮想DOMを用いるか

A. 問題の単純化と動作の高速化

- 仮想DOMは木構造
 - DOMの操作を一般のグラフ理論として解釈できる
 - 差分だけを検出できれば差分のある最上位以下のノードだけ更新できる
- 仮想DOMは実際のDOMとは独立
 - 軽量のノードだけを用いてDOMを動的に更新できる



そんなこと言われても
実際どう実装するんだ！

木の差分検出は
 $O(n^3)$ かかるぞ！



大事なことはすべて
Reactが教えてくれる

Reactの差分検出原理

<https://ja.reactjs.org/docs/reconciliation.html>



- ほぼ線形時間で近似解を求めることができる
 - 親同士が異なる要素ならそれ以降をすべてリビルドする
 - 同じ型の要素なら差分の属性のみを更新する
 - 比較する要素のペアは一意的なキーで揃える

これをPythonコードで実装💪

差分検出原理のコード

やってることは非常にシンプル



- ・ ノードが存在しないなら
新たにUI部品を生成する
- ・ ノードが存在するなら
生成済みのUI部品を受け渡す
- ・ ノード自身の状態変化は
ハッシュ値を求めて追跡
- ・ 再帰で渡すノード対をidでソート

```
@classmethod
def update_tree(cls, old_t, new_t, root=Root):
    if new_t is None:
        return
    if (old_t is None
        or new_t.w_type is not old_t.w_type):
        # building new tree
        new_t.build(root=root)
        return
    # copy widget from old v-DOM-node to new one
    new_t.widget = old_t.widget
    new_t.widget.owner = weakref.ref(new_t)
    if new_t.hashcode != old_t.hashcode:
        # update widget parameters
        new_t.widget.update(*new_t.args, **new_t.kwargs)
        new_t.need_update = True
    # continue comparison order by id
    for old_st, new_st in align(
        old_t.children, new_t.children, key='id'):
        App.update_tree(old_st, new_st, root=root)
```

magic: 😎|💣 = with + __new__

withステートメントをクラスで用いる弊害

What I initialize is not what we really want to initialize

- withステートメントにクラスのオブジェクトを渡すコード

👍 asにわたす値は
__enter__で自由に決定できる

👎 withに渡す時点で
Contextクラスの__init__が
走ってしまう

```
class Context:
    def __init__(self):
        # some great initializations...
        pass

    def __enter__(self):
        # push context
        return some_obj # for `as`

    def __exit__(self, t, v, trace):
        # pop context
        pass

... # In use

with Context() as context:
    pass
```

withステートメントをクラスで用いる弊害

What I initialize is not what we really want to initialize

withに渡す時点で初期化したいのは仮想DOM (View)



プログラマがwithに渡したいのは実際のDOM (Widget)

__new__メソッドによる初期化時の挙動のオーバーライド

__new__ とは

Pythonのインスタンス生成=🛵

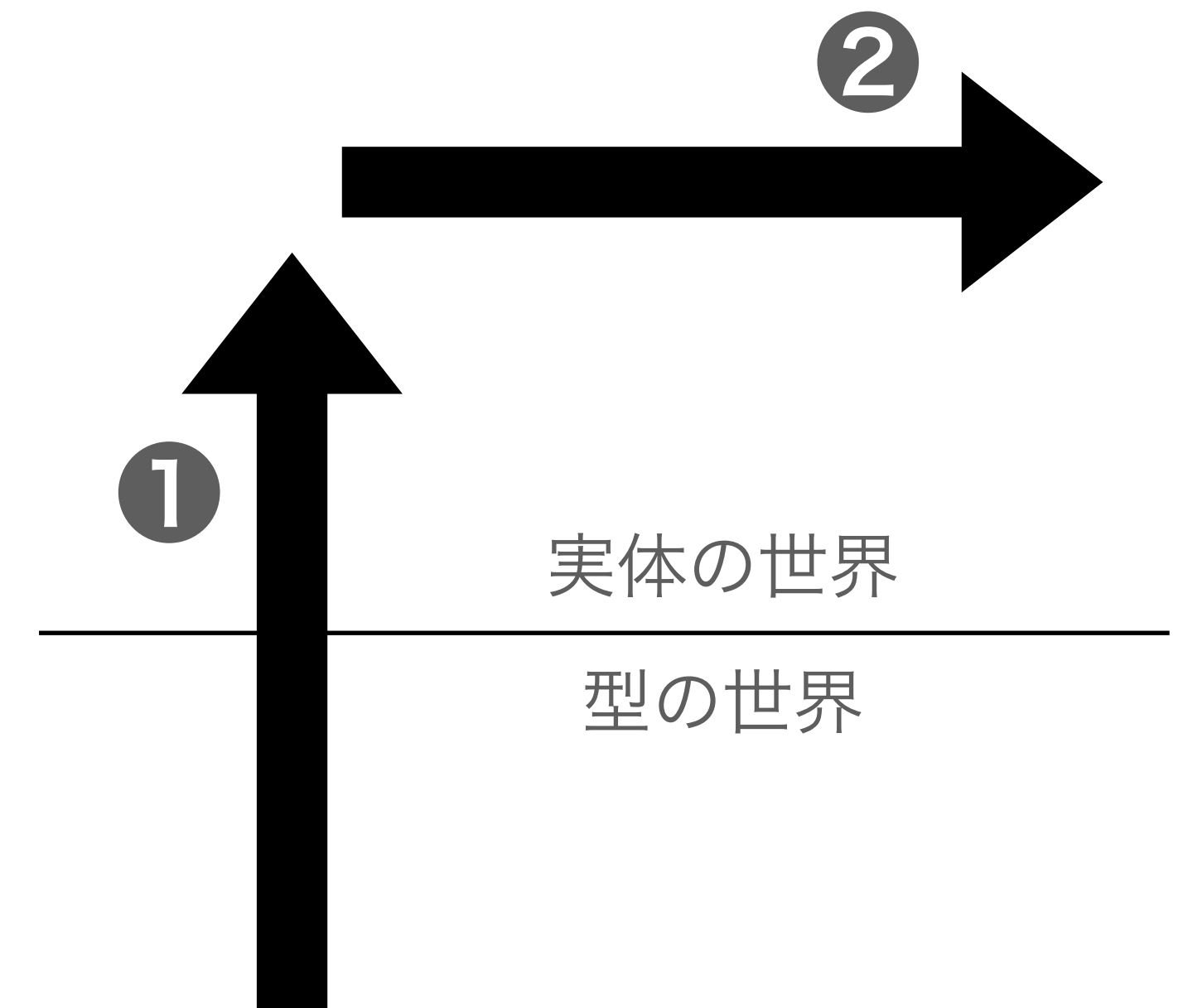
- Pythonはインスタンス生成時に2つのメソッドを呼び出す

1. `__new__(cls, ...)`

- クラスのインスタンス生成のために呼ばれる

2. `__init__(self, ...)`

- インスタンスの初期化のために呼ばれる



with + __new__

インスタンス化とコンテキストの束縛とでクラスを分離



1. 実際のDOMに対応するWidgetをインスタンス化しようとする
と仮想DOMのViewが同時に
インスタンス化され、
2. Widgetのインスタンスは
遅延評価で取得できる

```
def __new__(cls, *args, **kwargs):  
    view = View(cls, *args, **kwargs)  
    widget = super().__new__(cls)  
    widget.update(*args, **kwargs)  
  
    def get_initial_widget():  
        widget.owner = weakref.ref(view)  
        widget.update(*args, **kwargs)  
        return widget  
  
    view.get_initial_widget \  
        = get_initial_widget  
    return view
```


__new__を併用するメリット

アスペクト指向なコンテキスト束縛

- コンテキストの束縛を継承なしに別のクラスに移譲できる
 - > アスペクト指向な with によるコンテキスト束縛の実現
- Widgetクラスには__enter__や__exit__を書いていない
- __enter__と__exit__をラップするMapperを用意すれば複数のコンテキストを適用できる
 - ▶ <https://gist.github.com/urushiyama/c950f641e884c29ae9866dbadbf3dcc6>



__人人人人人人人人__
> 束縛された余白 <
~Y^Y^Y^Y^Y^Y^Y^Y^Y~

__new__を用いるデメリット

光？あるところに影あり

- 一般的なインスタンス化と挙動が異なる
 - バグフィックスの難易度上昇
- __init__を併用するとさらにカオス
 - 安易に__new__内でClass(...)を呼び出すと無限ループに陥る

最後に




最後に

これからの宣言的UIとフロントエンド

- 宣言的UIはデータセットに対する冪等性があるため
以下の技術と非常に相性が良い
 - 宣言型プログラミング言語（Elixir, Haskell, 先行事例的にはElmなど）
 - 直列化可能な構造体
 - アスペクト指向とDI（先行事例的にはVue Composition APIなど）
 - 宣言的で状態同期しやすいAPI（GraphQLなど）

最後に

これからの宣言的UIとフロントエンドとPython

- 宣言的UIはデータセットに対する冪等性があるため
以下の技術と非常に相性が良い
 - 宣言型プログラミング言語  引数のスコープで無名関数を定義できれば…
 - 直列化可能な構造体  @dataclass, pydantic
 - アスペクト指向とDI  typing.Protocol, monkeypatch
 - 宣言的で状態同期しやすいAPI  豊富なGraphQLライブラリ

最後に

これからの宣言的UIとフロントエンドとPython

- ・ 宣言的UIはデータセットに対する冪等性があるため
以下の技術と非常に相性が良い

宣言的プログラミング言語

👿 引数のスコープで無名関数を定義できれば...

👿👿👿👿👿 Pythonで汎用的なGUIを組むニーズ

- アスペクト指向とDI
- 宣言的で状態同期しやすいAPI

😊 typing.Protocol, monkeypatch

😊 豊富なGraphQLライブラリ

ご清聴ありがとうございました

補足：with の選定理由

文字列/引数と戻り値を用いるパターンと比較したメリット

- builtinのstatementが使える
 - if文やfor文が使える
 - Python 3.10ならmatchも使える
- 構造が複雑化しても丸括弧を多用せずに済む

```
with NewsColumn():  
    for news_item in news_list:  
        match news_item:  
            case [title]:  
                Bulletin(title)  
            case [title, summary]:  
                Headline(title, summary)  
            case _:  
                raise ValueError(  
                    "Unknown kind of news_item")
```

補足：なぜ「ラッパー」？

A. 「ラッパー」にしようとしていたから

- ・ 仮想DOMに基づいてWidgetを更新するラッパーコードを書けば理論上はTkinterでもKivyでも同様の書き方でラップできる
 - HTMLは文字列要素のラッパー
 - Kivyでもやろうとしていたが
(本業が楽しく忙しくなり) 未着手のまま

補足：データバインディングの展望

GraphQLを用いた仮想ステートを導入したい

