Лабораторная работа
Знакомство с фреймворком Kivy. Разработка программ, взаимодействующих с виджетами
Работа с разными видами разметки (BoxLayout, FloatLayout, GridLayout)

Цель работы: сформировать умения разрабатывать apk, iOS, exe-приложений с помощью фреймворка Kivy.

Задание:
1. Изучить теоретические сведения.
2. Выполнить задания в соответствии с вариантом.

Теоретические сведения

Установка Kivy:

```
pip install kivy
pip install kivy[base] kivy_examples --pre
--extra-index-url https://kivy.org/downloads/simple/
```

Следует обратить внимание, что некоторые из требуемых зависимостей могут быть не включены в устанавливаемый пакет (это зависит от типа и версии операционной системы вашего компьютера и от версии Python). Если возникнут проблемы при запуске написанных программных модулей, то можно использовать следующие дополнительные команды для установки необходимых отсутствующих библиотек, чтобы исправить возникающие ошибки:

```
pip install kivy-deps.angle
pip install kivy-deps.glew
pip install kivy-deps.gstreamer
pip install kivy-deps.sdl2
```

Структура приложений kivy имеет вид:

```
from kivy.app import App

class TutorialApp(App):
    def build(self):
        return
TutorialApp().run()
```

Виджеты Kivy:

- `Widget` — базовый класс, необходимый для создания виджетов.
- `Label` используется для рендеринга текста в окнах и диалоговых окнах.
- `TextInput` предоставляет поле для редактируемого обычного текста.
- `Button` запускает действия, когда пользователь нажимает на него.
- `CheckBox` предоставляет кнопку с двумя состояниями, которую можно либо установить, либо снять.
- `Image` используется для отображения изображения в ваших графических интерфейсах.
- `ProgressBar` визуализирует ход выполнения некоторых задач.
- `DropDown` предоставляет универсальный раскрывающийся список, в котором могут быть перечислены различные виджеты.

Некоторые примеры распространенных макетов:

- `BoxLayout` упорядочивает виджеты последовательно по вертикали или по горизонтали.
- `FloatLayout` размещает виджеты в определенной позиции в содержащем окне.

- `GridLayout` упорядочивает виджеты в сетке, определяемой строками и столбцами.
- `PageLayout` создает многостраничные макеты таким образом, чтобы можно было переключаться с одной страницы на другую.

Пример 1: возврат одного виджета (простого приложения Hello World)

```
from kivy.app import App
from kivy.uix.button import Button
class TutorialApp(App):
    def build(self):
        return Button(text="Hello World!")
TutorialApp().run()
```

Пример 2: возврат нескольких виджетов + кнопка печатает текст метки

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.uix.button import Button

class TutorialApp(App):
    def build(self):
        mylayout = BoxLayout(orientation="vertical")
        mylabel = Label(text= "My App")
        mybutton =Button(text="Click me!")
        mylayout.add_widget(mylabel)
        mybutton.bind(on_press= lambda a:print(mylabel.text))
        mylayout.add_widget(mybutton)
        return mylayout
TutorialApp().run()
```

Пример 3: использование класса (одиночный виджет) + кнопка печатает «Моя кнопка»

```
from kivy.app import App
from kivy.uix.button import Button

class Mybutton(Button):
    text="Click me!"
    on_press =lambda a : print("My Button")

class TutorialApp(App):
    def build(self):
        return Mybutton()
TutorialApp().run()
```

Пример 4: это то же самое, что и ex. 2, но он показывает, как использовать класс

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
```

```
from kivy.uix.label import Label
from kivy.uix.button import Button

class MyLayout(BoxLayout):
    #You don't need to understand these 2 lines to make it work!
    def __init__(self, **kwargs):
        super(MyLayout, self).__init__(**kwargs)

        self.orientation="vertical"
        mylabel = Label(text= "My App")
        self.add_widget(mylabel)
        mybutton =Button(text="Click me!")
        mybutton.bind(on_press= lambda a:print(mylabel.text))
        self.add_widget(mybutton)

class TutorialApp(App):
    def build(self):
        return MyLayout()
TutorialApp().run()
```

При использовании фрейморка Kivy программный код для создания элементов пользовательского интерфейса можно писать на Python, а можно для этих целей использовать специальный язык. Язык KV позволяет достаточно быстро и просто создавать прототипы программ и гибко вносить изменения в пользовательский интерфейс. Это также обеспечивает при программировании отделение логики приложения от пользовательского интерфейса.

Как связать файлы kv и kivy

Есть два способа загрузить программный код на KV в приложение:

– По соглашению об именах. В этом случае Kivy ищет файл с расширением .kv и с тем же именем, что и имя базового класса приложения в нижнем регистре, за вычетом символов «App». Например, если базовый класс приложения имеет имя MainApp, то для размещения кода на языке KV нужно использовать файл с именем main. kv. Если в этом файле задан корневой виджет, то он будет использоваться в качестве основы для построения дерева виджетов приложения.

– С использованием специального модуля (компоненты) Builder можно подключить к приложению программный код на языке KV либо из строковой переменной, либо из файла с любым именем, имеющем расширение». kv». Если в данной строковой переменной или в этом файле задан корневой виджет, то он будет использоваться в качестве основы для построения дерева виджетов приложения.

Пример Builder с подключением строковой переменной:

```
from kivy. app import App
from kivy.lang import Builder
qqq ='''
BoxLayout:
    Button:
        text: "Кнопка 1"
'''
class MainApp (App):
    def build (self):
        return Builder. load_string (qqq)
MainApp().run ()
```

Пример использования пользовательского класса: создан пользовательский класс MyBox на основе базового класса BoxLayout. При этом нужно явно выполнить импорт базового класса BoxLayout

```
from kivy. app import App
from kivy.lang import Builder
from kivy.uix.boxlayout import BoxLayout
KV = '''
MyBox: # контейнер (пользовательский класс)
    Button:
        text: "Кнопка 2"
'''
# пользовательский класс MyBox на основе базового класса BoxLayout
class MyBox (BoxLayout):
    pass
class MainApp (App):
    def build (self):
        return Builder.load_string (KV)
MainApp().run ()
```

Пример: приложение «hello world» в kivy (с помощью kv)

```
from kivy.app import App
from kivy.uix.label import Label
from kivy.lang import Builder

Builder.load_string('''
<SimpleLabel>:
    text: 'Hello World'
''')

class SimpleLabel(Label):
    pass

class SampleApp(App):
    def build(self):
        return SimpleLabel()

if __name__ == "__main__":
    SampleApp().run()
```

Для создания пользовательского класса непосредственно в коде на языке KV используется следующая конструкция:
<Имя_пользовательского_класса@Имя_базового_класса>

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
# пользовательский класс MyBox на основе базового класса BoxLayout
<MyBox@BoxLayout>
MyBox:
    Button:
        text: "Кнопка 1"
'''
class MainApp (App):
    def build (self):
        return Builder. load_string (KV)
MainApp().run ()
```

Пользовательский класс в Kivy еще называют динамическим классом. Динамический класс создается на основе базового класса, при этом для него можно сразу определить

свой набор свойств. Например, в контейнере BoxLayout имеется три кнопки, для которых заданы идентичные свойства:

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
BoxLayout:
    Button:
        text: 'Кнопка 1'
        pos_hint: {'center_x':.5, 'center_y':.6}
        font_size: '25'

    Button:
        text: 'Кнопка 2'
        pos_hint: {'center_x':.5, 'center_y':.6}
        font_size: '25'

    Button:
        text: 'Кнопка 3'
        pos_hint: {'center_x':.5, 'center_y':.6}
        font_size: '25'

'''
class MainApp (App):
    def build (self):
        return Builder. load_string (KV)
MainApp().run ()
```

Для того чтобы не повторять многократно задание одних и тех же свойств каждому элементу, можно сформировать динамический класс и в нем один раз задать этот набор свойств:

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
<MyButton@Button>:
    font_size: '25'
    pos_hint: {'center_x':.5, 'center_y':.5}
BoxLayout:
    orientation: 'vertical'
    MyButton:
        text: 'Кнопка 1'
    MyButton:
        text: 'Кнопка 2'
    MyButton:
        text: 'Кнопка 3'
'''
class MainApp (App):
    def build (self):
        return Builder.load_string (KV)
MainApp().run ()
```

Создан динамический класс MyButton на основе базового класса Button. Для класса MyButton один раз заданы свойства. Затем в контейнер BoxLayout, помещаются три кнопки MyButton, для которых задается всего одно свойство – text. Все остальные свойства этих кнопок будут наследованы от динамического класса MyButton@Button. Таким образом, программный код упрощается и сокращается количество строк.

На Python импорт внешних модулей выглядит следующим образом:
import numpy as np
На языке KV:

#:import np numpy

В языке KV имеется три зарезервированных ключевых слова, обозначающих отношение последующего содержимого к тому или иному элементу приложения:

- app: (приложение) позволяет обратиться к элементам приложения (например, из кода на KV можно обратиться к функциям, которые находится в разделе приложения, написанного на Python);
- root: (корень) позволяет обратиться к корневому виджету;
- self: (сам) позволяет обратиться к себе, и получить от виджета (от себя) свои же параметры

Пример: получить состояние кнопки

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
Button
    text: 'Состояние кнопки – ' +self.state
'''
class MainApp (App):
    def build (self):
        return Builder. load_string (KV)
MainApp().run ()
```

Пример: кнопка сделала запрос к свойству корневого виджета

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
BoxLayout:
    orientation: 'vertical'
    Button:
        text: root.orientation
'''
class MainApp (App):
    def build (self):
        return Builder.load_string (KV)
MainApp().run ()
```

Пример: обратиться к элементам приложения

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
BoxLayout:
    orientation: 'vertical'
    Label:
        text: app.name
    Button:
        text: 'Кнопка 1'
        on_press: app.press_button (self. text)
'''
class MainApp (App):
    def build (self):
        return Builder. load_string (KV)
    def press_button (self, instance):
        print ('Вы нажали на кнопку!')
        print (instance)
MainApp().run ()
```

С использованием ключевого слова id можно любому виджету назначить уникальное имя (идентификатор). Это имя можно использовать для ссылок на виджет, то есть обратиться к нему в коде на языке KV.

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
BoxLayout:
    orientation: 'vertical'
    Button:
        id: bt1
        text: 'Кнопка 1'
        on_press: lb1.text = bt1.text+' была нажата'
    Button:
        id: bt2
        text: 'Кнопка 2'
        on_press: lb1.text = bt2.text+' была нажата'
    Label:
        id: lb1
        text: 'Метка'
        on_touch_down: self. text = 'Это метка'
'''
class MainApp (App):
    def build (self):
        return Builder.load_string (KV)
MainApp().run ()
```

С использованием ключевого слова ids можно из кода на Python обратиться к виджету, который создан в разделе программы в коде на KV.

Пример: текст «Нажата кнопка», сначала был передан из фрагмента кода на KV во фрагмент кода на Python, а затем возвращен обратно.

```
from kivy. app import App
from kivy.lang import Builder
from kivy.uix.boxlayout import BoxLayout
KV = '''
box:
    Button:
        text: 'Кнопка'
        on_press: root.result ('Нажата кнопка')
    Label:
        id: itog
'''
class box (BoxLayout):
    def result (self, entry_text):
        self.ids ['itog'].text = entry_text
class MainApp (App):
    def build (self):
        return Builder.load_string (KV)
MainApp().run ()
```

Приложение состоит из двух программных модулей: базовый модуль на языке Python, и модуль с деревом виджетов на языке KV.
В базовом модуле приложения на языке Python всегда создается базовый класс, при этом используется зарезервированный шаблон имени – Class_nameApp. Модуль с деревом виджетов на языке KV так же имеет зарезервированный шаблон имени – «class_name. kv».
В этом случае базовый класс Class_nameApp ищет». kv» – файл с тем же именем, что и имя базового класса, но в нижнем регистре и без символов APP. Например, если базовый класс приложения имеет имя – «My_ClassAPP», то файл с кодом на языке KV должен иметь имя «my_class. kv». Если такое совпадение имен обнаружено, то программный код, содержащийся в этих двух файлах, будет объединен в одно приложение.
Пример файл 1.py

```
from kivy. app import App
class Basic_Class (App): # определение базового класса
    pass
My_App = Basic_Class ()
My_App.run ()
```

файл basic_class.kv:

```
Label:
    text: 'Метка из файла basic_class.kv'
    font_size: '16pt'
```

Обработка событий виджетов

В Kivy реализовано два ***способа реагирования на события***:
– явное связывание визуального элемента с заданной функцией;
– неявное связывание визуального элемента с заданной функцией.
Пример1: явное связывание визуального элемента с функцией (код на языке Python)

```
from kivy. app import App
from kivy. uix. button import Button
class MainApp (App):
    def build (self):
        button = Button (text="Кнопка",
                         size_hint= (.5,.5),
                         pos_hint= {'center_x':.5, 'center_y':.5})
        button.bind(on_press=self.press_button)
        return button
    def press_button (self, instance):
        print ("Вы нажали на кнопку!")
MainApp().run ()
```

Пример 1.1: кнопка печатает текст метки (с помощью def в python с использованием ids)

В .py:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout

class MyLayout(BoxLayout):
    def printMe(self):
        print(self.ids.mylabel.text)
class TutorialApp(App):
    def build(self):
        self.load_kv('myapp.kv')
        return MyLayout()
TutorialApp().run()
```

В myapp.kv:

```
<MyLayout>
    orientation:"vertical"
    Label:
        id:mylabel
        text:"My App"
    Button:
        text: "Click me!"
        on_press: root.printMe()
```

Пример2: явное связывание визуального элемента с функцией (с использованием языка KV)

```
from kivy. app import App
from kivy.lang import Builder
KV = '''
Button:
    text: 'Кнопка'
    size_hint:.5,.5
    pos_hint: {'center_x':.5, 'center_y':.5}
    on_press: app.press_button (root)
'''
class MainApp (App):
    def build (self):
        return Builder. load_string (KV)
    def press_button(self, instance):
        print ('Вы нажали на кнопку!')
MainApp().run ()
```

Пример3: неявное связывание визуального элемента с функцией

```
# файл 111.py
from kivy. app import App
from kivy. uix. button import Button
class Basic_Class1 (App):
    def build (self):
        button = Button (text='Кнопка',
                         size_hint= (.5,.5),
                         pos_hint= {'center_x':.5, 'center_y':.5})
    return button
    def press_button (self):
        print ('Вы нажали на кнопку!')
My_App = Basic_Class1 ()
My_App.run ()

# файл basic_class1.kv
<Button>:
    on_press: app.press_button()
```

Пример 4: создается холст с 2 точками и 1 строка между ними. Вы сможете перемещать точку и линию вокруг.

```
from kivy.app import App
from kivy.graphics import Ellipse, Line
from kivy.uix.boxlayout import BoxLayout


class CustomLayout(BoxLayout):

    def __init__(self, **kwargs):
        super(CustomLayout, self).__init__(**kwargs)

        self.canvas_edge = {}
        self.canvas_nodes = {}
        self.nodesize = [25, 25]
```

```
        self.grabbed = {}

        #declare a canvas
        with self.canvas.after:
            pass

        self.define_nodes()
        self.canvas.add(self.canvas_nodes[0])
        self.canvas.add(self.canvas_nodes[1])
        self.define_edge()
        self.canvas.add(self.canvas_edge)

    def define_nodes(self):
        """define all the node canvas elements as a list"""

        self.canvas_nodes[0] = Ellipse(
            size = self.nodesize,
            pos =  [100,100]
            )

        self.canvas_nodes[1] = Ellipse(
            size = self.nodesize,
            pos =  [200,200]
            )

    def define_edge(self):
        """define an edge canvas elements"""


        self.canvas_edge = Line(
            points =  [
                self.canvas_nodes[0].pos[0] + self.nodesize[0] / 2,
                self.canvas_nodes[0].pos[1] + self.nodesize[1] / 2,
                self.canvas_nodes[1].pos[0] + self.nodesize[0] / 2,
                self.canvas_nodes[1].pos[1] + self.nodesize[1] / 2
                ],
            joint = 'round',
            cap = 'round',
            width = 3
            )

    def on_touch_down(self, touch):
        for key, value in self.canvas_nodes.items():
            if (value.pos[0] - self.nodesize[0]) <= touch.pos[0] <=
(value.pos[0] + self.nodesize[0]):
                if (value.pos[1] - self.nodesize[1]) <= touch.pos[1] <=
(value.pos[1] + self.nodesize[1]):
                    touch.grab(self)
                    self.grabbed = self.canvas_nodes[key]
                    return True

    def on_touch_move(self, touch):
        if touch.grab_current is self:
            self.grabbed.pos = [touch.pos[0] - self.nodesize[0] / 2,
touch.pos[1] - self.nodesize[1] / 2]
            self.canvas.clear()
            self.canvas.add(self.canvas_nodes[0])
            self.canvas.add(self.canvas_nodes[1])
            self.define_edge()
            self.canvas.add(self.canvas_edge)
        else:
            # it's a normal touch
            pass
```

```
    def on_touch_up(self, touch):
        if touch.grab_current is self:
            # I receive my grabbed touch, I must ungrab it!
            touch.ungrab(self)
        else:
            # it's a normal touch
            pass

class MainApp(App):

    def build(self):
        root = CustomLayout()
        return root

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

Пример 5: простые всплывающие окна с Kivy.

```
from kivy.app import App
from kivy.uix.popup import Popup
from kivy.lang import Builder
from kivy.uix.button import Button

Builder.load_string('''
<SimpleButton>:
    on_press: self.fire_popup()
<SimplePopup>:
    id:pop
    size_hint: .4, .4
    auto_dismiss: False
    title: 'Hello world!!'
    Button:
        text: 'Click here to dismiss'
        on_press: pop.dismiss()
''')

class SimplePopup(Popup):
    pass

class SimpleButton(Button):
    text = "Fire Popup !"
    def fire_popup(self):
        pops=SimplePopup()
        pops.open()

class SampleApp(App):
    def build(self):
        return SimpleButton()

SampleApp().run()
```

Задания

Задание 1

Создать приложение, которое содержит следующие виджеты:

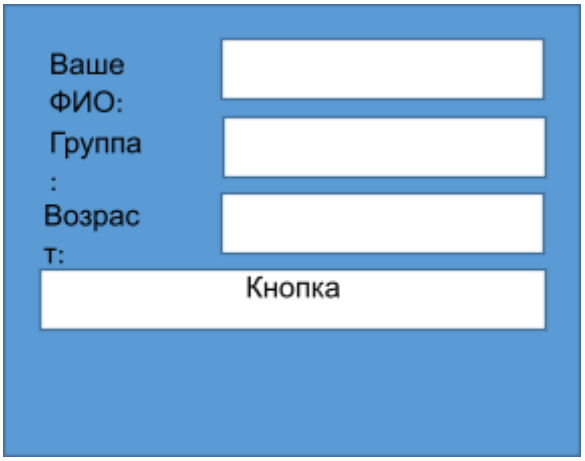
Label	Надпись «Это метка», размер шрифта – 20, синий
Button	Надпись «Это кнопка», размер – 10% от окна
CheckBox	Цвет флажка – красный, флажок поставлен
Image	Любое изображение
Slider	Вертикальное положение, мин.значение – 1, макс.значение – 100, начальное значение – 50, цвет следа бегунка - зеленый
ProgressBar	Макс.значение – 50, текущее значение - 20
TexInput	Размер шрифта – 30, красный, скрывать вводимые символы (заменяя их на *)
Swich	состояние -включено
Video	воспроизвести любое видео при необходимости установки кодеков: pip install ffpyplayer

Задание 2

Вариант 1

Создать оконное приложение, содержащие следующие виджеты:

При нажатии на кнопку появляется текст «ФИО из группы … - молодец! А ему всего лишь … лет»



Вариант 2

Создать оконное приложение, содержащие следующие виджеты:

При нажатии на кнопку идет загрузка Progressbar и появляется текст «Уважаемый ФИО, идет обработка ваших данных»

Задание 3

Создать оконное приложение, изменив задание из ЛР «Функции и модули».

Задание 4

Разработать калькулятор, выполняющий сложение, вычитание, умножение и деление чисел.

Контрольные вопросы:

1. Каково назначение виджетов? Перечислите основные виджеты Kivy.
2. Перечислите основные преимущества использования языка KV.
3. Приведите алгоритм работы программы, управляемой событиями.
4. Как связать файлы kv и kivy?
5. Каково назначение layouts Kivy: BoxLayout; FloatLayout, GridLayout?