Анатолий Постолит

Разработка кроссплатформенных мобильных и настольных приложений на Python

Практическое пособие

Анатолий Постолит

Разработка кроссплатформенных мобильных и настольных приложений на Python. Практическое пособие

«Издательские решения»

Постолит А.

Разработка кроссплатформенных мобильных и настольных приложений на Python. Практическое пособие / А. Постолит — «Издательские решения»,

ISBN 978-5-00-561871-9

Книга посвящена вопросам использования интерпретатора Python, фрейморка Kivy и библиотеки KivyMD для создания приложений, которые способны работать на любом устройстве (настольный компьютер, планшет, смартфон, мини-компьютер) и в любой операционной системе (Windows, Linux, MacOS, Android, iOS). Эти приложения адаптированы к работе на устройствах с сенсорным экраном, кроме того, они позволяют и обычный монитор настольного компьютера или ноутбука превратить в сенсорный экран.

ISBN 978-5-00-561871-9

Содержание

Разработка кроссплатформенных мобильных и настольных приложений на Python Практическое пособие

Анатолий Постолит

ISBN 978-5-0056-1871-9
Создано в интеллектуальной издательской системе Ridero

Введение

В последние годы кроссплатформенная технология разработки мобильных и настольных приложений становится все более популярной [44]. Кроссплатформенный подход позволяет создавать приложения для различных платформ с одной кодовой базой, что экономит время и деньги, и избавляет разработчиков от ненужных усилий.

Согласно исследованию Digital 2020 Reports [11], подготовленному компаниями We Are Social Inc. и Hootsuite Inc., число пользователей интернета по всему миру увеличивается на 9 человек в секунду. Это означает, что каждый день к мировому онлайн-сообществу присоединяется более 800 тысяч человек, которые пользуются как настольными, так и мобильными устройствами. Интересно, что мобильные приложения становятся все более популярными.

Проникновение смартфонов в повседневную жизнь растет во всем мире. Ожидается, что к 2024 году три из четырех используемых телефонов будут смартфонами. Согласно статистике StatCounter [12], доля пользователей настольных устройств снизилась до 45,66%. Это объясняется изменением нашего образа жизни. Мы проводим в интернете больше времени, чем когда-либо прежде. Почти каждый имеет доступ к смартфону или планшету. Учитывая то, что среднестатистический пользователь в среднем проводит в сети почти 7 часов в день, неудивительно, что более половины этого трафика поступает с мобильных устройств. Это, в свою очередь, способствует росту рынка мобильных приложений, что подтверждается статистикой. Согласно отчету Statista [33], за 2019 год, мировые доходы от мобильных приложений составили 461 млрд. долл., а к 2023 году платные загрузки и реклама в приложениях, как предполагается, принесут более 935 млрд. долл. дохода.

Сегодня на рынке мобильных платформ два крупных игрока – Android и iOS, которые вместе составляют около 99% от общей доли рынка мобильных операционных систем. Согласно различным статистическим данным, Android выигрывает по количеству пользователей, но нет недостатка и в сторонниках iOS, доля которого на рынке составляет 25,75%. В то время как Google Play Store может похвастаться большим количеством приложений (2,5 млн.), Apple App Store содержит более 1.8 млн. приложений. Одного этого факта достаточно, чтобы показать, что ни одну из двух платформ не следует упускать из виду.

Нативное решение, как следует из названия, предполагает разработку приложения на родном для данной платформы языке программирования: Java или Kotlin для Android, Objective-C или Swift для iOS. Будучи глубоко ориентированной на операционную систему, разработка нативных приложений имеет свои достоинства и недостатки. С одной стороны, нативное решение обеспечивает доступ ко всем функциям данной ОС. С другой стороны, если вы хотите охватить оба типа пользователей, вам придется создать два отдельных приложения, что потребует в два раза больше времени, денег и усилий.

Что касается настольных компьютеров, то здесь еще сложнее. Есть два компьютерных гиганта. Это Microsift с компьютерами под ОС Windows, и Apple с компьютерами под управлением операционных систем семейства Mac OS. А еще есть и народное творение Linux. Торговая марка «Linux» принадлежит создателю и основному разработчику ядра Линусу Торвальдсу. При этом проект Linux, в широком смысле, не принадлежит какой-либо организации или частному лицу. Вклад в его развитие и распространение осуществляют тысячи независимых разработчиков и компаний, взаимодействие между которыми осуществляется группами пользователей Linux. То есть, для того, чтобы приложением мог воспользоваться широкий круг пользователей, программный код необходимо писать в трех вариантах: под Windows, под Mac OS и под Linux. При этом для каждого варианта нужно использовать свой язык программирования, адаптированный для конкретной операционной системы.

Здесь на помощь приходит технология кроссплатформенного программирования. Кроссплатформенная мобильная разработка позволяет, как правило, охватить две операционные системы, iOS и Android, одним кодом. Для этих целей используются такие инструменты, как React Native, Flutter, Ionic, Xamarin, PhoneGap. Кроссплатформенная разработка настольных приложений обеспечивает создание программ, способных работать под Windows, MacOS и Linux. Для этих целей используются такие инструменты, как Electron JS, Qt, GTK, Avalonia, Tkinter. То есть упомянутый выше инструментарий используется для кроссплатформенной разработки либо мобильных, либо настольных приложений. А есть ли такое универсальное инструментальное средство, которое обеспечивает работу программы из одного кода и на персональных компьютерах, и на мобильных устройствах, и под любой операционной системой? А ответ на этот вопрос лежит в почти библейской истории, вот она.

Почти библейская история.

Обосновались на горе Олимп три компьютерных бога, и звались они: Google, Apple и Microsoft. У каждого бога был свой Эдемский сад, в котором обитали их сыновья: у Google сын Android, у Apple братья Ios и Macos, у Micrisoft сын Windows. И могли сыны божьи гулять, каждый по своему саду, и рвать плоды любые, и торговать ими. И росло в тех садах дерево познания. Но заповедал каждый бог своему сыну: от всякого дерева в саду ты будешь рвать, а от дерева познания, не рви и не ешь с него, ибо в день, в который вкусишь с него, или сорвешь и продашь с него, станешь ты мне не угодным.

А у подножья горы, в долине обитал простой люд, и у каждого простолюдина было свое имя, но боги обращались к ним по общему прозвищу – Linux. И не имели право простолюдины заходить в сады и плоды вкушать. А могли они только покупать плоды всякие из садов божьих, кроме плодов с дерева познания.

А на дереве познания сидел хитрый змий. И видел он, несправедливость, как боги наживались на простолюдинах, продавая им плоды садов своих. Однажды набрал он плодов с дерева познания и раздал их простолюдинам. И вкусивши плодов от древа познания, прозрели простолюдины, и поняли, что могут сотворить свои сады. И стали они сами сады возделывать, и плоды растить, и угощать друг друга плодами своими, и дарить, и менять, и торговать ими. И перестали они зависеть от прихотей компьютерных богов. Имя того змия был ***Python****, а имя плода запретного* ***Kivy****.*

Kivy – это фреймворк, созданный в 2011 году, с тех пор успешно развивается и распространяется бесплатно. Это среда программирования на Python, с открытым исходным кодом. В настоящее время возможности фреймворка Kivy значительно расширены за счет библиотеки KivyMD. Здесь аббревиатура MD означает Material Design. Material Design – это некий стандарт, созданный Google, которому нужно придерживаться при разработке приложений для Android и iOS.

Фреймворк Kivy с библиотекой KivyMD позволяет создавать кроссплатформенные приложения, способные работать на любом устройстве (настольный компьютер, планшет, смартфон, мини-компьютер) и в любой операционной системе (Windows, Linux, MacOS, Android, iOS, Raspberry Pi). Приложения адаптированы к устройствам с сенсорным экраном. Кроме того, такие приложения позволяют даже обычный монитор настольного компьютера превратить в сенсорный экран. При этом две кнопки мыши имитируют касание экрана пальцами и выполнение всех операций, свойственных сенсорным экранам (перемещение, перелистывание, вращение и масштабирование).

Материалы данной книги помогут, как начинающим программистам, так и имеющим опыт работы на Python, понять, как можно в одной инструментальной среде создавать приложения для любой платформы и для любого устройства. Освоив материал книги можно с минимальными потерями времени перейти к практическому программированию в данной инструментальной среде, которая упрощает создание как мобильных, так и настольных приложений.

В книге читатель найдет все материалы, необходимые для практического программирования, рассмотрены все классы, позволяющие создавать пользовательский интерфейс для мобильных и настольных приложений. Рассмотрены особенности языка программирования KV, который адаптирован для совместной работы с Python. Представлена структура и особенности приложений, создаваемых с использованием фреймворка Kivy. Основное внимание уделено использованию различных классов библиотеки KivyMD, а также показан ряд примеров создания простых мобильных приложений с использованием этой библиотеки.

Данная книга предназначена как для начинающих программистов (школьники и студенты), так и для специалистов с опытом, которые планируют заниматься или уже занимаются разработкой приложений с использованием **Python**. Сразу следует отметить, что программирование кроссплатформенных приложений требует от разработчиков больших знаний, умений и усилий, чем программирование традиционных приложений. Здесь кроме основного языка, который реализует логику приложения, требуется еще и знания языка разметки KV, необходимо иметь представления об объектно-ориентированном программировании (класса, объекты, свойства, методы). Если некоторые разделы книги вам покажутся трудными для понимания и восприятия, то не стоит отчаиваться. Нужно просто последовательно, по шагам повторить приведенные в книге примеры. После того, как вы увидите результаты работы программного кода, появится ясность – как работает тот или иной элемент изучаемого фреймворка.

В книге рассмотрены практически все элементарные действия, которые выполняют программисты, работая над реализацией кроссплатформенных приложений, имеется множество примеров и проверенных программных модулей. Рассмотрены базовых классы фреймворка Kivy и библиотеки KivyMD, методы и свойства каждого из классов и примеры их использования. Книга предназначена как для начинающих программистов, приступивших к освоению языка **Python**, так и имеющих опыт программирования на других языках.

Первая глава книги посвящена формированию инструментальной среды пользователя для разработки кроссплатформенных приложений (установка и настройка программных средств). Это в первую очередь интерпретатор **Python**, интерактивная среда разработки программного кода **PyCharm,** фреймворк **Kivy**, и библиотека **KivyMD**. С использованием **PyCharm** созданы простейшие первые приложения на **Kivy** и **KivyMD**

Вторая глава посвящена основным понятиям и определениям, которые используются в фреймворке **Kivy**. Описаны особенности языка **KV** и структура приложений на **Kivy**. Подробно описаны виджеты, которые используются для разработки пользовательского интерфейса, и правила работы с ними. Раскрывается понятие дерева виджетов, как основе пользовательского интерфейса. Подробно описаны виджеты для позиционирования элементов интерфейса в приложениях на **Kivy**. Описывается возможность идентификации виджетов и работы с их цветом. Рассмотрены классы Screen и ScreenManager для создания много экранных приложений.

В третьей главе приводятся основные понятия о структуре проектов на KivyMD и о базовых параметрах элементов пользовательского интерфейса, рассмотрены особенности много экранных приложений на основе менеджера экранов (ScreenManager). Описаны стили и темы для задания цветовой настройки приложений, стили шрифтов для вывода надписей

В четвертой главе описаны компоненты **KivyMD**, которые используются для позиционирования элементов интерфейса. Это некий аналог контейнеров, в которые вкладываются видимые пользователю элементы пользовательского интерфейса.

В пятой главе сделан обзор всех компонент пользовательского интерфейса KivyMD. Это самая объемная глава книги. Здесь говорится о том, для чего используется каждый из этих элементов, приводится пример его использования с полным листингом программного кода.

Показаны рисунки, иллюстрирующие, как внешний вид элемента интерфейса на экране приложения, так и его функциональные возможности.

Шестая глава посвящена обзору примеров кроссплатформенных приложений на Kivy и KivyMD. Здесь приводится описание функций приложений, полный листинг программных кодов, проиллюстрированы результаты их работы.

Седьмая заключительная глава посвящена созданию установочных и исполняемых файлов для приложений на Kivy и Python. Поскольку создание APK-пакетов для мобильных приложений под Android возможно только на компьютерах под управлением Linux, то в данной главе подробно описано, как на компьютер можно установить виртуальную машину VirtualDox и загрузить на нее операционную систему Linux. Далее описана утилита Buildozer, которая позволяет создавать APK-пакеты для мобильных приложений под Android и установочных файлов для мобильных приложений под iOS. Показан пример использования данной утилиты. Описаны возможности утилиты pyinstaller для создания исполняемых файлов для настольных приложений под Windows и MacOS (xOS). Показан пример использования данной утилиты.

На протяжении всей книги раскрываемые вопросы сопровождаются достаточно упрощенными, но полностью законченными примерами. Ход решения той или иной задачи сопровождается большим количеством иллюстративного материала. Желательно изучение тех или иных разделов выполнять непосредственно сидя за компьютером, тогда вы сможете последовательно повторять выполнение тех шагов, которые описаны в примерах, и тут же увидеть результаты своих действий. Это в значительной степени облегчает восприятие приведенных в книге материалов. Наилучший способ обучения – это практика. Все листинги программ приведены на языке **Python**. Это прекрасная возможность расширить свои знания о данном языке программирования, и понять, насколько он прост и удобен для создания кроссплатформенных приложений.

Итак, если Вас заинтересовали вопросы создания кроссплатформенных приложений с использованием **Python, Kivy** и **KivyMD**, то самое время перейти к изучению материалов этой книги.

Книга предназначена как для начинающих программистов, так и для подготовленных читателей, имеющих опыт работы с языком программирования **Python**. Кроме того, данная книга может быть использована как лабораторный практикум для школьников старших классов, студентов ВУЗов и слушателей компьютерных курсов при изучении практических приемов программирования кроссплатформенных приложений.

Примечание.

> К сожалению типографский макет издательства не учитывает тех особенностей текста, который необходим для издания литературы, связанной с информационными технологиями и программированием. Листинги программ содержат набор специальных символов – апострофы, двойные апострофы, тройные апострофы и другие специальные знаки, не обеспечивается необходимая поддержка табуляции и отступов. При верстки текста книги макет издательства автоматически заменяет нужные знаки и символы на те, которые «прошиты» в том или ином стиле макета. В связи с этим следует обратить внимание на некорректность некоторых фрагментов текста в листингах программ. В большей степени это касается фрагментов листингов, где присутствуют кавычки и апострофы. Это не ошибки автора, это недостатки стиля того или иного макета издательства. Чтобы обеспечить нужную табуляцию программного кода, вместо пробелов были использованы многоточия.

Глава 1. Инструментальные средства для разработки кроссплатформенных приложений на Python

Современным людям бывает просто необходимо иметь выход в Интернет со своего мобильного устройства. Средства сотовой связи обеспечивают подключение к сети Интернет с планшета или смартфона практически в любой точке вне дома или офиса, а специально созданные мобильные приложения позволяют решать как деловые задачи, так и выполнять развлекательные функции. Мобильные приложения действительно захватили нашу жизнь. Почти каждый день мы используем такие средства общения как WhatsApp и Viber, LinkedIn, обучающие приложения и игры.

Различные компании через мобильные приложения могут рассказать о своих товарах и услугах, найти потенциальных партнеров и клиентов, организовать продажу товаров. Рядовые пользователи взаимодействуют с торговыми точками, используют интернет-банкинг, общаются через мессенджеры, получают государственные услуги.

Для разработки мобильных приложений существует множество языков программирования, причем они позволяют создавать мобильные приложения для устройств, работающих либо только под Android, либо под iOS. Но из этих инструментальных средств хочется выделить связку: Python, фреймворк Kivy и библиотека KivyMD.

Kivy – это фреймворк Python, который упрощает создание кроссплатформенных приложений, способных работать в Windows, Linux, Android, OSX, iOS и мини компьютерах типа Raspberry Pi. Это популярный пакет для создания графического интерфейса на Python, который набирает большую популярность благодаря своей простоте в использовании, хорошей поддержке сообщества и простой интеграции различных компонентов.

Библиотека KivyMD построена на основе фреймворка Kivy. Это набор виджетов Material Design (MD) для использования с Kivy. Данная библиотека предлагает достаточно элегантные компоненты для создания интерфейса – UI (user interface – пользовательский интерфейс), в то время как программный код на Kivy используется для написания основных функций приложения, например, доступ к ресурсам Интернет, обращение к элементам мобильного устройства, таким как видеокамера, микрофон, GPS приемник и т. п.

Используя Python и Kivy можно создавать действительно универсальные приложения, которые из одного программного кода будут работать:

– на настольных компьютерах (OS X, Linux, Windows);

– на устройствах iOS (iPad, iPhone);

– на Android-устройствах (планшеты, смартфоны);

– на любых других устройства с сенсорным экраном, поддерживающие TUIO (Tangible User Interface Objects).

Kivy дает возможность написать программный код один раз и запустить его на совершенно разных платформах.

Для ускорения процесса написания программного кода удобно использовать специализированную инструментальную среду – так называемую *интегрированную среду разработки* (IDE, Integrated Development Environment). Эта среда включает полный комплект средств, необходимых для эффективного программирования на Python. Обычно в состав IDE входят текстовый редактор, компилятор или интерпретатор, отладчик и другое программное обеспечение. Использование IDE позволяет увеличить скорость разработки программ (при условии предварительного обучения работе с такой инструментальной средой). Для написания программного кода на Python наиболее популярной инструментальной средой является IDE

PyCharm – это кроссплатформенная среда разработки, которая совместима с Windows, macOS, Linux.

Из материалов этой главы вы узнаете:

– что такое мобильные приложения;

– как установить и проверить работу интерпретатора Python;

– как установить интегрированную среду разработки PyCharm;

– с помощью какого инструментария можно загрузить в Python дополнительные пакеты программных средств

– как загрузить фреймворк Kivy и библиотеку KivyMD;

– как создать первое простейшее приложение с использованием Kivy и KivyMD.

1.1. Мобильные приложения

Мобильное приложение (от англ. «Mobile app») это программное обеспечение, предназначенное для работы на смартфонах, планшетах и других мобильных устройствах, разработанное для конкретной платформы (iOS, Android, Windows Phone и т. д.). Многие мобильные приложения предустановлены на самом устройстве или могут быть загружены на него из онлайновых магазинов приложений.

Буквально 15—20 лет назад на вопрос, что такое мобильное приложение, владелец сотового телефона не нашел бы ответа. Однако с появлением смартфонов возможности мобильных устройств перестали ограничиваться функциями звонков, отправки СМС и простейшими играми. Сегодня можно сказать, что мобильное приложение – это специально разработанное программное обеспечение под функциональные возможности различных мобильных устройств. Назначение программного обеспечения может быть самым разнообразным: сервисы, магазины, развлечения, игры, онлайн-помощники и другое. Эти приложения скачиваются и устанавливаются самим пользователем через специальные платформы, такие как App Store, Google Play бесплатно или за определенную плату.

Довольно часто пользователи путаются в функциональных различиях мобильной версией сайта и мобильного приложения для смартфона, планшета или другого гаджета. Мобильный вариант сайта представляет собой переработанный, а в некоторых вариантах адаптированный дизайн и контент веб-страниц для удобного просмотра на небольшом дисплее смартфона. Самый простой способ – это создать копию основного сайта для настольного компьютера и подстроить его под разрешение мобильного устройства. Более сложный вариант – это создать новый дизайн web-страниц, с которыми будет удобно взаимодействовать пользователю посредством сенсорного экрана.

Мобильное приложение – это программный пакет, функционал и дизайн которого «заточен» под возможности конкретной мобильной платформы. Вот несколько основных плюсов мобильных приложений:

– интерфейс программы создан конкретно под работу на мобильном устройстве через сенсорный экран;

– удобная и понятная для пользователей гаджетов навигация через мобильное меню;

– лучшее взаимодействие с пользователем через сообщения, пуш-уведомления, напоминания;

– приложение может выполнять функции даже в фоновом режиме, чего нельзя сказать о сайте;

– для работы с программой не нужно открывать браузер, а многие приложения поддерживают свои функций и при отключенном интернете;

– реализована возможность хранение персональных данных пользователя (эта функция расширяет возможности персонализации приложений, например, вызывает такси на домашний адрес, запись на прием к врачу по медицинскому полису и т.п.;

– более гибкая, по сути прямая, обратная связь с торговой компанией или иным сервисом;

– можно задействовать больше ресурсов (например, подключить геолокацию, видеокамеру, датчик ускорения, Bluetooth модуль и т.п.).

На самом деле функционал мобильных приложений уже давно превзошел адаптированные сайты. Сегодня можно скачать и установить на смартфон программы для бизнеса, обучения, органайзеры с опциями напоминания, развлекательный контент и игры, программы различных сервисных служб.

Для разработки мобильных приложений используются различные языка программирования. Среди них можно выделить:

– Java – один из самых популярных языков программирования, который предлагает широкий спектр функций, он считается лучшим языком для разработки под Android;

– Kotlin – это язык программирования со статической типизацией, его можно использовать в сочетании с JAVA для создания более эффективных и высокопроизводительных приложений под Android;

– Swift – в основном используется для разработки приложений для iOS. Достаточно долгое время Swift сохранял монополию в бизнесе по разработке приложений для iOS;

– Dart – это быстрый, объектно-ориентированный язык программирования, основанный на парадигме, который используется для разработки кроссплатформенных приложений. Этот язык программирования, созданный Google, позиционируется в качестве альтернативы JavaScript;

– C # – является еще одним объектно-ориентированным языком, который широко используется для мобильной разработки. Он в основном используется для платформы Windows Mobile;

– C ++ -считается хорошим выбором для разработки приложений для Android. То, что прочно удерживает рынок мобильной индустрии, это системы на базе Android;

– Xamarin – это бесплатная кроссплатформенная среда разработки мобильных приложений с открытым исходным кодом, используемая для создания приложений с использованием. NET и C #. Xamarin расширяет платформу для разработчиков. NET, предоставляя пользователям доступ к инструментам и технологиям для разработки приложений для iOS, Android и Windows.

Большинство из этих языков программирования ориентированы на разработку приложений под определенную платформу (например, Java – под Android, Swift под iOS). В отличие от них Python с пакетами Kivy и KivyMD действительно универсальный набор инструментов разработок кроссплатформенных приложений для любых операционных систем настольных компьютеров и любых платформ мобильных устройств.

1.2. Интерпретатор Python

Язык программирования Python является весьма мощным инструментальным средством для разработки различных систем. Однако наибольшую ценность представляет даже не столько сам этот язык программирования, сколько набор подключаемых библиотек, на уровне которых уже реализованы все необходимые процедуры и функции. Разработчику достаточно написать несколько десятков строк программного кода, чтобы подключить требуемые библиотеки, создать набор необходимых объектов, передать им исходные данные и отобразить итоговые результаты.

Для установки интерпретатора Python на компьютер, прежде всего надо загрузить его дистрибутив. Скачать дистрибутив Python можно с официального сайта, перейдя по ссылке: **https://www.python.org/downloads/** (рис. 1.1).

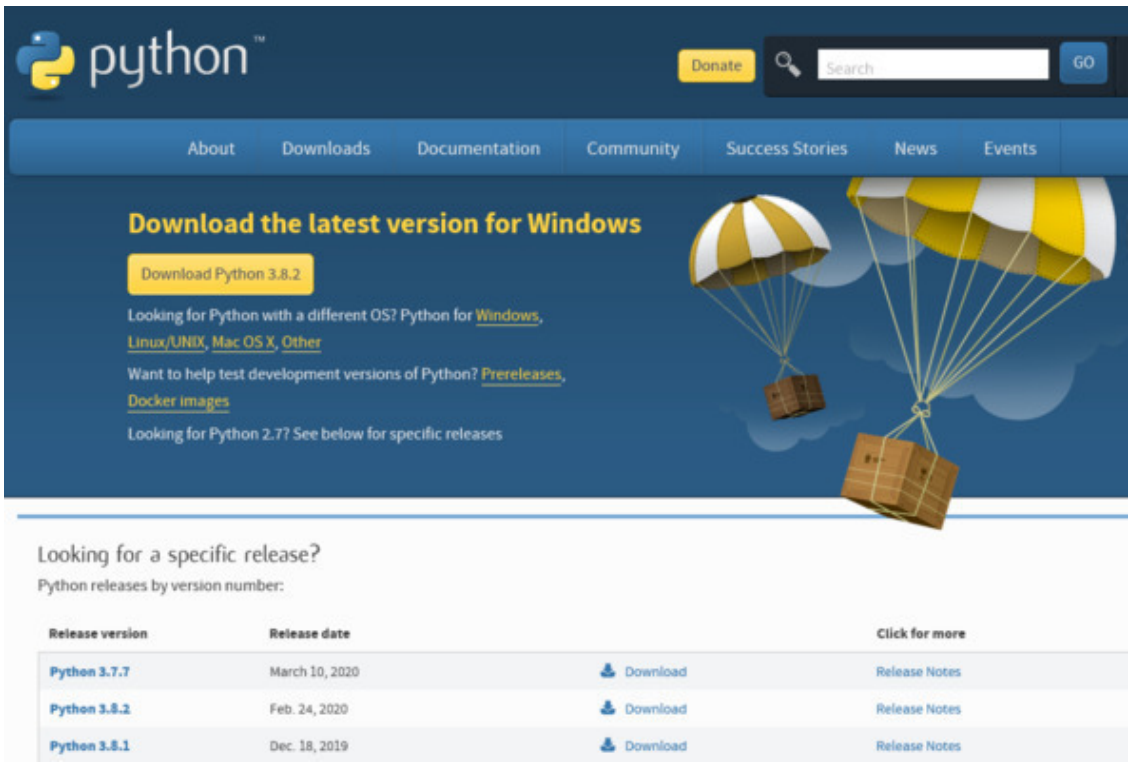


Рис. 1.1. Сайт для скачивания дистрибутива языка программирования Python

1.2.1. Установка Python в Windows

Для операционной системы Windows дистрибутив Python распространяется либо в виде исполняемого файла (с расширением exe), либо в виде архивного файла (с расширением zip). На момент подготовки этой книги была доступна версия Python 3.8.3.

Порядок установки Python в Windows следующий:

– Запустите скачанный установочный файл.

– Выберите способ установки (рис. 1.2).

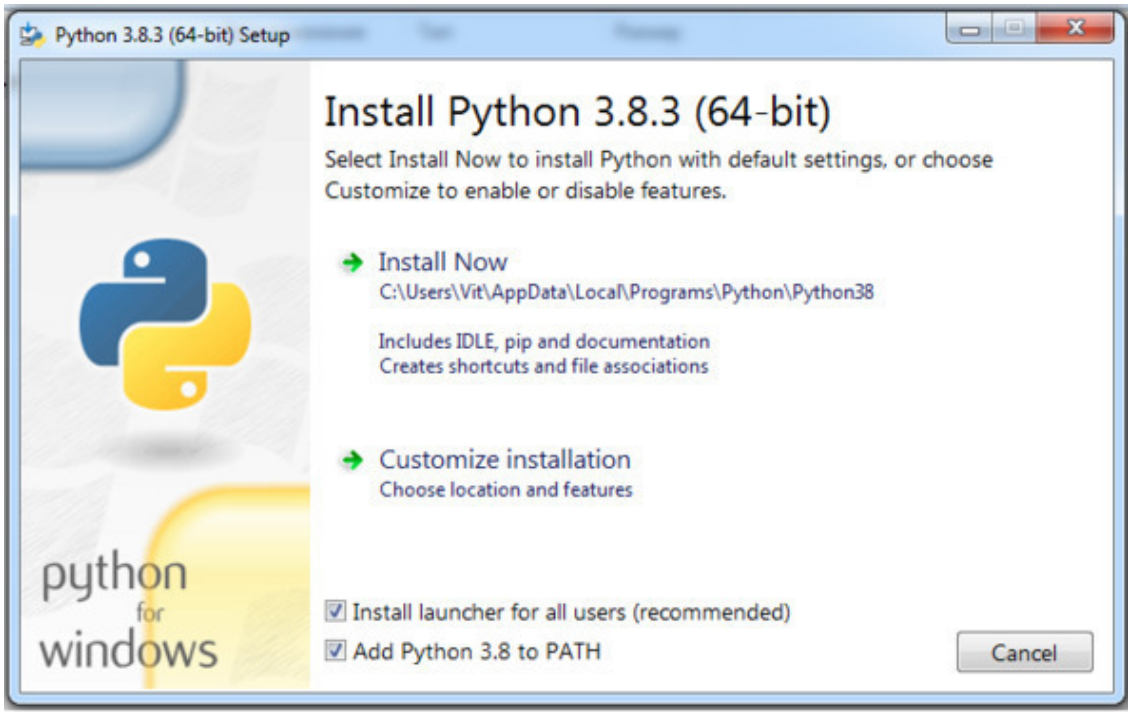


Рис. 1.2. Выбор способа установки Python

В открывшемся окне предлагаются два варианта: **Install Now** и **Customize installation**:

– при выборе **Install Now** Python установится в папку по указанному в окне пути. Помимо самого интерпретатора будут инсталлированы IDLE (интегрированная среда разработки), pip (пакетный менеджер) и документация, а также созданы соответствующие ярлыки и установлены связи (ассоциации) файлов, имеющих расширение py, с интерпретатором Python;

– **Customize installation** – это вариант настраиваемой установки. Опция **Add Python 3.8 to PATH** нужна для того, чтобы появилась возможность запускать интерпретатор без указания полного пути до исполняемого файла при работе в командной строке.

– Отметьте необходимые опции установки, как показано на рис. 1.3 (доступно при выборе варианта **Customize installation**).

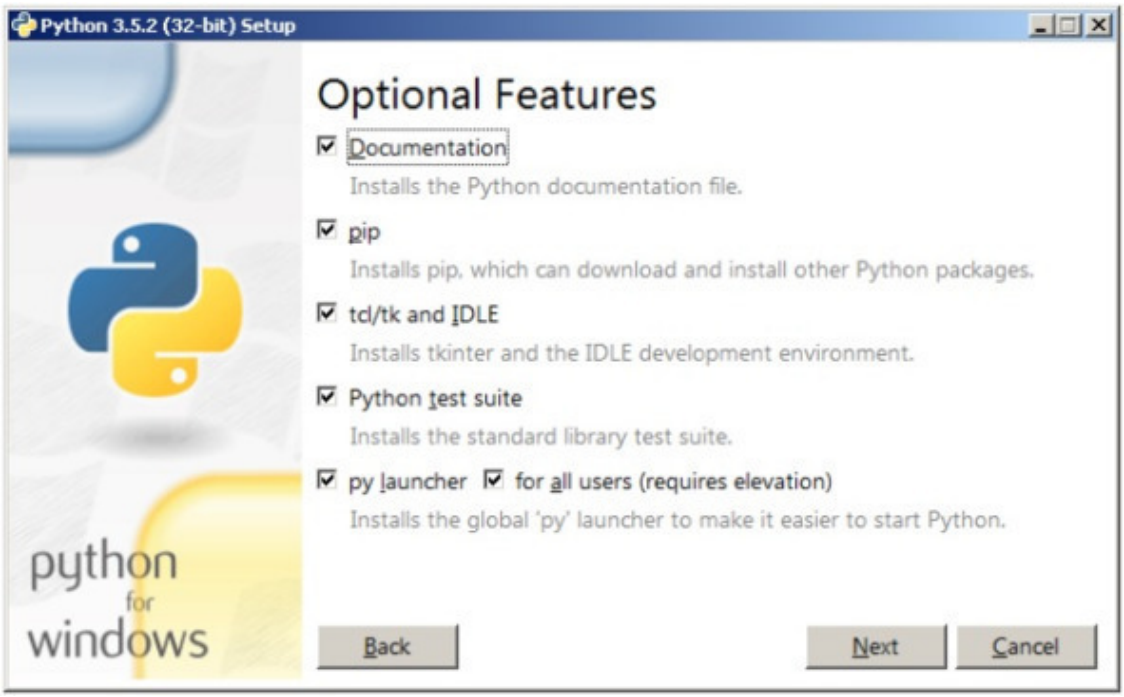


Рис. 1.3. Выбор опций установки Python

На этом шаге нам предлагается отметить дополнения, устанавливаемые вместе с интерпретатором Python. Рекомендуется выбрать как минимум следующие опции:

– **Documentation** – установка документации;

– **pip** – установка пакетного менеджера pip;

– **tcl/tk and IDLE** – установка интегрированной среды разработки (IDLE) и библиотеки для построения графического интерфейса (tkinter).

– На следующем шаге в разделе **Advanced Options** (Дополнительные опции) выберите место установки, как показано на рис. 1.4 (доступно при выборе варианта **Customize installation**).

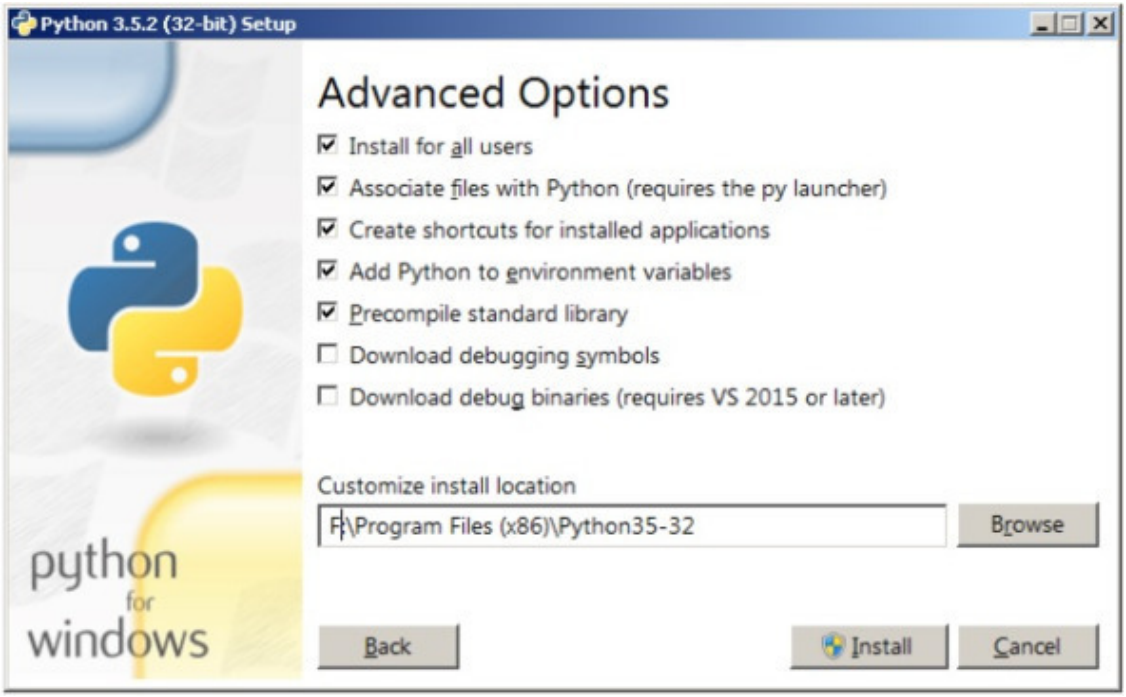


Рис. 1.4. Выбор места установки Python

Помимо указания пути, этот раздел позволяет внести дополнительные изменения в процесс установки с помощью опций:

– **Install for all users** – установить для всех пользователей. Если не выбрать эту опцию, то будет предложен вариант инсталляции в папку пользователя, устанавливающего интерпретатор;

– **Associate files with Python** – связать файлы, имеющие расширение py, с Python. При выборе этой опции будут внесены изменения в Windows, позволяющие Python запускать скрипты по двойному щелчку мыши;

– **Create shortcuts for installed applications** – создать ярлыки для запуска приложений;

– **Add Python to environment variables** – добавить пути до интерпретатора Python в переменную PATH;

– **Precomple standard library** – провести перекомпиляцию стандартной библиотеки.

Последние два пункта связаны с загрузкой компонентов для отладки, их мы устанавливать не будем.

– После успешной установки Python вас ждет следующее сообщение (рис. 1.5).

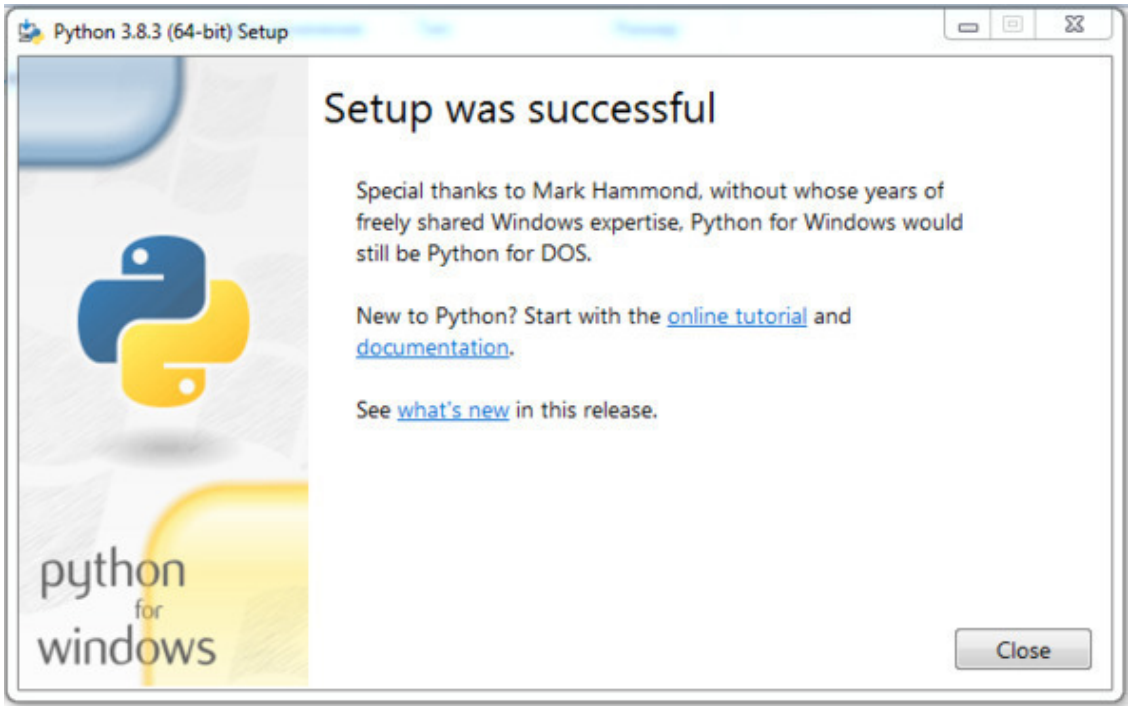


Рис. 1.5. Финальное сообщение после установки Python

1.2.2.Установка Python в Linux

Чаще всего интерпретатор Python уже входит в состав дистрибутива Linux. Это можно проверить, набрав в окне терминала команду:

> python

или

> python3

В первом случае, вы запустите Python 2, во втором – Python 3. В будущем, скорее всего, во все дистрибутивы Linux, включающие Python, будет входить только третья его версия. Если у вас при попытке запустить Python выдается сообщение о том, что он не установлен или установлен, но не тот, что вы хотите, то у вас есть возможность взять его из репозитория.

Для установки Python из репозитория Ubuntu воспользуйтесь командой:

> sudo apt-get install python3

1.2.3. Проверка интерпретатора Python

Для начала протестируем интерпретатор в командном режиме. Если вы работаете в Windows, то нажмите комбинацию клавиш <Win> + <R> и в открывшемся окне введите: python. В Linux откройте окно терминала и в нем введите: python3 (или python).

В результате Python запустится в командном режиме. Выглядеть это будет примерно так, как показано на рис. 1.6 (иллюстрация приведена для Windows, в Linux результат будет аналогичным).

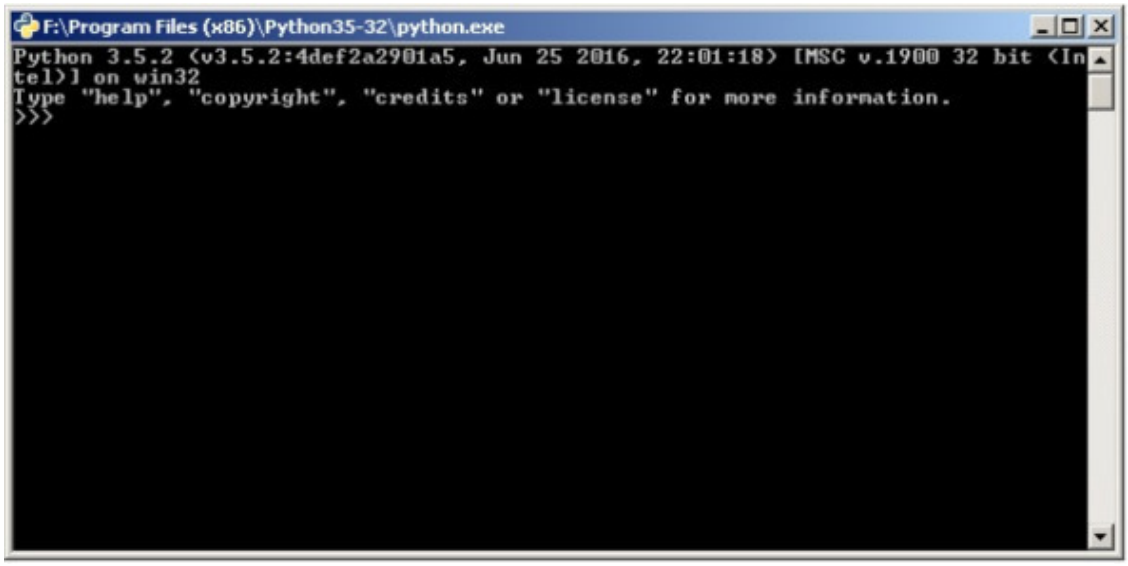


Рис. 1.6. Результат запуска интерпретатора Python в окне терминала

В этом окне введите программный код следующего содержания:
print («Hello, World!»)
В результате вы увидите следующий ответ (рис. 1.7).

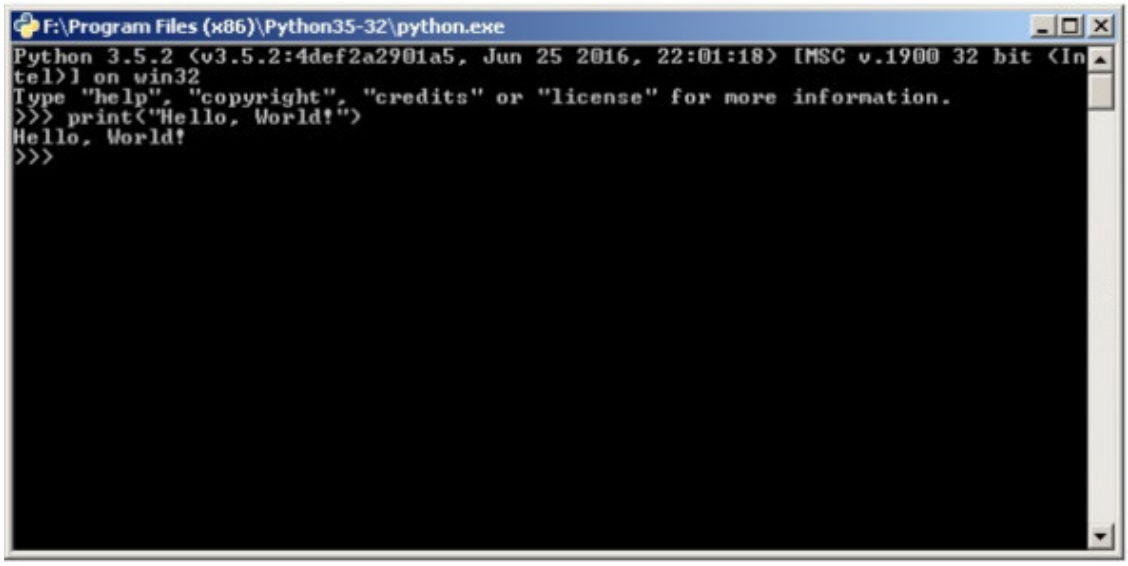


Рис. 1.7. Результат работы программы на Python в окне терминала

Получение такого результата означает, что установка интерпретатора Python прошла без ошибок.

1.3. Интерактивная среда разработки программного кода PyCharm

В процессе разработки программных модулей удобнее работать в интерактивной среде разработки (IDE), а не в текстовом редакторе. Для Python одним из лучших вариантов считается IDE PyCharm от компании JetBrains. Для скачивания его дистрибутива перейдите по ссылке: **https://www.jetbrains.com/pycharm/download/** (рис. 1.8).

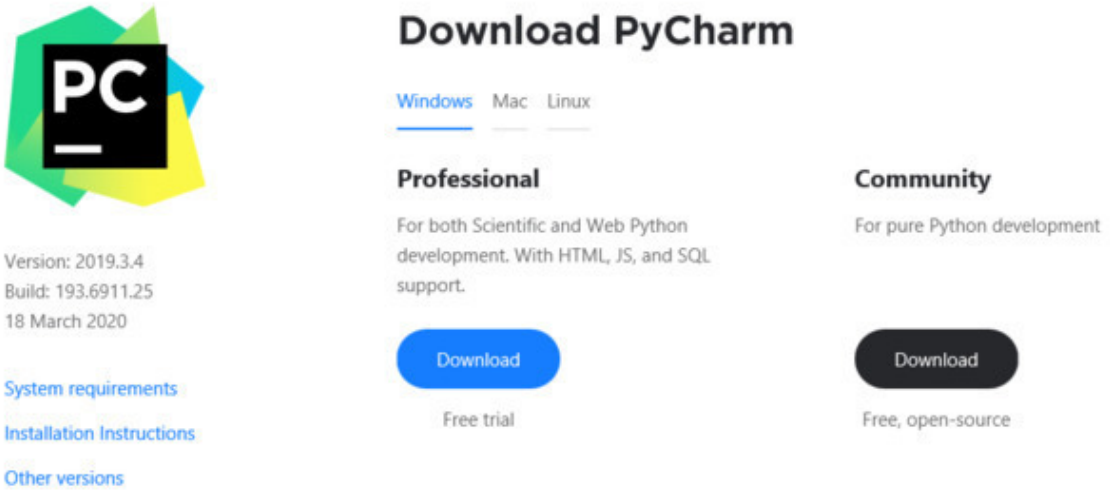


Рис. 1.8. Главное окно сайта для скачивания дистрибутива PyCharm

Эта среда разработки доступна для Windows, Linux и macOS. Существуют два вида лицензии PyCharm: **Professional** и **Community**. Мы будем использовать версию **Community**, поскольку она бесплатная и ее функционала более чем достаточно для наших задач. На момент подготовки этой книги была доступна версия PyCharm 2020.1.2.

1.3.1. Установка PyCharm в Windows

Запустите на выполнение скачанный дистрибутив PyCharm (рис. 1.9).

Рис. 1.9. Начальная заставка при инсталляции PyCharm

Выберите путь установки программы (рис. 1.10).

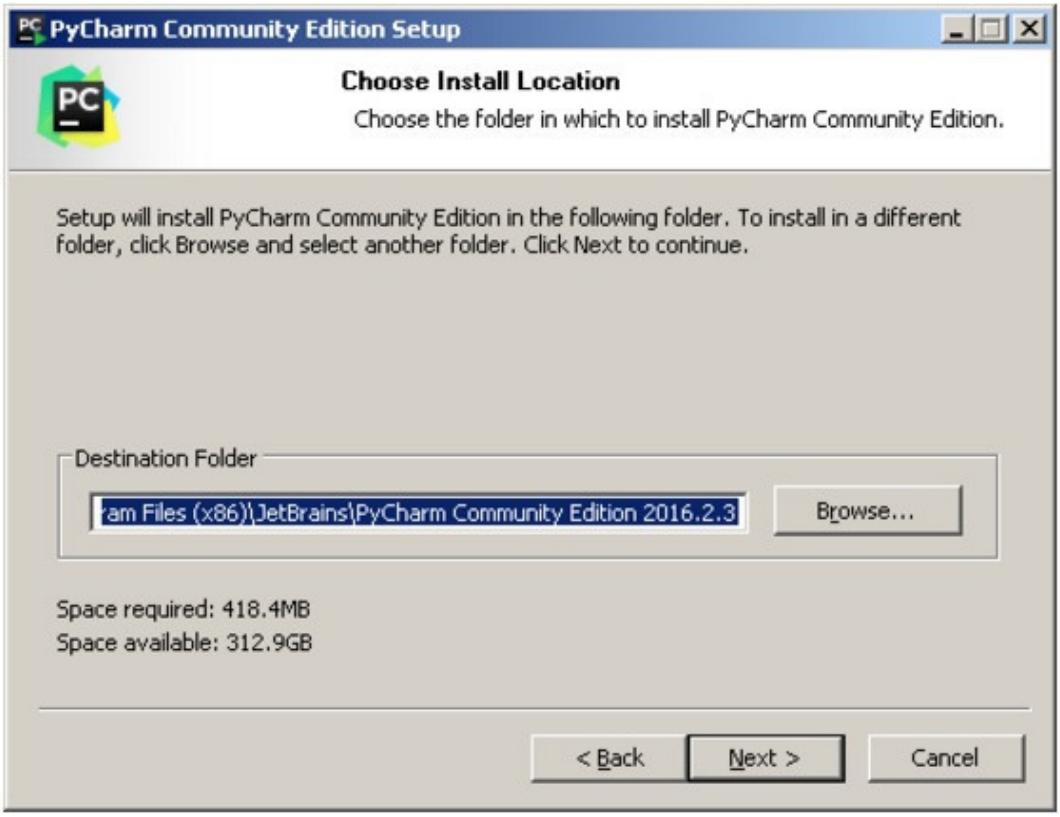


Рис. 1.10. Выбор пути установки PyCharm

Укажите ярлыки, которые нужно создать на рабочем столе (запуск 32- или 64-разрядной версии PyCharm), и отметьте флажком опцию. **py** d области **Create associations,** если требуется ассоциировать с PyCharm файлы с расширением py (рис. 1.11).

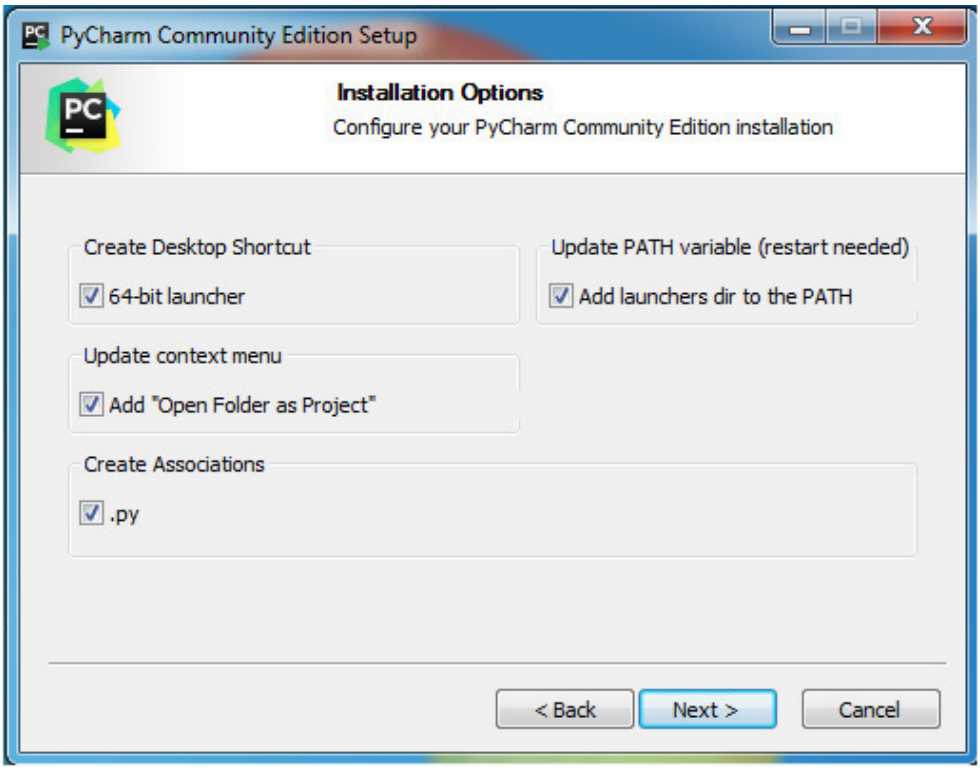


Рис. 1.11. Выбор разрядности устанавливаемой среды разработки PyCharm

Выберите имя для папки в меню **Пуск** (рис. 1.12).

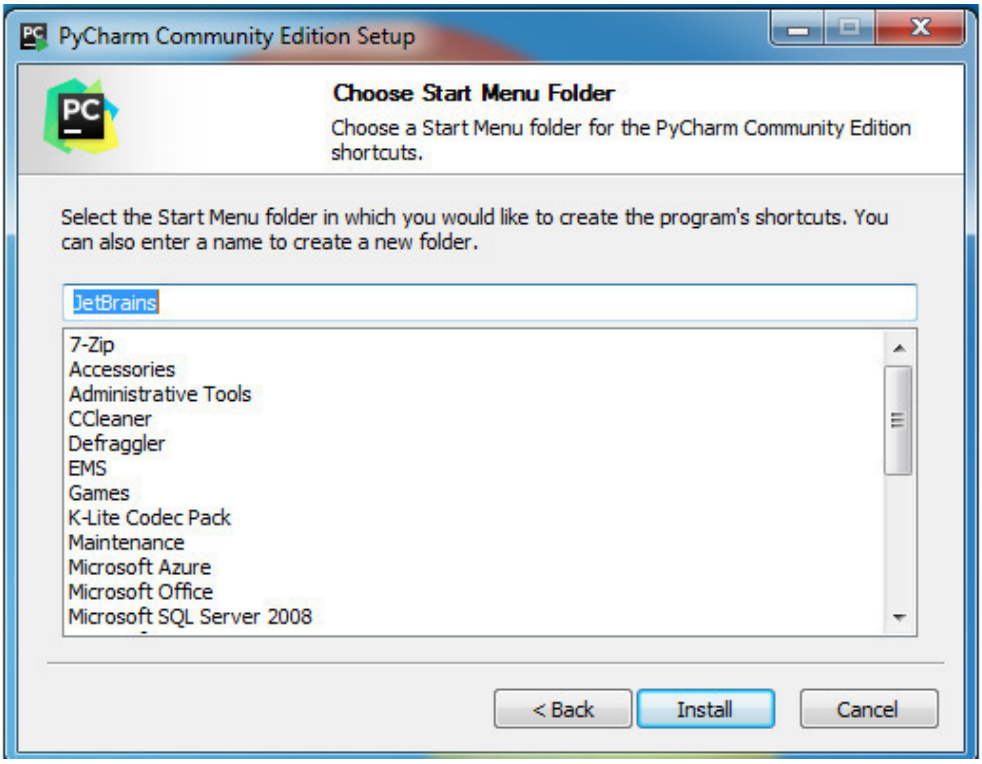


Рис. 1.12. Выбор имени папки для PyCharm в меню **Пуск**

Далее PyCharm будет установлен на ваш компьютер (рис. 1.13).

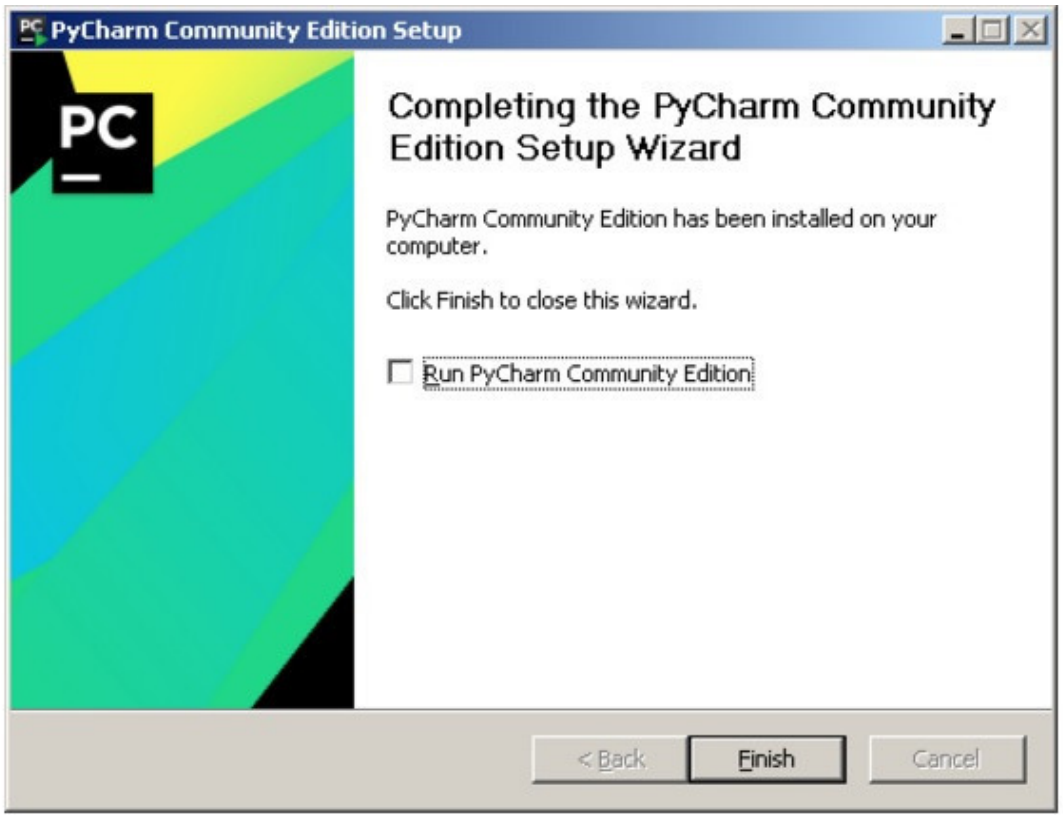


Рис. 1.13. Финальное окно установки пакета PyCharm

1.3.2. Установка PyCharm в Linux

Скачайте с сайта программы ее дистрибутив на свой компьютер.
Распакуйте архивный файл, для чего можно воспользоваться командой:
> tar xvf имя_архива. tar. gz
Результат работы этой команды представлен на рис. 1.14.

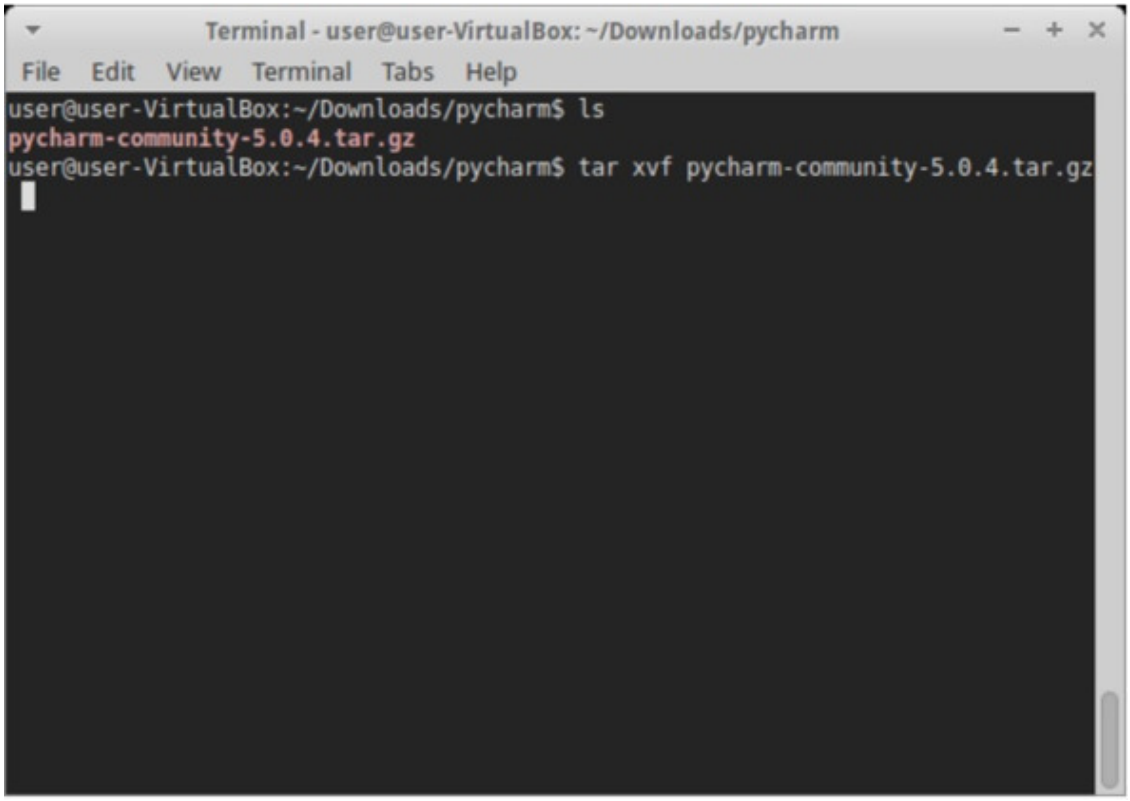


Рис. 1.14. Результат работы команды распаковки архива PyCharm

Перейдите в каталог, который был создан после распаковки дистрибутива, найдите в нем подкаталог bin и зайдите в него. Запустите установку PyCharm командой:
> ./pycharm.sh
Результат работы этой команды представлен на рис. 1.15.

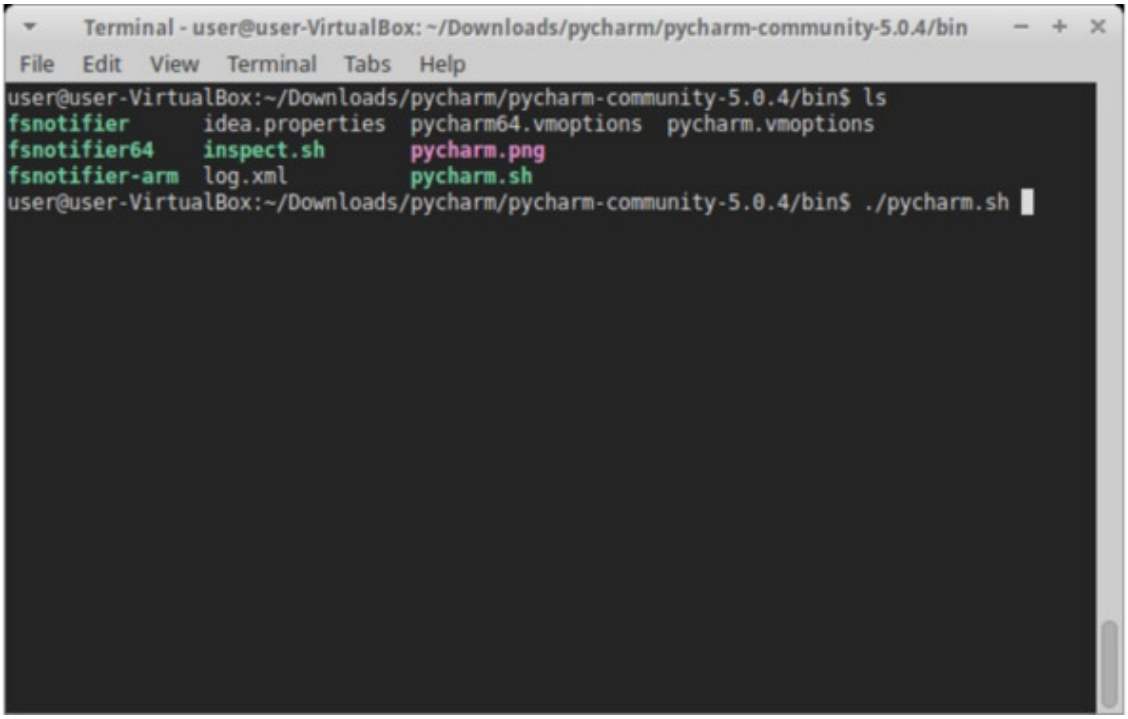


Рис. 1.15. Результаты работы команды инсталляции PyCharm

1.3.3. Проверка PyCharm

Запустите PyCharm и выберите вариант **Create New Project** в открывшемся окне (рис. 1.16).

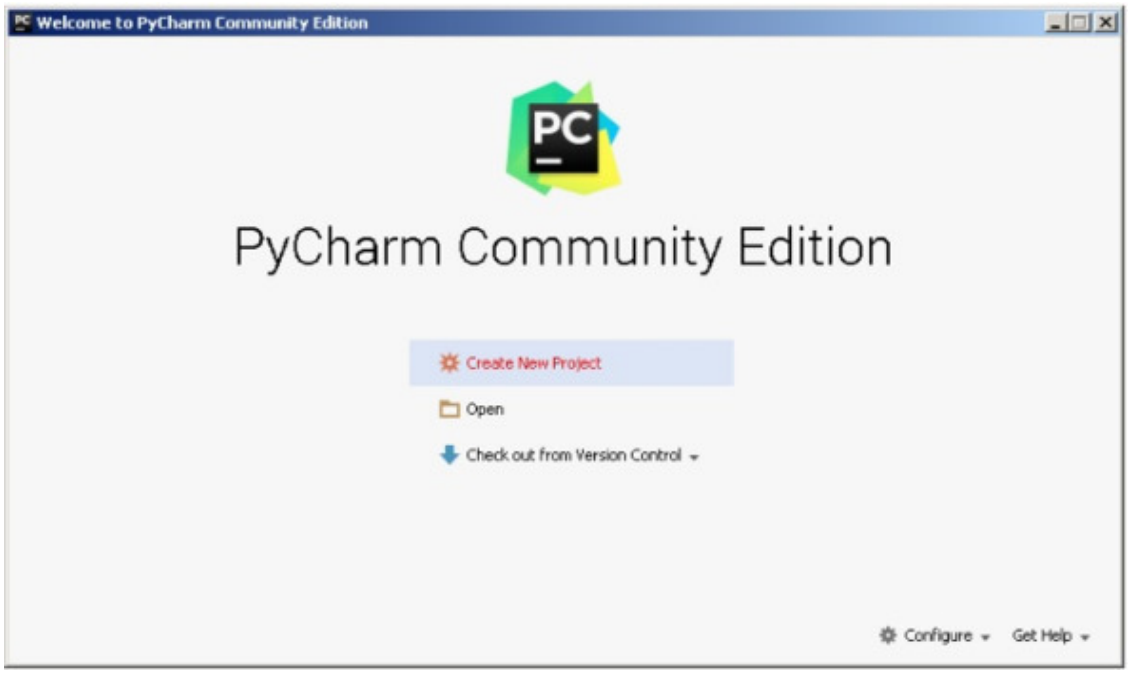


Рис. 1.16. Создание нового проекта в среде разработки PyCharm

Укажите путь до создаваемого проекта Python и интерпретатор, который будет использоваться для его запуска и отладки (рис. 1.17).

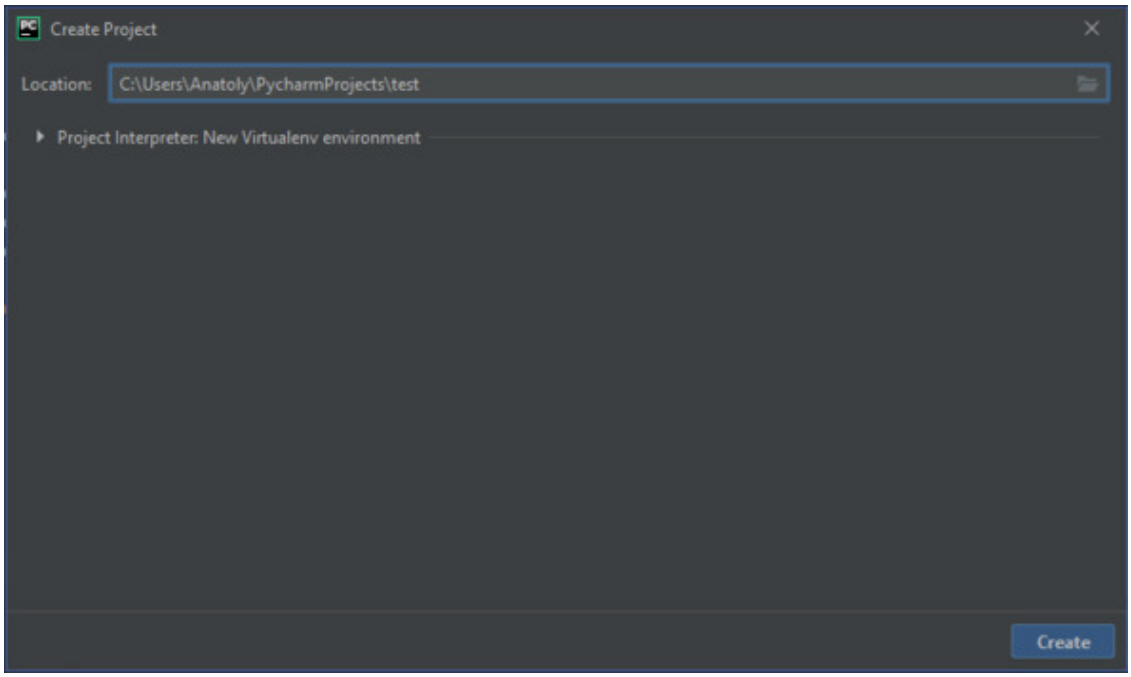


Рис. 1.17. Указание пути до проекта в среде разработки PyCharm

Добавьте в проект файл, в котором будет храниться программный код Python (рис. 1.18).

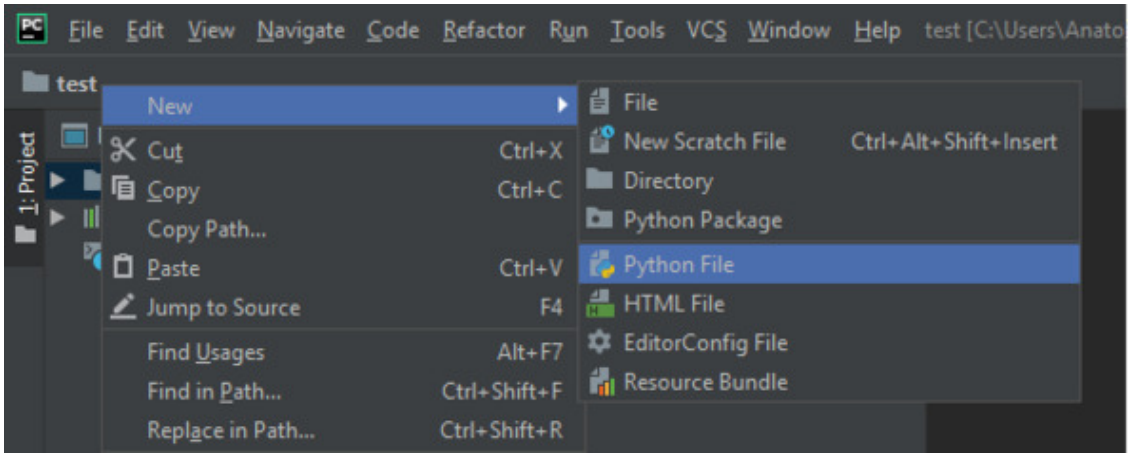


Рис. 1.18. Добавление в проект файла для программного кода на Python

Введите одну строчку кода программы (рис. 1.19).

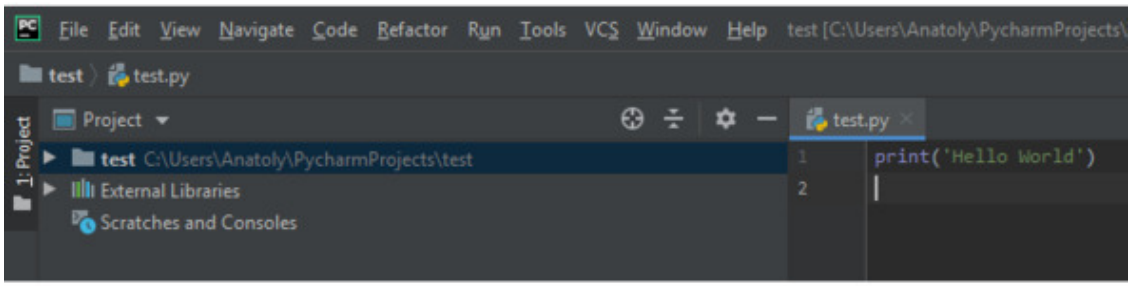


Рис. 1.19. Одна строка программного кода на Python в среде разработки PyCharm

Запустите программу командой **Run** (рис. 1.20).

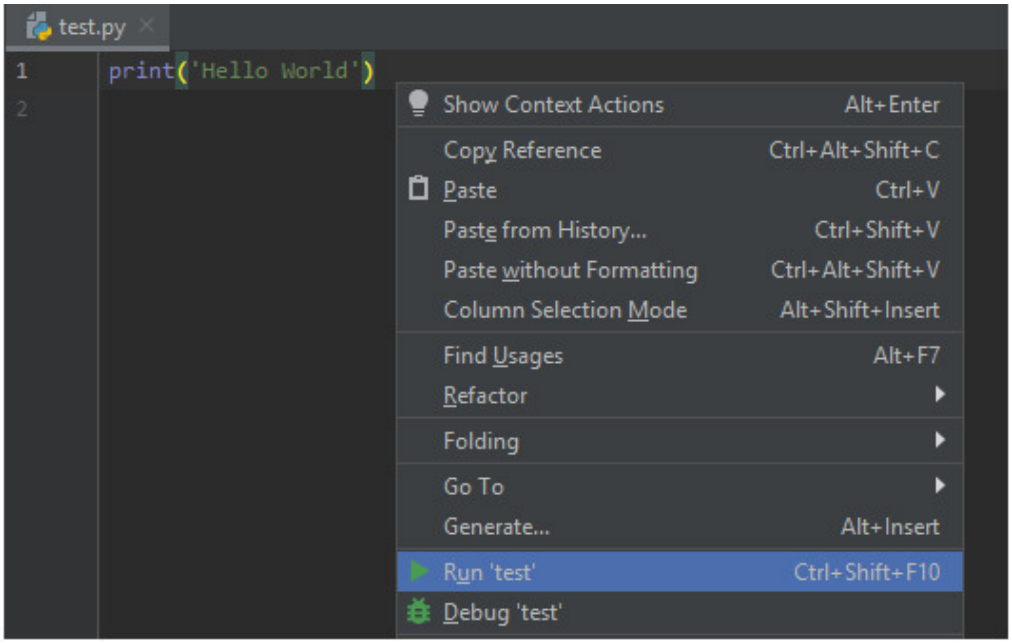


Рис. 1.20. Запуск программного кода на Python в среде разработки PyCharm

В результате в нижней части экрана должно открыться окно с выводом результатов работы программы (рис. 1.21).

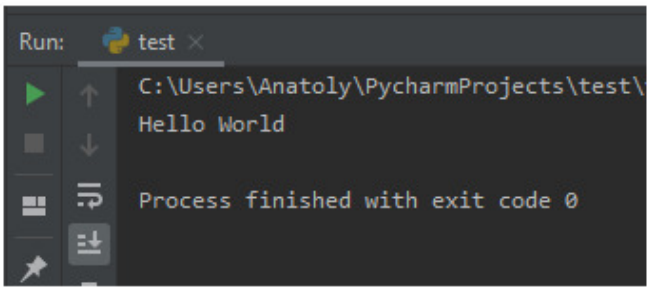


Рис. 1.21. Вывод результатов работы программы на Python в среде разработки PyCharm

Можно перейти к следующему разделу.

1.4. Инструментарий для загрузки в Python пакетов программных средств

В процессе разработки программного обеспечения на Python часто возникает необходимость воспользоваться пакетом (библиотекой), который в текущий момент отсутствует на вашем компьютере.

В этом разделе вы узнаете о том, откуда можно взять нужный вам дополнительный инструментарий для разработки ваших программ. В частности:

– где взять отсутствующий пакет;
– как установить pip – менеджер пакетов в Python;
– как использовать pip;
– как установить пакет;
– как удалить пакет;
– как обновить пакет;
– как получить список установленных пакетов;
– как выполнить поиск пакета в репозитории.

1.4.1. Репозиторий пакетов программных средств PyPI

Необходимость в установке дополнительных пакетов возникнет достаточно часто, поскольку решение практических задач обычно выходит за рамками базового функционала, который предоставляет Python. Это, например, создание веб-приложений, обработка изображений, распознавание объектов, нейронные сети и другие элементы искусственного интеллекта, геолокация и т. п. В таком случае, необходимо узнать, какой пакет содержит функционал, который вам необходим, найти его, скачать, разместить в нужном каталоге и начать использовать. Все указанные действия можно выполнить и вручную, однако этот процесс поддается автоматизации. К тому же скачивать пакеты с неизвестных сайтов может быть весьма опасно.

В рамках Python все эти задачи автоматизированы и решены. Существует так называемый Python Package Index (PyPI) – репозиторий, открытый для всех разработчиков на Python, в котором вы можете найти пакеты для решения практически любых задач. При этом у вас отпадает необходимость в разработке и отладке сложного программного кода – вы можете воспользоваться уже готовыми и проверенными решениями огромного сообщества программистов на Python. Вам нужно просто подключить нужный пакет или библиотеку к своему проекту и активировать уже реализованный в них функционал. В этом и заключается преимущества Python перед другими языками программирования, когда небольшим количеством программного кода можно реализовать решение достаточно сложных практических задач. Там также есть возможность выкладывать свои пакеты. Для скачивания и установки нужных модулей в ваш проект используется специальная утилита, которая называется pip. Сама аббревиатура, которая на русском языке звучит как «пип», фактически раскрывается как «установщик пакетов» или «предпочитаемый установщик программ». Это утилита командной строки, которая позволяет устанавливать, переустанавливать и деинсталлировать PyPI пакеты простой командой pip.

1.4.2. Менеджер пакетов в Python – pip

Менеджер пакетов pip – это консольная утилита (без графического интерфейса). После того, как вы ее скачаете и установите, она пропишется в PATH и будет доступна для использования. Эту утилиту можно запускать как самостоятельно – например, через терминал в Windows или Linux, а также в терминальном окне PyCharm командой:

> pip <*аргументы*>

pip можно запустить и через интерпретатор Python:

> python -m pip <аргументы>

Ключ -m означает, что мы хотим запустить модуль (в нашем случае pip).

При развертывании современной версии Python (начиная с Python 2.7.9 и более поздних версий), pip устанавливается автоматически. В PyCharm проверить наличие модуля pip достаточно просто – для этого нужно войти в настройки проекта через меню **File | Settings | Project Interpreter**. Модуль pip должен присутствовать в списке загруженных пакетов и библиотек (рис. 1.22).

Рис. 1.22. Проверка наличия в проекте модуля pip

В случае отсутствия в списке этого модуля последнюю его версию можно загрузить, нажав на значок + в правой части окна и выбрав модуль pip из списка (рис. 1.23).

Рис. 1.23. Загрузка модуля pip

1.4.3. Использование менеджера пакетов pip

Здесь мы рассмотрим основные варианты использования pip: установку пакетов, удаление и обновление пакетов.

Pip позволяет установить самую последнюю версию пакета, конкретную версию или воспользоваться логическим выражением, через которое можно определить, что вам, например, нужна версия не ниже указанной. Также есть поддержка установки пакетов из репозитория. Рассмотрим, как использовать эти варианты (здесь Name – это имя пакета).

– Установка последней версии пакета:

> pip install Name

– Установка определенной версии:

> pip install Name==3.2

– Установка пакета с версией не ниже 3.1:

> pip install Name> =3.1

– Для того чтобы удалить пакет, воспользуйтесь командой:

> pip uninstall Name

– Для обновления пакета используйте ключ – upgrade:

> pip install – upgrade Name

– Для вывода списка всех установленных пакетов служит команда:

> pip list

– Если вы хотите получить более подробную информацию о конкретном пакете, то используйте аргумент show:

> pip show Name

– Если вы не знаете точного названия пакета или хотите посмотреть на пакеты, содержащие конкретное слово, то вы можете это сделать, используя аргумент search:

> pip search «test».

Если вы запускаете pip в терминале Windows, то терминальное окно автоматически закроется после того, как эта утилита завершит свою работу. При этом вы просто не успеете увидеть результаты ее работы. Чтобы терминальное окно не закрывалось автоматически, команды pip нужно запускать в нем с ключом /k. Например, запуск процедуры установки пакета tensorflow должен выглядеть так, как показано на рис. 1.24.

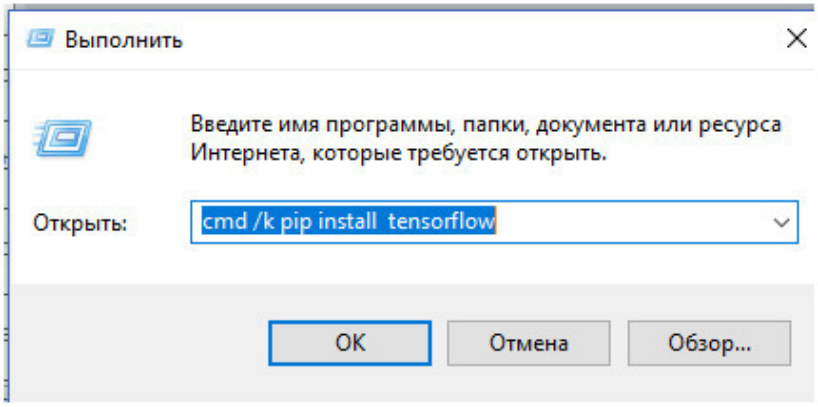


Рис. 1.24. Выполнение команды модуля pip в терминальном окне Windows

Если же пакет pip запускается из терминального окна PyCharm, то в использовании дополнительных ключей нет необходимости, так как терминальное окно после завершения работы программ не закрывается. Пример выполнения той же команды в терминальном окне PyCharm показан на рис. 1.25.

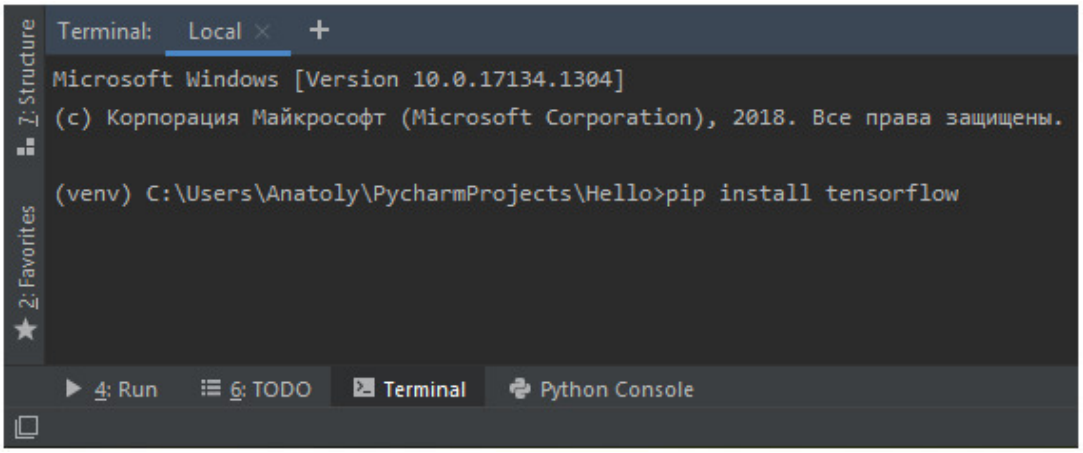


Рис. 1.25. Выполнение команды модуля pip в терминальном окне PyCharm

1.5. Загрузка фреймворка Kivy и библиотеки KivyMD

Итак, основной инструментарий для разработки программ на языке Python установлен, и мы можем перейти к установке дополнительных модулей, с помощью которых можно вести разработку кроссплатформенных мобильных и настольных приложений. В этом разделе мы установим фреймворк Kivy и библиотеку KivyMD.

Запустим среду разработки PyCharm и создадим в ней новый проект с именем Kivy_Project. Для этого в главном меню среды выполните команду **File | New Project** (рис. 1.26).

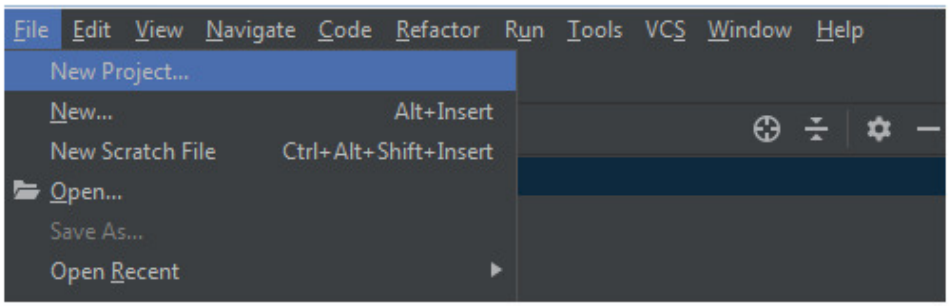


Рис. 1.26. Создание нового проекта в среде разработки PyCharm

Откроется окно, где вы можете задать имя создаваемому проекту, определить виртуальное окружение для этого проекта и указать каталог, в котором находится интерпретатор Python. В данном окне необходимо задать новому проекту имя Kivy_Project, после чего нажать кнопку **Create** (рис. 1.27).

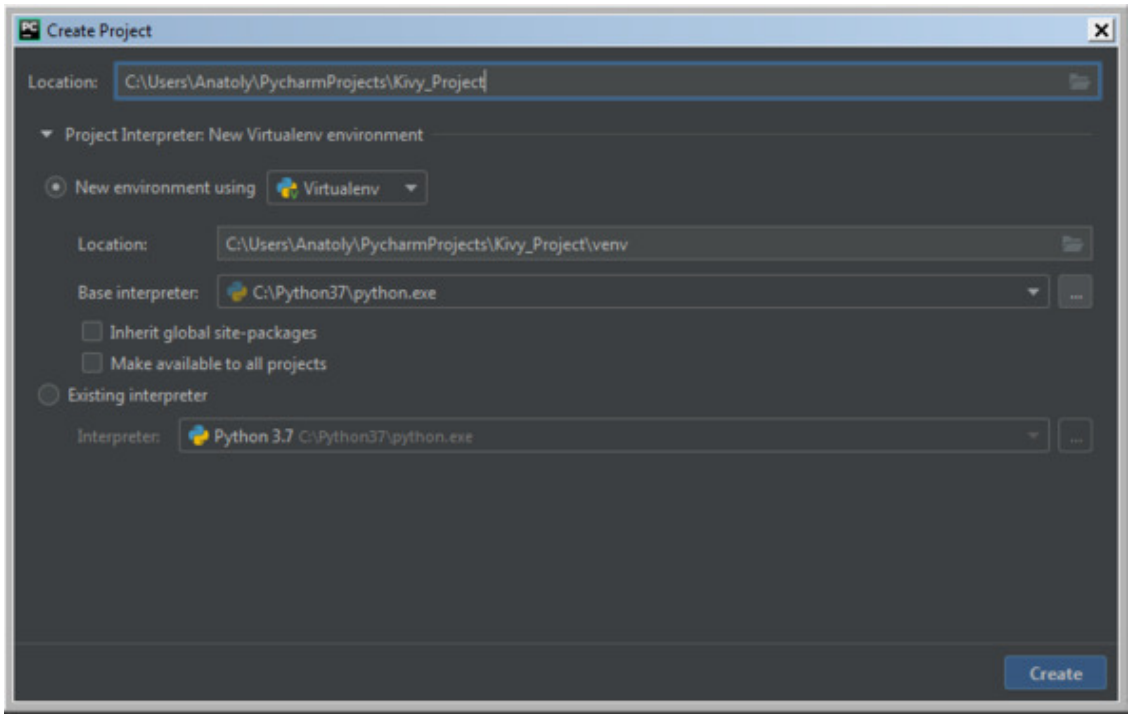


Рис. 1.27. Задаем новому проекту имя Kivy_Project в среде разработки PyCharm

Будет создан новый проект. Это, по сути дела, шаблон проекта, в котором пока еще ничего нет (рис. 1.28).

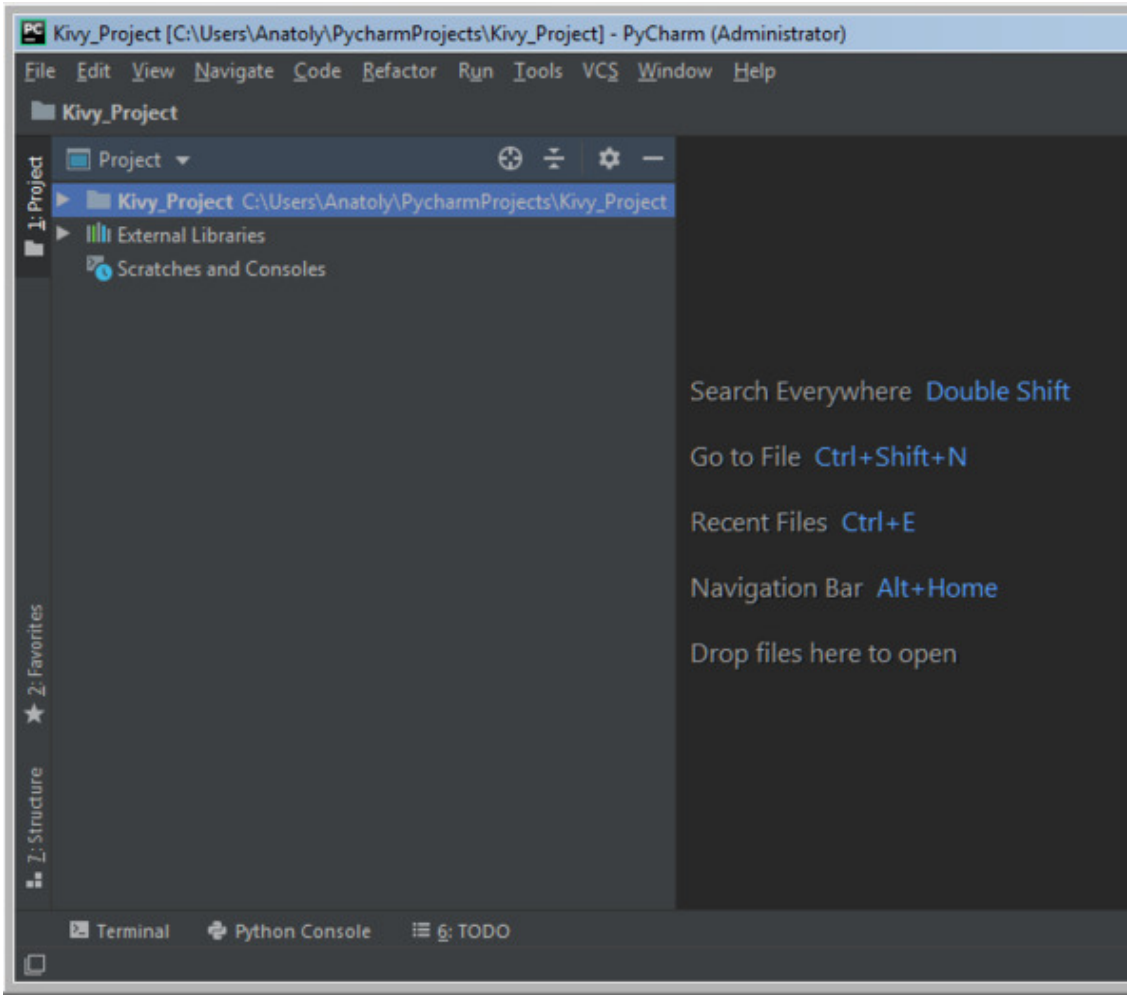


Рис. 1.28. Интерфейс PyCharm с окном пустого проекта

Теперь в виртуальное окружение созданного проекта нужно добавить фреймворк Kivy – это фактически дополнительная библиотека к Python, и установить этот инструментарий можно так же, как и любую другую библиотеку. Подключение данной библиотеки к проекту можно выполнить двумя способами: через меню PyCharm, или с использованием менеджера пакетов pip в терминальном окне PyCharm.

Для установки библиотеки Kivy первым способом нужно в меню **File** выбрать опцию **Settings** (рис. 1.29).

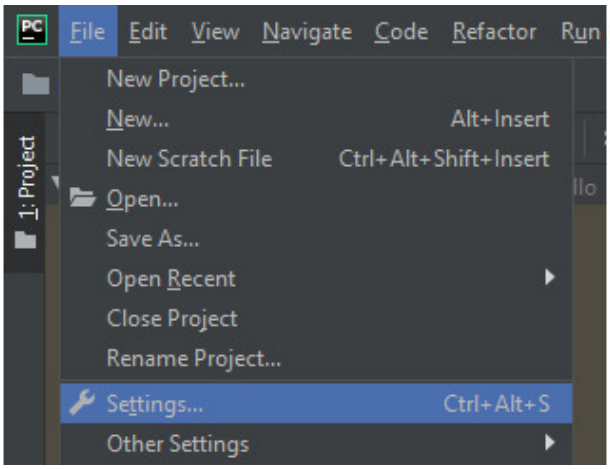


Рис. 1.29. Вызов окна **Settings** настройки параметров проекта

В левой части открывшегося окна настроек выберите опцию **Project Interpreter**, при этом в правой части окна будет показана информация об интерпретаторе языка Python и подключенных к нему библиотеках (рис. 1.30).

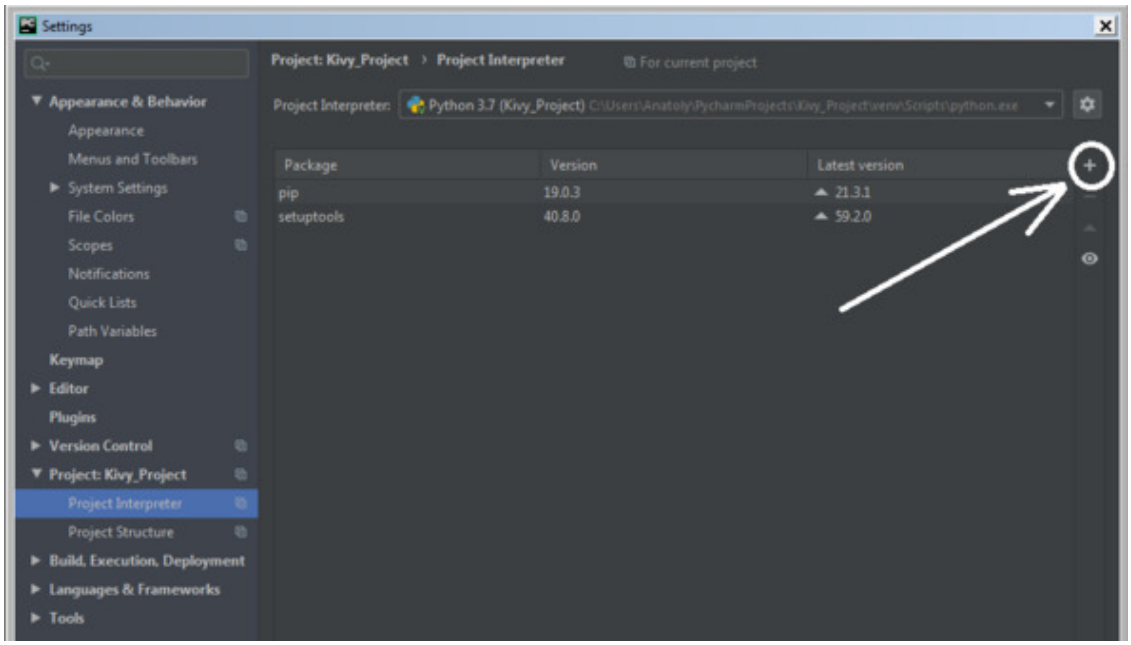


Рис. 1.30. Информация об интерпретаторе языка Python

Чтобы добавить новую библиотеку, нужно нажать на значок "+" в правой части окна, после чего будет отображен полный список доступных библиотек. Здесь можно либо пролистать весь список и найти библиотеку Kivy, либо набрать наименование этой библиотеки в верхней строке поиска, и она будет найдена в списке (рис. 1.31).

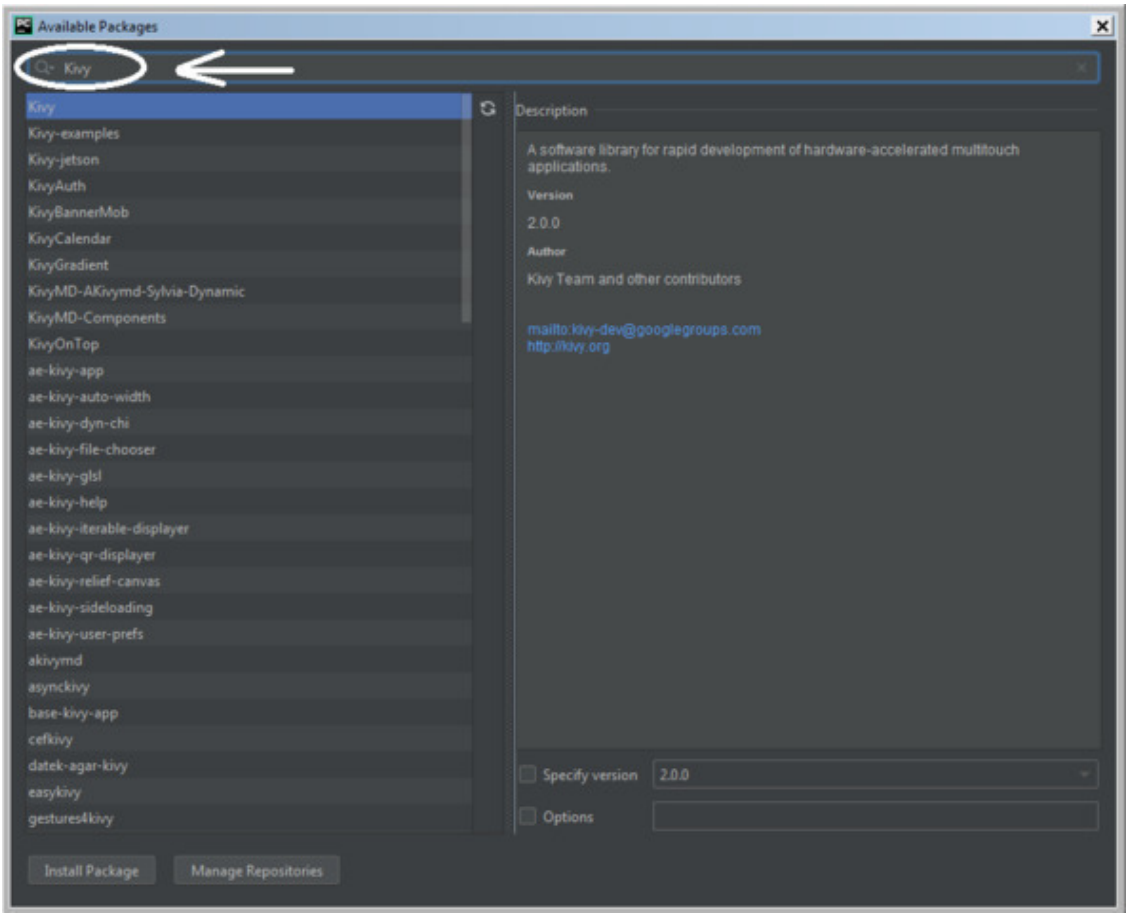


Рис. 1.31. Поиск библиотеки Kivy в списке доступных библиотек

Нажмите на кнопку **Install Package**, после этого выбранная библиотека и сопровождающие ее модули будут добавлены в ваш проект (рис. 1.32).

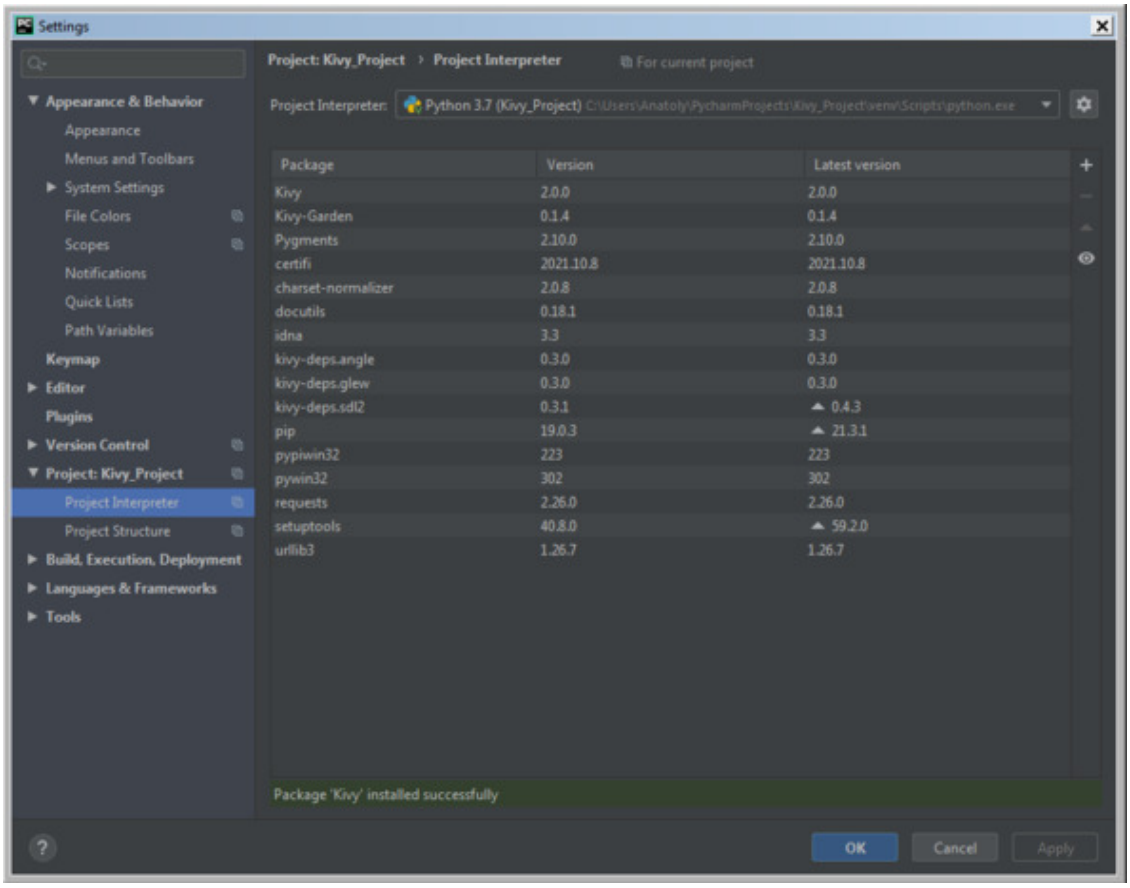


Рис. 1.32. Библиотека Kivy добавлена в список подключенных библиотек

Аналогичные действия выполним с библиотекой KivyMD. Чтобы добавить эту библиотеку, нужно нажать на значок "+" в правой части окна, после чего будет отображен полный список доступных библиотек. Здесь можно либо пролистать весь список и найти библиотеку kivymd, либо набрать наименование этой библиотеки в верхней строке поиска, и она будет найдена в списке (рис. 1.33).

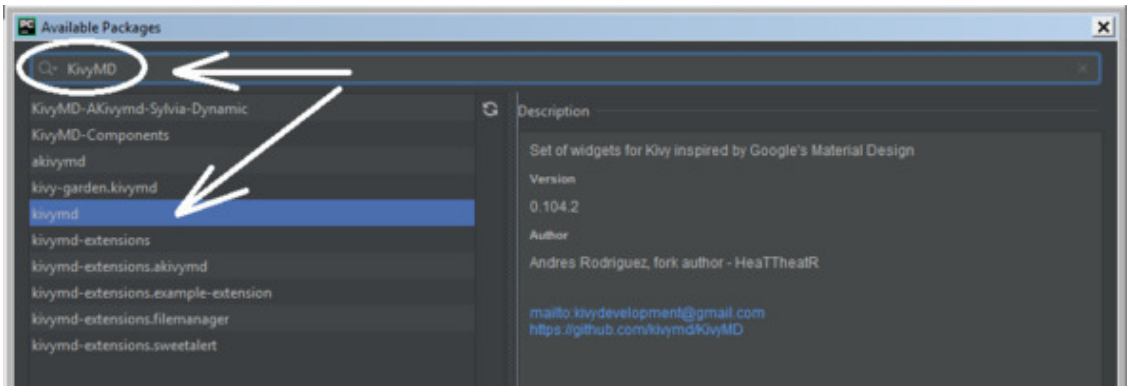


Рис. 1.33. Поиск библиотеки KivyMD в списке доступных библиотек

Нажмите на кнопку **Install Package**, после этого выбранная библиотека будет добавлена в ваш проект (рис. 1.34).

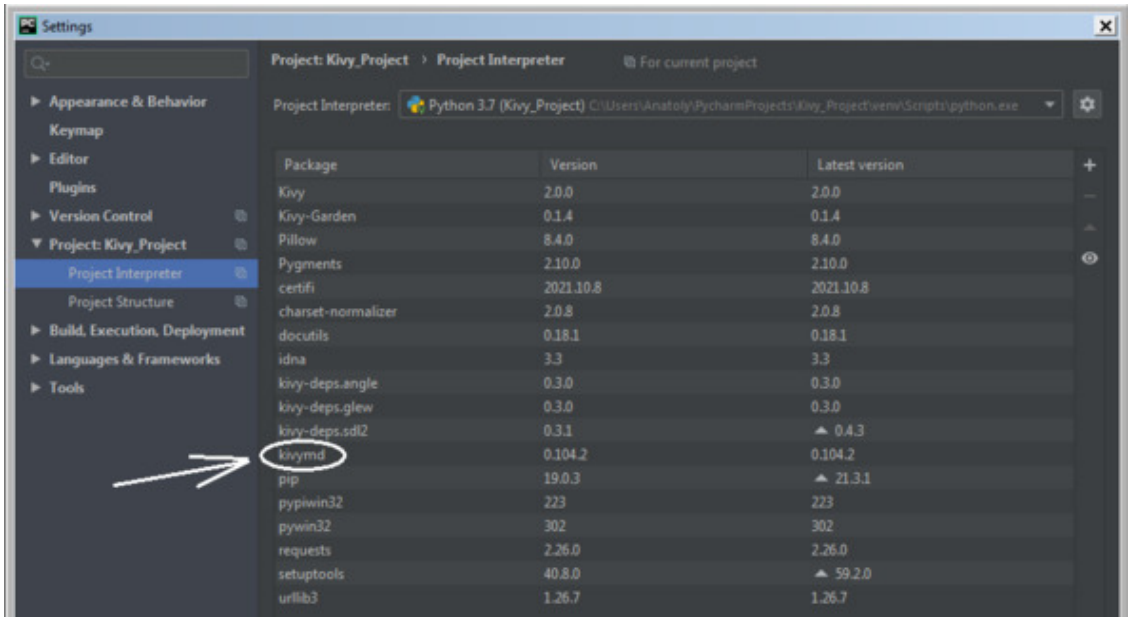


Рис. 1.34. Библиотека KivyMD добавлена в список подключенных библиотек

Для установки вышеназванных пакетов вторым способом (с использование диспетчера пакетов pip) достаточно войти в окно терминала среды PyCharm. Для подключения пакета Kivy, набрать команду- pip install kivy (рис.1.35), и нажать клавишу Enter.

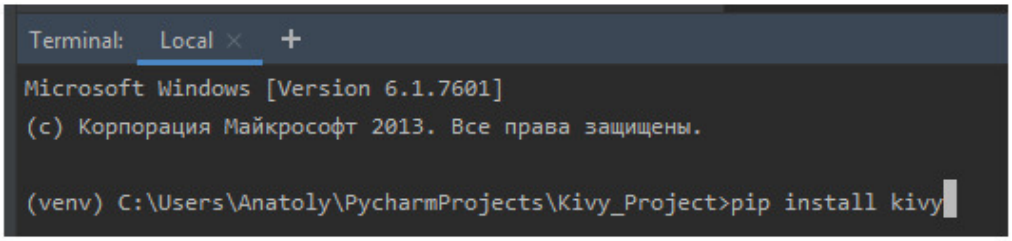


Рис. 1.35. Добавление библиотек Kivy в список подключенных библиотек в окне терминала PyCharm

Аналогично, для подключения пакета KivyMD в окне терминала среды PyCharm нужно набрать команду – pip install kivymd (рис.1.36), и нажать клавишу Enter.

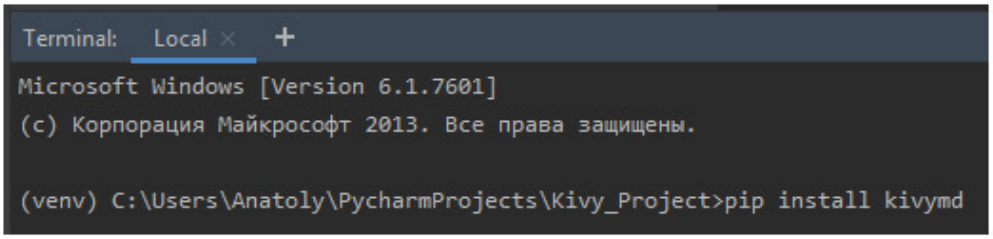


Рис. 1.36. Добавление библиотек KivyMD в список подключенных библиотек в окне терминала PyCharm

После этих действий все необходимые компоненты будут подключены к созданному проекту.

Примечание.

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 вы можете 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.

Теперь у нас есть минимальный набор инструментальных средств, который необходим для разработки мобильных приложений на языке Python. Впрочем, в процессе рассмотрения конкретных примеров нам понадобится загрузка еще ряда дополнительных пакетов и библиотек. Их описание и процедуры подключения будут представлены в последующих главах.

1.6. Первые приложения на Kivy и KivyMD

Начнем изучение Kivy с написания простейшего приложения, состоящего всего из пяти строчек программного кода. В предыдущем разделе мы создали проект с именем Kivy_Project, теперь в этом проекте создадим новый Python файл с именем First_App. Для этого в созданном нами проекте кликнем правой кнопкой мыши на имени проекта и в появившемся меню выберем опции: New-> Python File (рис.1.37).

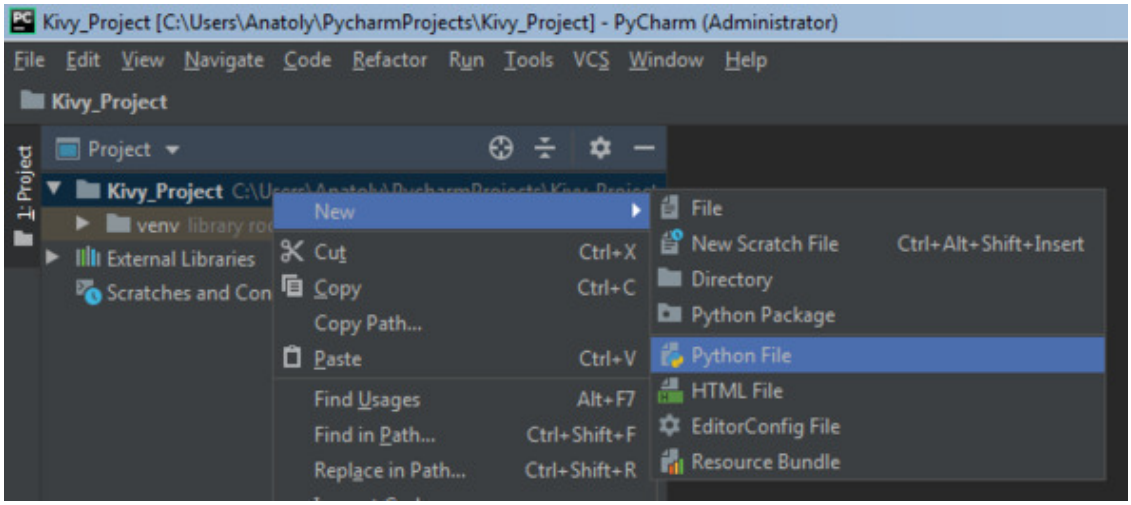


Рис. 1.37. Создание Python файла в среде PyCharm

В появившемся окне зададим имя новому файлу – First_App (рис.1.38)

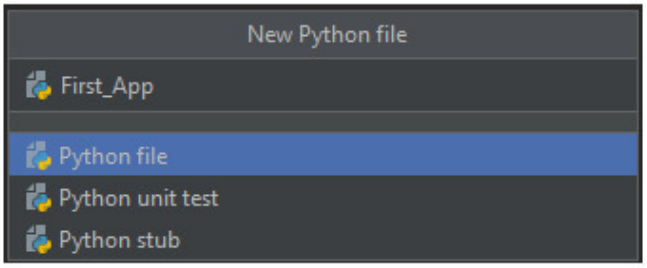


Рис. 1.38. Задание имени Python файлу, создаваемому в среде PyCharm

После того, как будет нажата клавиша Enter, будет создан пустой файл с именем First_App.py (рис.1.39).

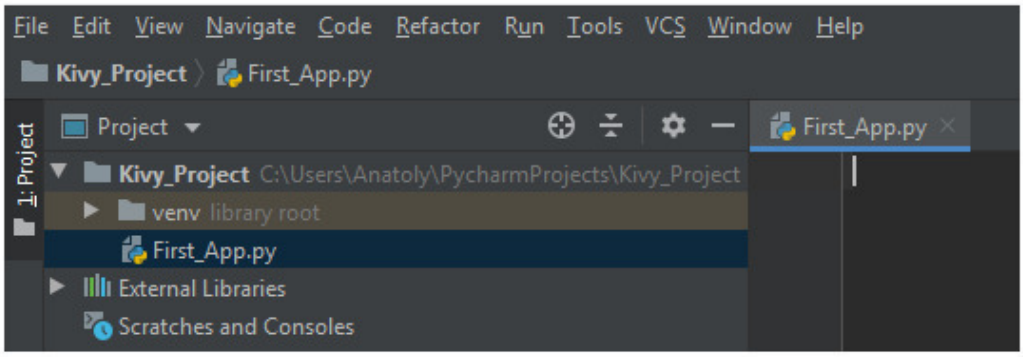


Рис. 1.39. Python файл с именем First_App.py, созданный в среде PyCharm

Теперь в открывшемся окне редактора файле First_App.py, наберем следующий программный код (листинг 1.1).

Листинг 1.1. Программный код простейшего приложения на Kivy (модуль First_App.py)

```
import kivy. app # импорт фрейморка kivy

class TestApp (kivy. app. App): # формирование базового класса
…… …… … …… …… …… … … … … приложения
……pass

app = TestApp () # создание объекта (приложения app) на основе
…… …… …… …… …… базового класса
app.run () # запуск приложения
```

В первой строке был выполнен импорт модулей, обеспечивающих работу приложений на Kivy (import kivy. app). Далее был сформирован базовый класс приложения с именем TestApp. Пока это пустой класс, внутри которого не выполняется никаких действий. В следующей строке на основе базового класса «kivy. app. App» создан объект app – по сути это и есть наше первое приложение. И, наконец, в последней строке с использованием метода run будет осуществлен запуск приложения с именем app. Вот собственно и все.

Примечание.

Одна из особенностей Python заключается в том, что для оформления блоков кода вместо привычных фигурных скобок, как в C, C ++, Java, используются отступы (или табуляция). Отступы – важная концепция языка Python и без правильного их оформления в программе будут возникать ошибки. В IDE PyCharm пробелы (отступы) формируются автоматически, поэтому программистам проще отслеживать правильность расстановки пробелов. Если вы переносите программный код примеров на свой компьютер из листингов данной книги, то внимательно проверяйте правильность расстановки отступов. В листингах программ наличие отступов условно показаны многоточием.

Можно запустить наше первое приложение, для этого кликнем павой кнопкой в окне редактора программного кода и в открывшемся меню выберем опцию Run «First_App’ (рис.1.40).

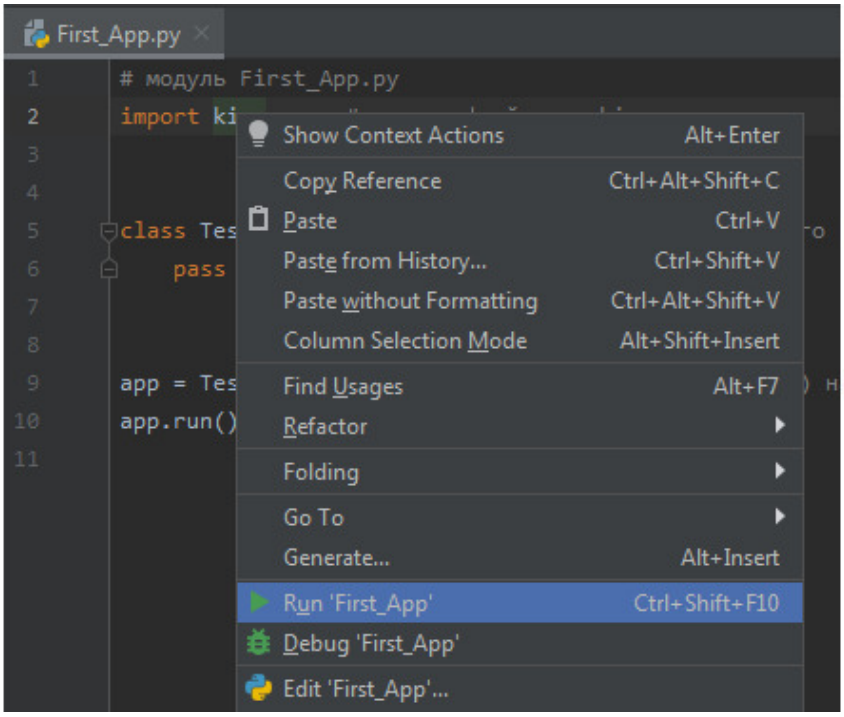


Рис. 1.40. Запуск первого приложения с именем First_App.py в среде PyCharm

Поле этих действий на экран будет выведено окно созданного приложения (рис.1.41).

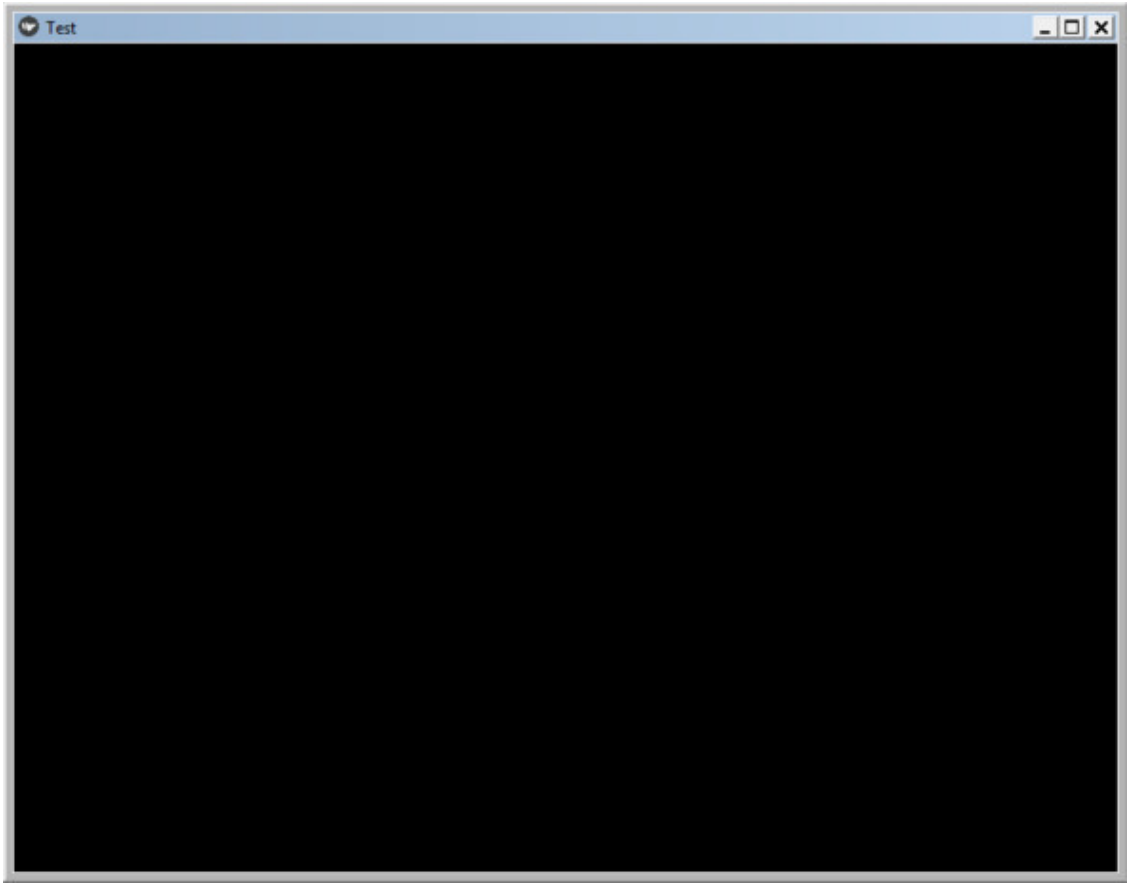


Рис. 1.41. Окно первого приложения с именем First_App.py, работающего на персональном компьютере

На экране появится пустое черное окно с титульной строкой в верхней части. В титульной строке будет отображена иконка с логотипом Kivy, имя нашего приложения (Test) и три стандартные кнопки (свернуть приложение, развернуть приложение, закрыть приложение). Поскольку мы для приложения не задавали никаких параметров, то все их значения устанавливаются по умолчанию: размер окна, цвет экрана (черный), имя приложения формируется на основе имени базового класса без символов App, то есть от имени базового класса остаются только символы Test. Поскольку в базовом классе не было задано никаких визуальных элементов, то окно приложения является пустым.

Размеры окна по умолчанию, будут зависеть от устройства, на котором запускается приложение. На вышеприведенном рисунке изображены пропорции окна приложения, запущенного на компьютерах, работающих под операционными системами Windows и Linux. Если это же приложение будет запущено на мобильном устройстве (например, на смартфоне), то пропорции экрана будут иными, то есть соответствовать размеру экрана мобильного устройства.

Вопрос о том, как загрузить данное приложение на смартфоне будет освещен в последней главе, а пока поэкспериментируем с простейшими приложениями на Kivy и KivyMD.

Напишем приложение, в базовом классе которого будут выполняться простейшие действия, например, выведено сообщение – «Привет от Kivy». Создадим новый Python файл и напишем в нем следующий код (листинг 1.2).

Листинг 1.2. Программный код приложения «Привет от Kivy» (модуль First_App_Kivy.py)

```
# модуль First_App_Kivy.py
```

```
import kivy. app # импорт фрейморка kivy
import kivy.uix.label # импорт визуального элемента label (метка)

class MainApp (kivy. app. App): # формирование базового класса
…… …… …… …… …… …… …… …… …… …… …… …… приложения
…… def build (self): # формирование функции в базовом классе
…… … … return kivy.uix.label.Label (text=«Привет от Kivy!»)

app = MainApp (title=«Первое приложение на Kivy») #Задание имени
…… … … … … … … … …… … … … … … … … … … … … … … … … …приложения
app.run () # запуск приложения
```

В данном приложении импортируются уже два модуля: приложение (import kivy. app), и элемент пользовательского интерфейса label – метка (import kivy.uix.label). Далее в базовом классе приложения (MainApp) определяем функцию, называемую build. В этой функции размещаются виджеты (элементы графического интерфейса), которые появятся на экране при запуске приложения. В нашем примере мы задали виджет – метка на основе модуля kivy.uix.label с использованием класса Label, и свойству метки (text), присвоили значение «Привет от Kivy».

В следующей строке на основе базового класса создан объект app – наше приложение, и этому приложению задали свое имя (title=«Первое приложение на Kivy»). И, наконец, в последней строке с использованием метода run будет осуществлен запуск приложения с именем app.

Создадим точно такое же простейшее приложение с использованием библиотеки KivyMD (листинг 1.3).

Листинг 1.3. Программный код приложения «Привет от KivyMD» (модуль First_App_Kivy_MD.py)

```
# модуль First_App_Kivy_MD.py
from kivymd. app import MDApp
from kivymd.uix.label import MDLabel

class MainApp (MDApp):
……def build (self):
…… … … return MDLabel (text=«Привет от KivyMD!», halign=«center»)

app = MainApp (title=«Первое приложение на KivyMD»)
app.run ()
```

Этот программный код по свое структуре практически не отличается от предыдущего кода. Разница лишь в том, что мы импортировали модули от библиотеки KivyMD (первые две строки), и в строке инициализации метки задали ей положение в центре экрана (halign=«center»). В библиотеке KivyMD по умолчанию она была бы прижата к левой части экрана.

Если теперь запустить эти два приложения на выполнение, то мы получим следующие сообщения (рис.1.42).

Рис. 1.42. Окна первого приложения на Kivy и KivyMD с визуальным элементом

Из вышеприведенного рисунка видно, что эти приложения отличаются только цветом фона окна (для Kivy он по умолчанию черный, для KivyMD – белый).

Следует отметить еще одну особенность, любой визуальный элемент занимает все пространство окна и, если не заданы параметры его размещения, то для Kivy виджет располагается в центре экрана, для KivyMD – прижимается к левому краю экрана. В титульной строке приложения мы видим логотип Kivy. Этот логотип задается по умолчанию и всегда присутствует в приложениях, которые запускаются на настольных компьютерах под Windows, но он будет отсутствовать в приложениях, запускаемых на Linux и на смартфонах. Однако не зависимо от платформы этот логотип будет отображен на значке запуска приложения. При этом разработчик может сопоставить разработанное приложение с любым своим логотипом.

Внесем изменения в приведенные выше программные коды, определив для приложений собственный логотип, и сделав код более привычным для программистов. Модифицированный программный код приложения на Kivy с указанием собственного логотипа в файле pyt. ico, приведен в листинге 1.4.

Листинг 1.4. Модифицированный программный код приложения на Kivy (модуль First_App_Kivy2)

```
# модуль First_App_Kivy2.py
from kivy. app import App # импорт приложения фрейморка kivy
from kivy.uix.label import Label # импорт элемента label (метка)

class MainApp (App): # формирование базового класса приложения
…… def build (self): # формирование функции в базовом классе
```

```
…… … … self. title = «Приложение на Kivy’ # Имя приложения
…… … … self. icon =». /pyt. ico’ # иконка (логотип) приложения
…… … … label = Label (text=«Привет от Kivy и Python!») # метка
…… … ….return label # возврат значения метки

if __name__ == '__main__': # условие вызова приложения
…… app = MainApp () # Задание приложения
…… app.run () # запуск приложения
```

Первые две строчки данного программного кода не изменились – импортируются два модуля: приложение (import kivy. app) и элемент пользовательского интерфейса label – метка (import kivy.uix.label). Далее в базовом классе приложения (MainApp) определяем функцию с именем build. В данной функции мы определяет имя для нашего приложения – self. title (то, что будет отображаться в титульной строке приложения) и задаем собственную иконку. Для данного примера была взята иконка в виде логотипа Python – файл pyt. ico, который поместили в корневой каталог проекта. Задание собственной иконки для приложения выполнили с помощью строки программного кода – self. icon =». /pyt. ico’. В следующей строке программы создали метку и присвоили ей значение «Привет от Kivy и Python», а команда return вернет это значение приложению. Последние три строчки уже знакомы пользователям Python:

– определяем условие вызова приложения (if __name__);

– определяем само приложение с указанием заголовка главного окна (app = MainApp (title=«Первое приложение»);

– запускаем приложение на исполнение – app.run ().

Аналогичные изменения сделаем и для программного кода приложения на KivyMD (листинг 1.5).

Листинг 1.5. Модифицированный программный код приложения на KivyMD (модуль First_App_KivyMD2.py)

```
# модуль First_App_KivyMD2.py
from kivymd. app import MDApp
from kivymd.uix.label import MDLabel

class MainApp (MDApp):
…….def build (self):
…… … … self. icon = 'icon.png’
…… … … self. title = «Приложение на KivyMD»
…… … …label = MDLabel (text=«Привет от KivyMD и Python»,
…… …… …… …… …… …… halign=«center»)
…… … … return label

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

Здесь в качестве логотипа использовано изображение из файла – icon.png.

После запуска этих двух программ получим следующий результат (рис.1.43).

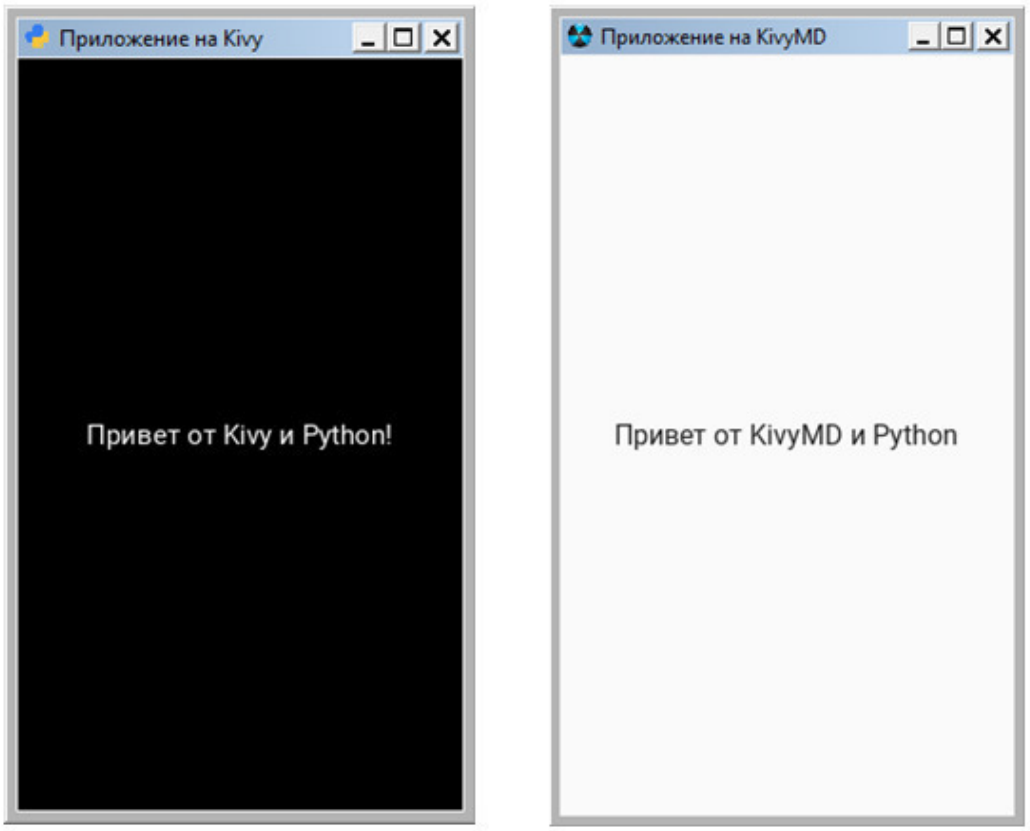


Рис. 1.43. Окна приложений на Kivy и KivyMD с собственным логотипом

Как видно из данного рисунка, в титульной строке окна приложения появились пользовательские иконки и название приложения.

Краткие итоги

В этой главе мы познакомились с основными инструментальными средствами, с помощью которых можно разрабатывать кроссплатформенные приложения на языке программирования Python, как для настольных компьютеров, так и для мобильных устройств. Это интерпретатор Python, интерактивная среда разработки программного кода PyCharm, фреймворк Kivy и библиотека KivyMD. Установив на свой компьютер эти инструментальные средства уже можно приступать к написанию программного кода, что мы и сделали, написав несколько простейших программ.

Теперь можно перейти к следующей главе и более детально познакомиться с фреймворком Kivy, с особенностями встроенного языка KV, а также с основными виджетами, которые используются для создания пользовательского интерфейса.

Глава 2. Фреймоворк Kivy, язык KV и виджеты, как основа пользовательского интерфейса

В этой главе мы рассмотрим вопросы, связанные с особенностями приложений, написанных с использованием фреймворка Kivy. Познакомимся с языком KV, и с виджетами – контейнерами, которые обеспечивают позиционирование элементов интерфейса на экране. В частности, будут рассмотрены следующие материалы:

– особенности фреймворка Kivy и общие представления о дереве виджетов;

– базовые понятия о синтаксисе языка KV и возможности создания пользовательских классов и объектов;

– возможности разделения приложений на логически и функционально связанные блоки;

– понятия о свойствах и параметрах виджетов;

– описание виджетов, используемых для позиционирования видимых элементов интерфейса.

Итак, приступим к знакомству с основами работы с фрейморком Kivy.

2.1. Общее представление о фрейморке Kivy

Фреймворк Kivy – это кроссплатформенная бесплатная библиотека Python с открытым исходным кодом. С ее использованием можно создавать приложения для любых устройств (настольные компьютеры, планшеты, смартфоны). Данный фреймворк заточен на работу с сенсорными экранами, однако приложения на Kivy с таким же успехом работают и на обычных мониторах. Причем даже на устройстве с обычным монитором приложение будет вести себя так, как будто оно имеет сенсорный экран. Kivy работает практически на всех платформах: Windows, OS X, Linux, Android, iOS, Raspberry Pi.

Этот фрейморк распространяется под лицензией MIT (лицензия открытого и свободного программного обеспечения) и на 100% бесплатен для использования. Фреймворк Kivy стабилен и имеет хорошо документированный API. Графический движок построен на основе OpenGL ES2.

Примечание.

> OpenGL ES2 – подмножество графического интерфейса, разработанного специально для встраиваемых систем (мобильные телефоны, мини компьютеры, игровые консоли).

В набор инструментов входит более 20 виджетов, и все они легко расширяемы.

Примечание.

> Виджет – это небольшое приложение для компьютера или смартфона, которое обычно реализуется в виде класса и имеет набор свойств и методов. Через виджеты обеспечивается взаимодействие приложения с пользователем. Виджет может быть видимым в окне приложения, а может быть скрытым. Но даже в скрытом виджете запрограммирован определенный набор функций.

При использовании фрейморка Kivy программный код для создания элементов пользовательского интерфейса можно писать на Python, а можно для этих целей использовать специальный язык. В литературе можно встретить разное обозначение этого языка: язык kivy язык KV, KV. Далее во всех разделах этой книги он будет обозначен, как KV.

Язык KV обеспечивает решение следующих задач:

– создавать объекты на основе базовых классов Kivy.

– формировать дерево виджетов (создавать контейнеры для размещения визуальных элементов и указывать их расположение на экране);

– задавать свойства для виджетов;

– естественным образом связывать свойства виджетов друг с другом;

– связывать виджеты с функциями, в которых обрабатываются различные события.

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.

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.

У компоненты Builder есть два метода для загрузки в приложение кода на языке KV:

– Builder. load_file ('path/name_file. kv') – если код на языке KV подгружается из файла (здесь path – путь к файлу, name_file. kv – имя файла);

– Builder. load_string (kv_string) – если код на языке KV подгружается из строковой переменной (kv_string – имя строковой переменной).

2.2. Язык KV и его особенности

2.2.1. Классы и объекты

По мере того, как приложение усложняется, становится трудно поддерживать конструкцию дерева виджетов и явное объявление привязок. Чтобы преодолеть эти недостатки, альтернативой является язык KV, также известный как язык Kivy или KVlang. Язык KV позволяет создавать дерево виджетов в декларативной манере, позволяет очень быстро создавать прототипы и оперативно вносить изменения в пользовательский интерфейс. Это также помогает отделить логику приложения от пользовательского интерфейса.

Язык KV, как и Python, является объектно-ориентированным языком. Все элементы интерфейса представляют собой объекты, которые строятся на основе базовых классов. Каждый класс имеет набор свойств, зарезервированных методов и событий, которые могут быть обработаны с помощью функций. В языке KV принято соглашение: имя класса всегда начинается с заглавной буквы (например, Button – кнопка, Label – метка), а имя свойства с маленькой буквы (например, text, text_size, font_size).

Самый простой способ использования классов в KV – это употребление их оригинальных имен. Проверим это на простом примере. Создадим файл с именем K_May_Class1.py и напишем в нем следующий код (листинг 2.1).

Листинг 2.1. Пример использования базового класса (модуль K_My_Class1.py)

```
# модуль K_May_Class1.py
from kivy. app import App
from kivy.lang import Builder

KV =«»»
BoxLayout: # контейнер (базовый класс BoxLayout)
……Button: # кнопка (класс Button)
…… … … text: «Кнопка 1» # свойство кнопки (надпись)
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

Примечание.

Мы еще не знакомились с виджетами Kivy, а в этом коде используется два виджета: видимый виджет Button (кнопка), и виджет – контейнер BoxLayout (коробка). Более подробно о них будет сказано в последующих разделах. А пока будем использовать их в своих примерах. В листинге присутствуют тройные кавычки – «»», в редакторе программного кода вместо них нужно использовать тройной апостроф – «'''».

В этом коде в текстовой переменной KV создан виджет – контейнер на основе базового класса BoxLayout, в нем размещена кнопка (Button), свойству кнопки text присвоено значение «Кнопка 1» (на языке KV свойство от его значения отделяется знаком двоеточия «:»). При этом нет необходимости явно импортировать базовые классы, они загрузятся автоматически. После запуска приложения получим следующий результат (рис.2.1).

Рис. 2.1. Результаты выполнения приложения из модуля K_May_Class1.py

В этом примере мы косвенно познакомились с виджетом – контейнером BoxLayout и простейшим деревом виджетов. Здесь в виджет – контейнер была помещена кнопка. Более подробно виджеты – контейнеры будут рассмотрены в последующих разделах.

Разработчик может в коде на Python переопределить имя базового класса, то есть создать собственный пользовательский класс. Например, разработчик хочет использовать класс BoxLayout, но при этом дать ему другое имя, например, MyBox. Проверим это на простом примере. Создадим файл с именем K_May_Class2.py и напишем в нем следующий код (листинг 2.2).

Листинг 2.2. Пример использования пользовательского класса (модуль K_My_Class2.py)

```
# модуль K_May_Class2.py
from kivy. app import App
from kivy.lang import Builder
from kivy.uix.boxlayout import BoxLayout

KV = «»»
MyBox: # контейнер (пользовательский класс)
……Button: # кнопка (класс Button)
…… … …text: «Кнопка 2» # свойство кнопки (надпись на кнопке)
«»»

# 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 ()
```

В этом программном коде на языке Python создан пользовательский класс MyBox на основе базового класса BoxLayout. При этом нужно явно выполнить импорт базового класса BoxLayout:

from kivy.uix.boxlayout import BoxLayout

После запуска приложения получим следующий результат (рис.2.2).

Рис. 2.2. Результаты выполнения приложения из модуля K_May_Class2.py

Однако есть более простой способ создания пользовательского класса непосредственно в коде на языке KV. Для этого используется следующая конструкция:

<Имя_пользовательского_класса@Имя_базового_класса>

Проверим это на простом примере. Создадим файл с именем K_May_Class3.py и напишем в нем следующий код (листинг 2.3).

Листинг 2.3. Пример использования пользовательского класса (модуль K_My_Class3.py)

```
# модуль K_May_Class3.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
# пользовательский класс MyBox
# на основе базового класса BoxLayout
<MyBox@BoxLayout>

MyBox: # контейнер (пользовательский класс)
…… Button: # кнопка (класс Button)
…… … … text: «Кнопка 3» # свойство кнопки (надпись на кнопке)
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В этом программном коде пользовательский класс MyBox на основе базового класса BoxLayout создан непосредственно в коде на KV:

<MyBox@BoxLayout>

При этом не нужно явно выполнить импорт базового класса BoxLayout, и не нежно объявлять пользовательский класс в разделе программы на Python. При этом сам программный код получается компактным и более понятным.

Примечание.

> В этом случае строка, в которой сформирован пользовательский класс, должна находиться между символами <…>.

После запуска приложения получим следующий результат (рис.2.3).

Рис. 2.3. Результаты выполнения приложения из модуля K_May_Class3.py

2.2.2. Динамические классы

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:

```
BoxLayout:
…… Button:
…… … … text: «Кнопка 1»
…… … … pos_hint: {’center_x’:.5, ’center_y’:.6}
…… … … font_size: ’25sp’
…… … … markup: True
…… Button:
…… … … text: " Кнопка 2»
…… … … pos_hint: {’center_x’:.5, ’center_y’:.6}
…… … … font_size: ’25sp’
…… … … markup: True
…… Button:
…… … … text: " Кнопка 3»
…… … … pos_hint: {’center_x’:.5, ’center_y’:.6}
…… … … font_size: ’25sp’
…… … … markup: True
```

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:

```
<MyButton@Button>:
…… pos_hint: {’center_x’:.5, ’center_y’:.6}
…… font_size: ’25sp’
…… markup: True

BoxLayout:
…… MyButton:
…… … … text: " Кнопка 1»
…… MyButton:
…… … … text: " Кнопка 2»
…… MyButton:
…… … … text: " Кнопка 3»
```

Не вдаваясь в смысл этих свойств, проверим это на простом примере. Создадим файл с именем K_May_Class4.py и напишем в нем следующий код (листинг 2.4).

Листинг 2.4. Пример использования динамического класса (модуль K_My_Class4.py)

```
# модуль K_May_Class4.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
<MyButton@Button>:
```

```
…… font_size: ’25sp’
…… pos_hint: {’center_x’:.5, ’center_y’:.6}
…… markup: True

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 ()
```

В этом программном коде создан динамический класс MyButton на основе базового класса Button. Для класса MyButton один раз заданы три 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. После запуска приложения получим следующий результат (рис.2.4).

Рис. 2.4. Результаты выполнения приложения из модуля K_May_Class3.py

2.2.3. Зарезервированные слова и выражения в языке KV

В языке KV существует специальный синтаксис для задания значений переменным и свойствам. На Python для присвоения значений переменным используется знак «=», то есть применяется такая конструкция: name = value. На языке KV для задания значений свойствам виджетов используется знак двоеточия «:», например, name: value. В предыдущих примерах мы уже неоднократно встречались с такой конструкцией, например:

```
Button:
…… text: «Кнопка 1»
```

На Python импорт (подключение) внешних модулей выглядит следующим образом:

```
import numpy as np
```

На языке KV этот код будет выглядеть так:

```
#: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;

– args – (аргументы) позволяет указать аргументы при обращении к функциям;

– ids – (идентификаторы) позволяет обратиться к параметрам виджета через его идентификатор.

Ключевое слово self. Ключевое слово self ссылается на «текущий экземпляр виджета». С его помощью можно, например, получить значения свойств текущего виджета. Рассмотрим это на простейшем примере. Создадим файл с именем Button_Self.py и напишем в нем следующий код (листинг 2.5).

Листинг 2.5. Демонстрация использования ключевого слова self (модуль Button_Self.py)

```
# Модуль Button_Stlf.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Button
…… text: «Состояние кнопки – %s»% self.state
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

Обычно свойству кнопки text присваивается значение типа: «Кнопка», «Подтвердить», «OK», «Да», «Нет» и т. п. Здесь же свойству кнопки text, через префикс self присвоено значение ее же свойства – состояние кнопки (self.state). Получается, что кнопка сделала запрос сама к себе. После запуска приложения получим следующий результат (рис.2.5).

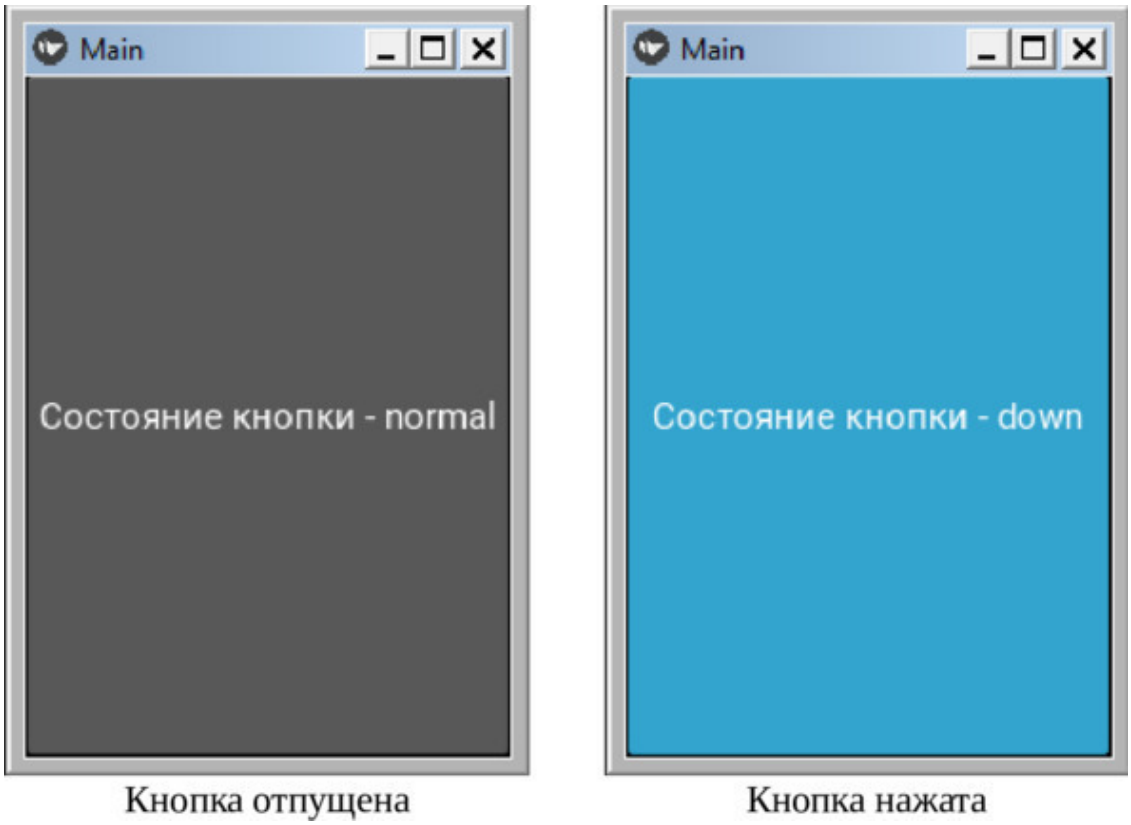


Рис. 2.5. Результаты выполнения приложения из модуля Button_State.py

Как видно из данного рисунка, после изменения состояния кнопка от себя получила значение своего же свойства.

Ключевое слово root. Ключевое слово root (корень) позволяет получить ссылку на параметры корневого виджета. Рассмотрим это на простейшем примере. Создадим файл с именем Button_Root.py и напишем в нем следующий код (листинг 2.6).

Листинг 2.6. Демонстрация использования ключевого слова root (модуль Button_Root.py)

```
# Модуль Button_Root.py
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 ()
```

Здесь создан корневой виджет BoxLayout и его свойству orientation задано значение – «vertical». Затем в корневой виджет вложен элемент Button (кнопка). Свойству кнопки text, через префикс root присвоено значение свойства корневого виджета – orientation (root. orientation). Получается, что кнопка сделала запрос к свойству корневого виджета. После запуска приложения получим следующий результат (рис.2.6).

Рис. 2.6. Результаты выполнения приложения из модуля Button_Root.py

Как видно из данного рисунка, на кнопке отобразился текст, который соответствует значению свойства корневого виджета.

Ключевое слово app. Это ключевое слово позволяет обратиться к элементу, который относится к приложению. Это эквивалентно вызову функции, которая находится в коде приложения, написанного на Python. Рассмотрим это на простейшем примере. Создадим файл с именем Button_App.py и напишем в нем следующий код (листинг 2.7).

Листинг 2.7. Демонстрация использования ключевого слова app (модуль Button_App.py)

```
# Модуль Button_App.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
BoxLayout:
…… orientation: ’vertical’
…… Button:
…… … … text: «Кнопка 1»
…… … … on_press: app.press_button (self. text)
…… Label:
…… … … text: app.name
«»»
```

```
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 ()
```

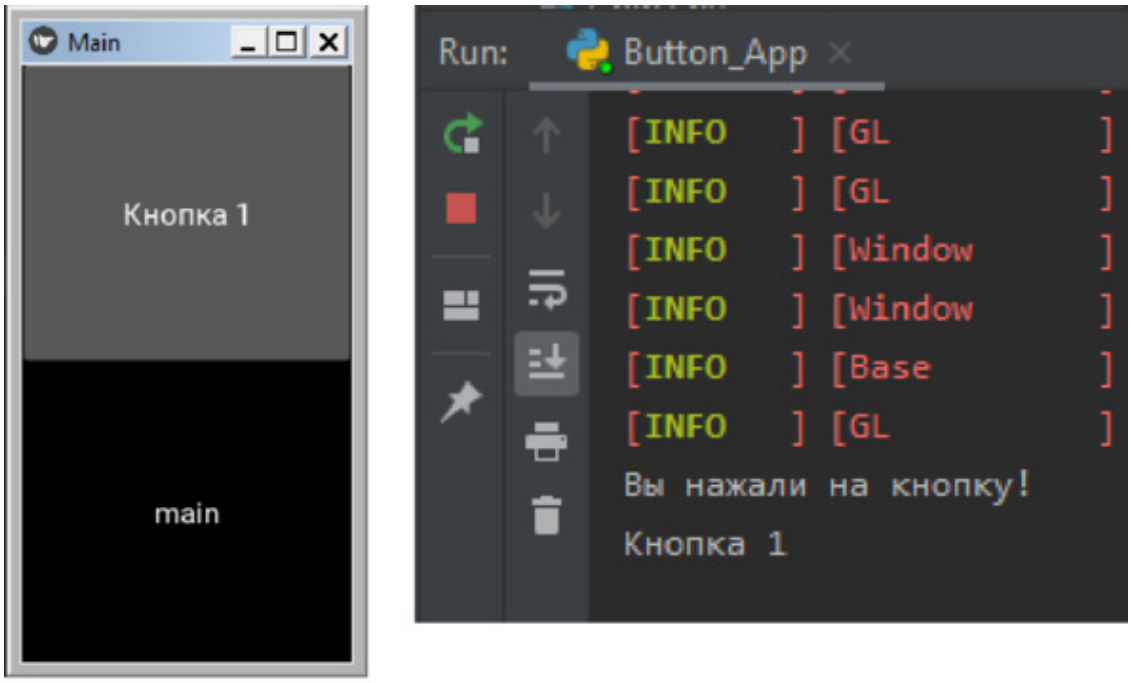
Примечание.

В этом модуле используется виджет BoxLayout. Более подробно с особенностями этого виджета можно ознакомиться в соответствующем разделе книги.

В этом модуле создан корневой виджет BoxLayout. Затем в корневой виджет вложено два элемента – Button (кнопка) и Label (метка). Событие нажатия кнопки (on_press), будет обработано функцией press_button. Эта функция находится в приложении Main, поэтому перед именем функции стоит префикс app – app.press_button (self. text). То есть в данной строке указано, что мы через префикс app обращаемся к приложению Main, в частности к функции press_button, и передаем в эту функцию свойство text данной кнопки (self. text).

Метка Label имеет свойство text. Этому свойству через префикс app присваивается имя приложения (Main).

Получается, что с использованием префикса app кнопка обратилась к функции приложения и передала ему свое свойство, а метка Label получила значение своего свойства из приложения Main. После запуска данного модуля получим следующий результат (рис.2.7).



Окно приложения — Результаты работы функции press_button

Рис. 2.7. Результаты выполнения приложения из модуля Button_App.py

Как видно из данного рисунка, метка Label показала имя приложения (main), а функция обработки события нажатия на кнопку выдала свойство text этой кнопки – «Кнопка 1».

Ключевое слово args. Это ключевое слово используется при обращении к функциям обратно вызова для передачи им аргументов. Это относится к аргументам, переданным обратному вызову. Рассмотрим это на простейшем примере. Создадим файл с именем Button_Args.py и напишем в нем следующий код (листинг 2.8).

Листинг 2.8. Демонстрация использования ключевого слова args (модуль Button_Args.py)

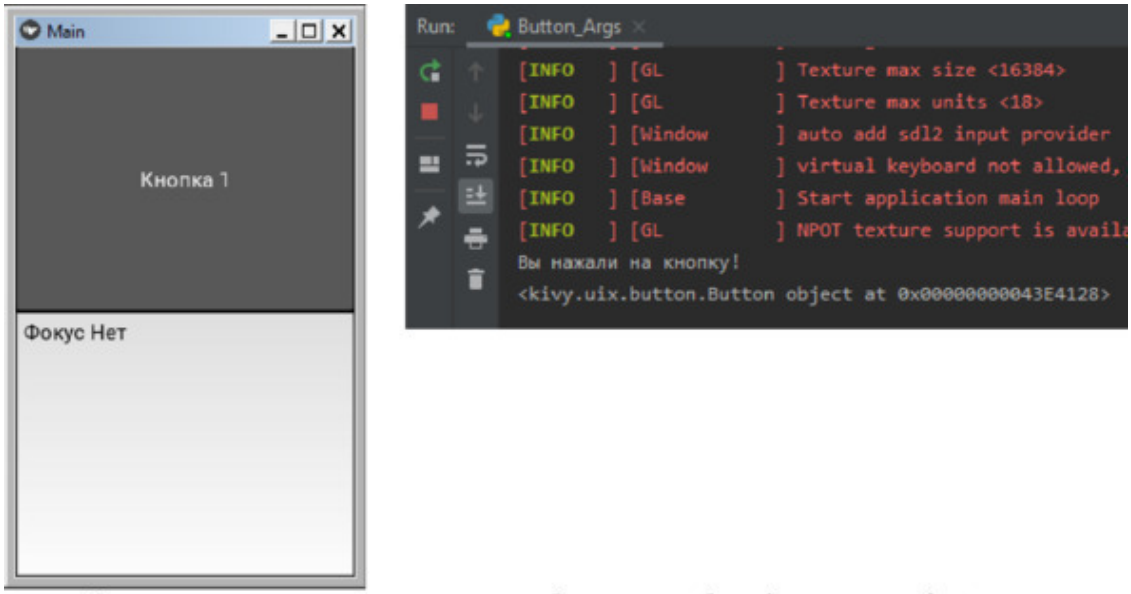
```
# Модуль Button_Args.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
BoxLayout:
…… orientation: ’vertical’
…… Button:
…… … … text: «Кнопка 1»
…… … … on_press: app.press_button (*args)
…… TextInput:
…… … … on_focus: self.insert_text («Фокус» if args [1] else «Нет»)
«»»

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 ()
```

В этом модуле создан корневой виджет BoxLayout. Затем в корневой виджет вложено два элемента – Button (кнопка) и TextInput (поле для ввода текста). Событие нажатия кнопки (on_press), будет обработано функцией press_button (*args). В скобках указаны аргументы, которые будут переданы в данную функцию (звездочка * говорит о том, что будут переданы все аргументы от текущего виджета).

У виджета TextInput определено событие получения фокуса (on_focus). Для обработки этого события будет использоваться функция insert_text (вставить текст):

self.insert_text («Фокус" if args [1] else «Нет»)

Какой текст будет вставлен, зависит от значения args [1]. Если тестовое поле получит фокус, то в поле ввода будет вставлен текст «Фокус», если поле для ввода текста потеряет фокус, то будет вставлено слово «Нет». После запуска данного модуля получим следующий результат (рис.2.8).



Окно приложения Результаты работы функции press_button

Рис. 2.8. Результаты выполнения приложения из модуля Button_Args.py

Как видно из данного рисунка, в поле для ввода текста TextInput показаны результаты обработки события получения и потери фокуса, а в окне терминал показаны результаты обработки события нажатия на кнопку. В обоих случаях для передачи аргументов использовалось ключевое слово args.

Ключевое слово ids. Ключевые слова ids (идентификаторы) и id (идентификатор) используются для идентификации виджетов. 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.

Рассмотрим следующий код:

```
Button:
…… id: but1
…… text: «Кнопка 1»
Label:
…… text: but1.text
```

В этом коде создано два элемента интерфейса: виджет Button (кнопка), и виджет Label (метка). Кнопке через ключевое слово id присвоено уникальное имя – but1, через которое теперь можно обращаться к свойствам данного элемента. Свойству text метки Label присвоено значение «but1.text». То есть метка обратилась к кнопке bat1 и получила от кнопки значение его свойства text. В итоге метка покажет на экране текст «Кнопка 1».

Рассмотрим это на простейшем примере. Создадим файл с именем Button_Id.py и напишем в нем следующий код (листинг 2.9).

Листинг 2.9. Демонстрация использования ключевого слова id (модуль Button_Id.py)

```
# Модуль Button_Id.py
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
…… Button:
…… … … id: bt2
…… … … text: «Кнопка 2»
…… … … on_press: lb1.text = bt2.text
…… Label:
…… … … id: lb1
…… … … text: «Метка»
…… … … on_touch_down: self. text = «Метка»
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В этом модуле создан корневой виджет BoxLayout. Затем в корневой виджет вложено три элемента: две кнопки Button, и метка Label. Кнопки имеют идентификаторы «bt1» и «bt2», а метка идентификатор «lb1». При касании кнопки bt1 (событие on_press) свойству метки text будет присвоено значение аналогичного свойства кнопки bt2, что запрограммировано в выражении:

```
on_press: lb1.text = bt1.text xt
```

При касании кнопки bt2 (событие on_press) свойству метки text будет присвоено значение аналогичного свойства кнопки bt2, что запрограммировано в выражении:

```
on_press: lb1.text = bt2.text
```

При касании метки lb1 (событие on_touch_down) свойству метки text будет присвоено значение «Метка», что запрограммировано в выражении:

```
on_touch_down: self. text = «Метка»
```

В итоге после касания всех элементов содержание метки будет меняться. После запуска приложения получим следующий результат (рис.2.9).

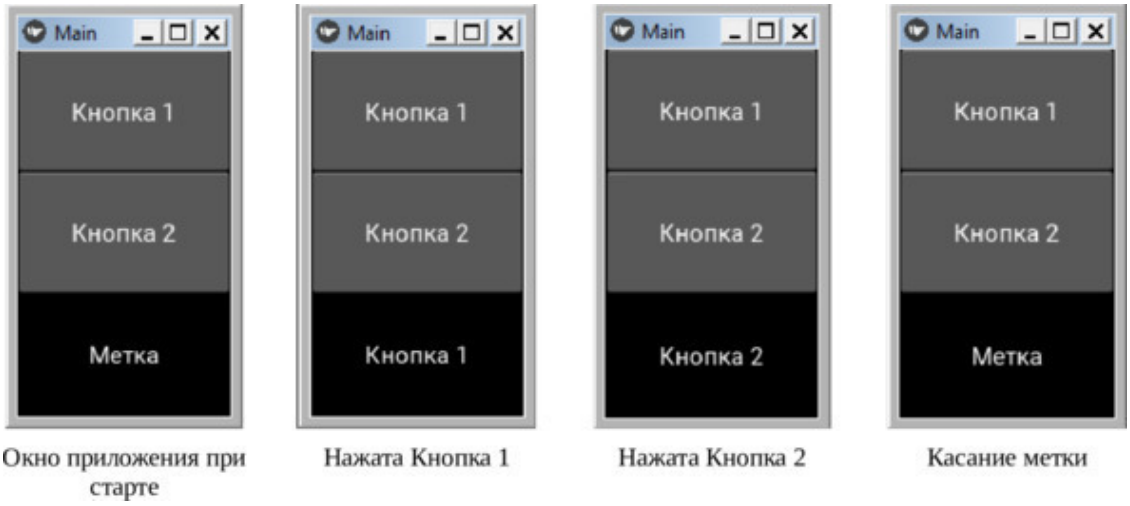


Рис. 2.9. Результаты выполнения приложения из модуля Button_Id.py

Как видно из данного рисунка, после касания элементов интерфейса меняется текст у метки Label, то есть сработала ссылка одного виджета на другой через их идентификаторы.

С использованием ключевого слова ids можно из кода на Python обратиться к виджету, который создан в разделе программы в коде на KV. Рассмотрим это на простейшем примере. Создадим файл с именем Button_Ids.py и напишем в нем следующий код (листинг 2.10).

Листинг 2.10. Демонстрация использования ключевого слова ids (модуль Button_Ids.py)

```
# Модуль Button_Ids.py
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 ()
```

Здесь во фрагменте модуля, написанного на Python, создан пользовательский класс box на основе базового класса BoxLayout. Это по своей сути контейнер, в который на языке KV вложено два элемента: кнопка Button и метка Label, которая имеет имя (идентификатор) «itog».

При касании кнопки возникает событие (on_press). Это событие обрабатывается в корневом виджете root, в функции result, куда передается текст «Нажата кнопка». В функции def result этот текст принимается в параметр entry_text. А вот в следующей строке как раз происходит использование ключевого слова ids:

self.ids [«itog»].text = entry_text

Эта строка говорит о том, что свойству text элемент корневого виджета с именем (идентификатором) [«itog»] нужно присвоить значение параметра entry_text. Поскольку данному параметру будет передано значение «Нажата кнопка», то этот текст отобразится на экране. После запуска приложения получим следующий результат (рис.2.10).

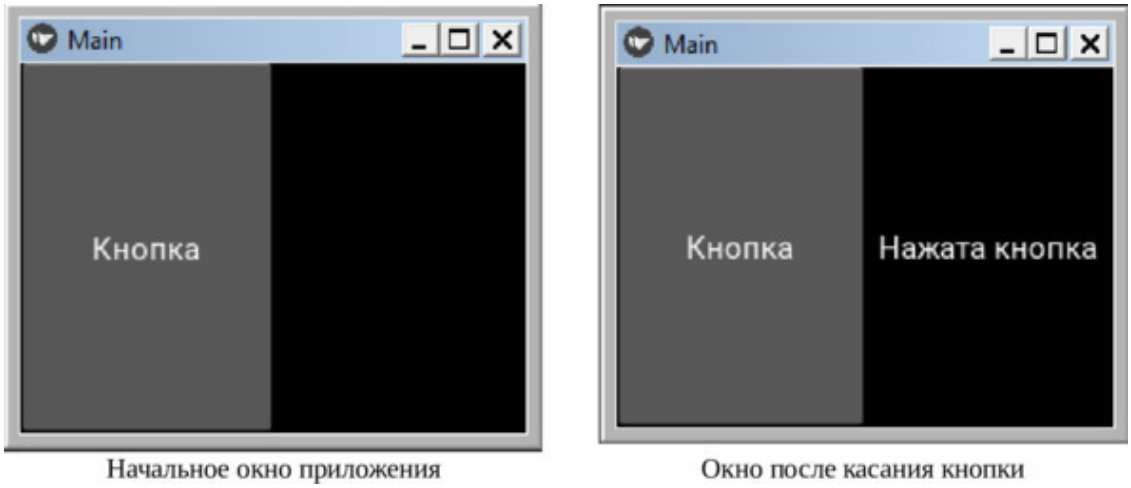


Рис. 2.10. Результаты выполнения приложения из модуля Button_Ids.py

Как видно из данного рисунка, текст «Нажата кнопка», сначала был передан из фрагмента кода на KV во фрагмент кода на Python, а затем возвращен обратно. Для этого были использованы ключевые слова ids и id.

Рассмотрим использование идентификаторов виджетов для обмена параметрами между фрагментами кода на языке KV и на Python, на развернутом примере простейшего калькулятора. Для этого создадим файл с именем Simpl_Calc.py и напишем в нем следующий код (листинг 2.11).

Листинг 2.11. Демонстрация использования ключевых слов ids и id (модуль Simpl_Calc.py)

```
# Модуль Simpl_Calc.py
from kivy. app import App
from kivy.lang import Builder
from kivy.uix.boxlayout import BoxLayout

KV = «»»
box:
…… #корневой виджет
…… id: root_widget
…… orientation: ’vertical’

…… #поле для ввода исходных данных
…… TextInput:
…… …… id: entry
```

```
…… …… font_size: 32
…… …… multiline: False

…… #кнопка для выполнения расчета
…… Button:
…… …… text: «=»
…… …… font_size: 64
…… …… #on_press: root.result (entry. text)
…… …… on_press: root_widget.result (entry. text)

…… #поле для показа результатов расчета
…… Label:
…… …… id: itog
…… …… text: «Итого»
…… …… font_size: 64
«»»

# класс, задающий корневой виджет
class box (BoxLayout):
…… # Функция подсчета результата
…… def result (self, entry_text):
…… …… if entry_text:
…… …… …… try:
…… …… …… …… # Формула для расчета результатов
…… …… …… …… result = str (eval (entry_text))
…… …… …… …… self.ids [«itog»].text = result
…… …… …… except Exception:
…… …… …… …… # если введено не число
…… …… …… …… self.ids [«itog»].text = «Ошибка»

# базовый класс приложения
class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В этом модуле создан базовый класс приложения MainApp, который обеспечивает запуск приложения, и пользовательский класс box на основе базового класса BoxLayout (контейнер – коробка). В этом классе создана функция def result, в которой происходят вычисления. В коде на языке KV созданы следующие элементы интерфейса:

– корневой виджет box (id: root_widget);

– поле для ввода текста TextInput (id: entry);

– кнопка для запуска процесса выполнения расчетов (id: itog);

– метка Label для вывода на экран результатов расчета.

С использованием имен-идентификаторов происходит обмен данными межу фрагментами кода на языке KV и на Python. После запуска приложения получим следующий результат (рис.2.11).

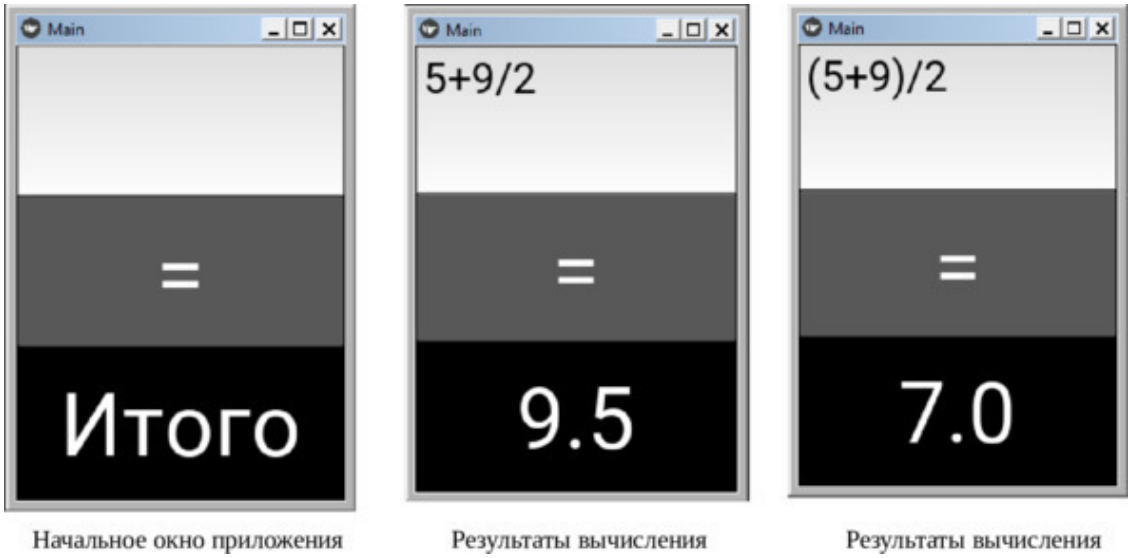


Рис. 2.11. Результаты выполнения приложения из модуля Simpl_Calc.py

Как видно из данного рисунка, мини – калькулятор работает корректно. При этом интерфейс приложения создан на языке KV, а расчеты выполняются в коде на языке Python.

Для ввода данных в текстовое поле необходимо использовать клавиатуру. При этом Kivy определит, на каком устройстве запущено приложение: на настольном компьютере, или на мобильном устройстве. В зависимости от этого будут задействованы разные клавиатуры:

– если приложение запущено на настольном компьютере, то будет использоваться клавиатура этого компьютера;

– если приложение запущено на мобильном устройстве, то будет использоваться всплывающая клавиатура этого устройства.

Если скомпилировать приведенное выше приложение и запустить его на смартфоне, то получим следующий результат (рис.2.12).

Начальный экран

Получение фокуса – русский язык

Получение фокуса – английский язык

Потеря фокуса – исчезновение клавиатуры

Рис. 2.12. Результаты выполнения приложения из модуля Simpl_Calc.py на мобильном устройстве

Как видно из данного рисунка, при загрузке приложения клавиатура на экране отсутствует. Как только поле для ввода текста получает фокус (касание поля), то появляется клавиатура. На этой клавиатуре можно набирать алфавитно-цифровую информацию и переключать язык ввода. При нажатии на кнопку со знаком «=» клавиатура исчезает, и становятся видны результаты расчета.

В коде на языке Kivy допускается использования некоторых операторов и выражений Python. При этом выражение может занимать только одну строку и должно возвращать значение. Рассмотрим это на простом примере. Создадим файл с именем Button_If.py и напишем в нем следующий код (листинг 2.12).

Листинг 2.12. Демонстрация использования выражений в KV (модуль Button_If.py)

```
# Модуль Button_If.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
BoxLayout:
…… orientation: ’vertical’
…… Button:
…… …… id: bt1
…… …… text: «Кнопка 1»
…… Label:
…… …… text: «Отпущена» if bt1.state == ’normal’ else «Нажата»
«»»

class MainApp (App):
……def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В этом модуле создан корневой виджет BoxLayout. Затем в корневой виджет вложено два элемента: кнопка Button, и метка Label. Кнопка имеет идентификатор «id: bt1». Свойству метки text присвоено выражение:

```
text: «Кнопка отпущена» if bt1.state == ’normal’ else «Кнопка нажата»
```

Это выражение говорит о том, что, если кнопка будет находиться в нормальном состоянии, то метка покажет текст «Кнопка отпущена». А если кнопка будет находиться в нажатом состоянии, то метка покажет текст «Кнопка Нажата». После запуска приложения получим следующий результат (рис.2.13).

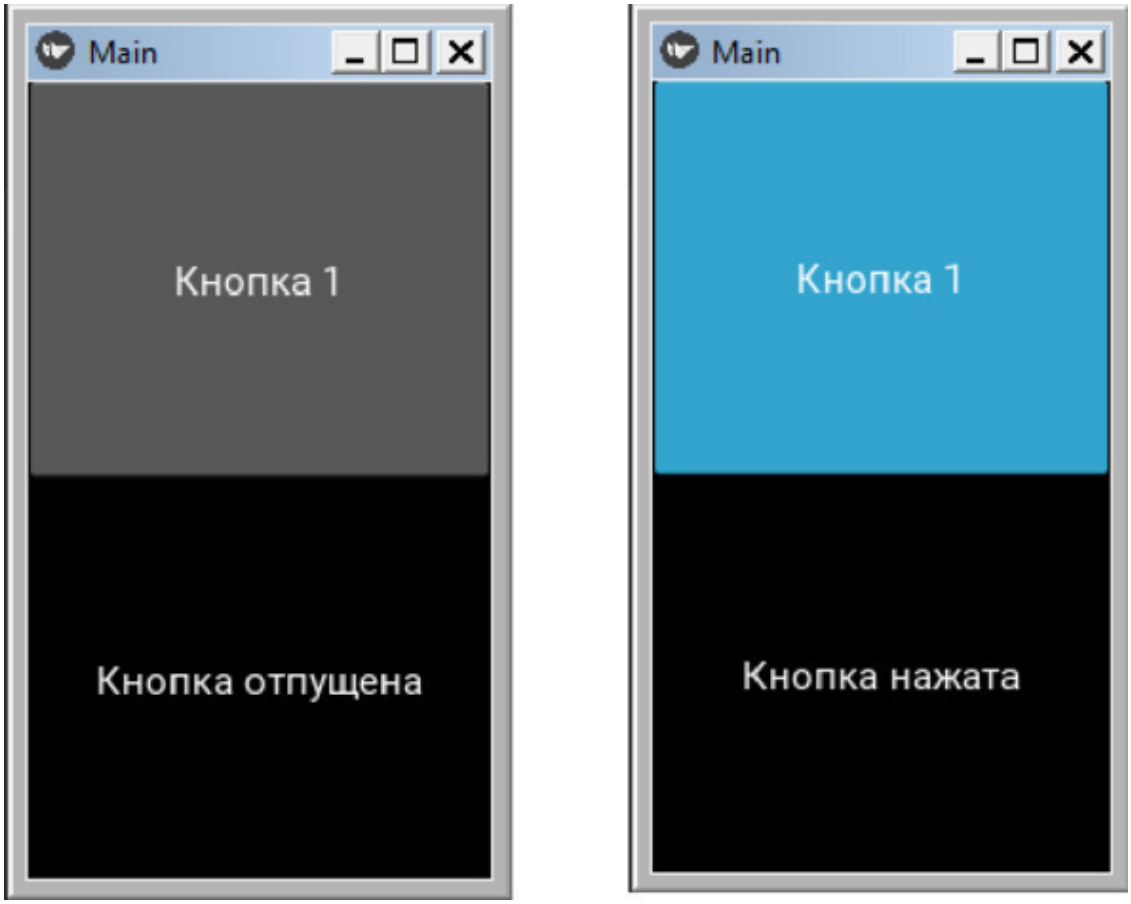


Рис. 2.13. Результаты выполнения приложения из модуля Button_If.py

Итак, мы познакомились с некоторыми особенностями языка KV, с тем, как можно создавать и использовать классы на языке KV и идентифицировать виджеты. Теперь можно более детально познакомиться со структурой приложений на Kivy, а затем перейти к базовым элементам, на основе которых строится пользовательский интерфейс – к виджетам.

2.3. Структура приложений на Kivy

Когда мы пишем большие и сложные приложения, то включение совершенно разных функциональных блоков в один и тот же программный модуль будет вносить беспорядок в программный код. Листинг такого модуля будет длинным и трудно понимаемым даже для самого автора проекта. Большое количество виджетов, расположенных в разных частях длинного программного кода, затруднит построение дерева виджетов и их привязку к параметрам экрана. К счастью в Kivy эта проблема успешно решена путем использования языка KV. Он позволяет сгруппировать все виджеты в одном месте, создать собственное дерево виджетов и естественным образом связывать как свойства виджетов друг с другом, так и с функциями обратного вызова (функциями обработки событий). Это позволяет достаточно быстро создавать прототипы пользовательского интерфейса и гибко вносить в него изменения. Это также позволяет отделить программный код, реализующий функции приложения, от программного кода реализации пользовательского интерфейса.

Есть два способа объединения фрагментов программного кода на Python и KV:

– Метод соглашения имен;

– Использование загрузчика Builder.

Рассмотрим два этих способа на простейших примерах.

2.3.1. Компоновка приложения из фрагментов методом соглашения имен

Допустим, что 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. Рассмотрим использования этого метод на примере (листинг 2.13).

Листинг 2.13. Демонстрация метода соглашения имен (главный модуль приложения, модуль Soglashenie_Imen.py)

```
# модуль Soglashenie_Imen.py
from kivy. app import App # импорт класса – приложения

class Basic_Class (App): # определение базового класса
…… pass

My_App = Basic_Class () # приложение на основе базового класса
My_App.run () # запуск приложения
```

В этом модуле просто создан базовый класс Basic_Class, внутри которого нет выполнения каких либо действий. Теперь создадим файл с именем basic_class. kv и размести в нем следующий код (листинг 2.14).

Листинг 2.14. Текстовый файл, модуль basic_class. kv

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

В этом коде мы создали метку (Label), и свойству метке (text) присвоили значение – «Метка из файла basic_class. kv', указали размер шрифта, которым нужно вывести на экран данный текст – '16pt'. Теперь запустим наше приложение и получим следующий результат (рис.2.14).

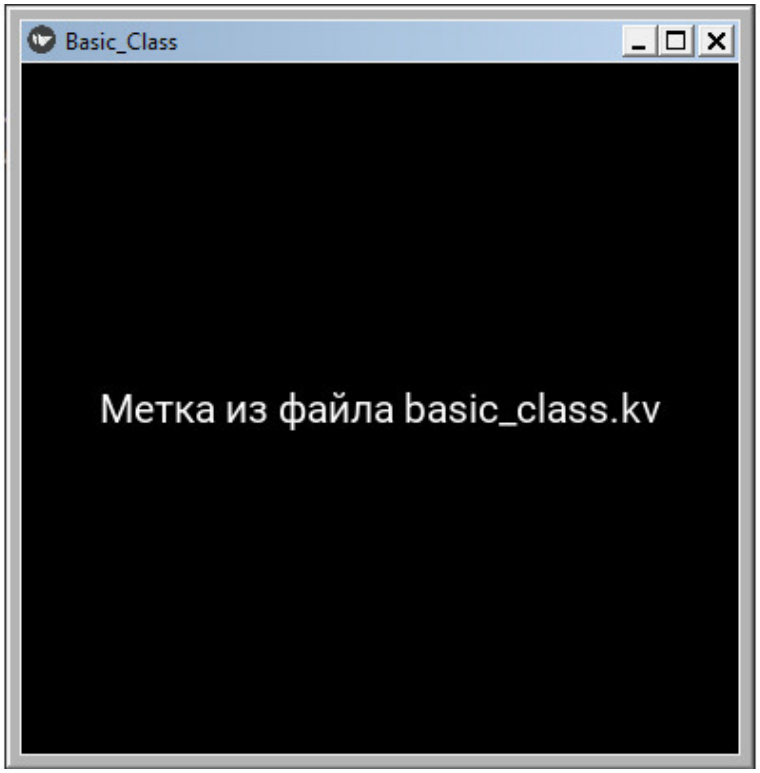


Рис. 2.14. Окно приложения Basic_Class при наличии файла basic_class. kv

Здесь сработал метод соглашения имен, который работает следующим образом:

– По умолчанию при запуске программного модуля базовый класс приложения (Basic_Class) ищет файл с именем – basic_class. kv.

– Если такой файл в папке с приложением имеется, то описанные там визуальные элементы выводятся на экран.

Таким образом, в Kivy реализован первый способ объединения фрагментов приложения, расположенных в разных файлах. Если использовать данный способ, то необходимо выполнять одно важное условие – файл с именем – basic_class. kv должен находиться в то же папке приложения, где находится программный модуль с базовым классом (в нашем случае файл с именем Soglashenie_Imen.py).

2.3.2. Компоновка приложения из фрагментов с использованием загрузчика Builder

Чтобы использовать загрузчик Builder, сначала необходимо выполнить его импорт с использованием следующего кода:

```
from kivy.lang import builder
```

Теперь с помощью Builder можно напрямую загрузить код на языке KV либо из текстового файла проекта с любым именем (но с расширением. kv), либо из текстовой строки базового программного модуля.

Сделать загрузку виджетов из файла. kv можно с использованием следующей команды:

```
Builder. load_file (». Kv/file/path»)
```

Сделать загрузку виджетов из текстовой строки программного модуля можно с использованием следующей команды:

```
Builder. load_string (kv_string)
```

Рассмотрим это на простых примерах. Напишем программный код для загрузки кода на языке KV из текстового файла проекта, файл Metod_Builder.py (листинг 2.15).

Листинг 2.15. Демонстрация метода Builder (загрузка кода на KV из текстового файла) модуль Metod_Builder.py

```
# модуль Metod_Builder.py
from kivy. app import App # импорт класса – приложения
from kivy.lang import Builder # импорт метода Builder

kv_file = Builder. load_file (». /basic_class. kv»)

class Basic_Class (App): # определение базового класса
…… def build (self):
…… … … return kv_file

My_App = Basic_Class () # приложение на основе базового класса
My_App.run () # запуск приложения
```

Здесь мы создали переменную kv_file и, с использованием метода Builder. load_file, загрузили в нее код из файла». /basic_class. kv', который находится в головной папке проекта. Затем в базовом классе создали функцию def build (self), которая возвращает значение переменной kv_file. Результат работы приложения будет таким же, как приведено на предыдущем рисунке. При использовании данного метода явным образом задается путь к файлу basic_class. kv, поэтому, в отличие от метода соглашения имен, данный файл может находиться в любой папке проекта и иметь любое имя.

Проверим, как это работает. Изменим приведенный выше программный код следующим образом, файл Metod_Builder2.py (листинг 2.16).

Листинг 2.16. Демонстрация метода Builder Metod_Builder2.py (загрузка кода на KV из текстового файла, расположенного в произвольном месте приложения), модуль Metod_Builder2.py

```
# модуль Metod_Builder2.py
from kivy. app import App # импорт класса – приложения
from kivy.lang import Builder # импорт метода Builder
```

```
# загрузка кода из KV файла
kv_file = Builder. load_file (». /KV_file/main_screen. kv')

class Basic_Class (App): # определение базового класса
…… def build (self):
…… … … return kv_file

My_App = Basic_Class () # приложение на основе базового класса
My_App.run () # запуск приложения
```

Здесь мы создали переменную kv_file и, с использованием метода Builder. load_file, загрузили в нее код из файла «main_screen. kv', который находится в папке проекта KV_file. Теперь создадим папку с именем KV_file, в этой папке сформируем файл с именем main_screen. kv и внесем в него следующий программный код (листинг 2.17).

Листинг 2.17. Текстовый файл (модуль main_screen. kv)

```
# файл main_screen. kv
Label:
…… text: («Метка из файла./KV_file/main_screen. kv')
…… font_size: '16pt'
```

При запуске данной версии приложения получим следующий результат (рис.2.15).

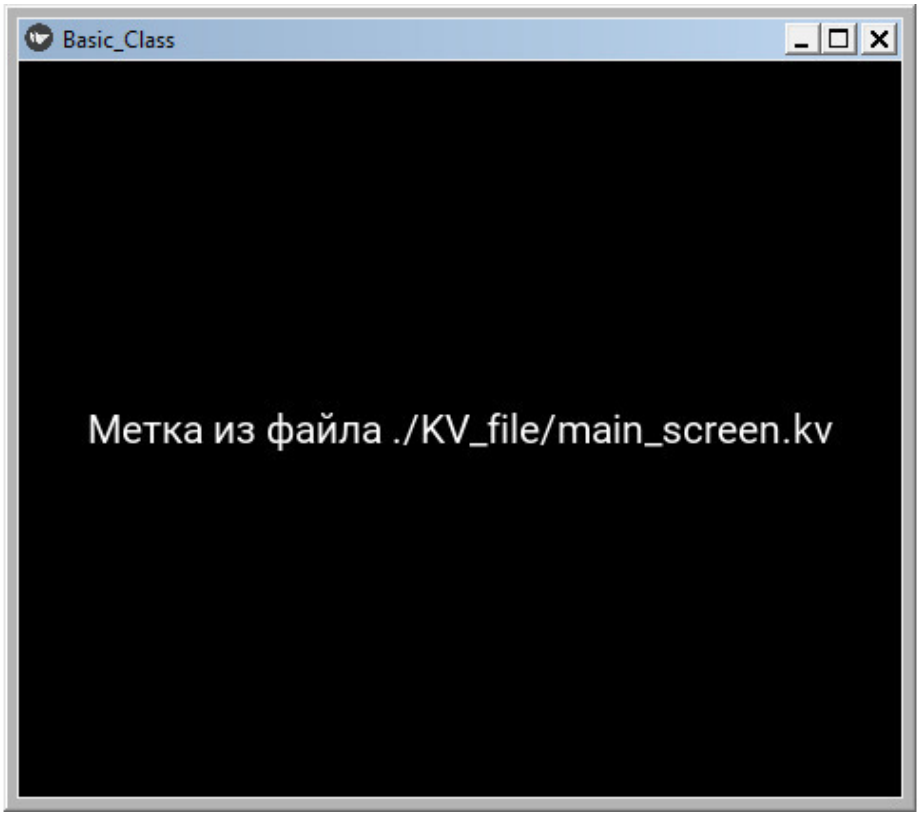


Рис. 2.15. Окно приложения Basic_Class при наличии файла main_screen. kv

Таким образом, в Kivy реализован второй способ отделение кода с логикой работы приложения от кода с описанием интерфейса. Если использовать данный способ, то – файл с кодом на KV (<имя файла>.kv) может иметь любое имя и находиться в любой папке приложения.

Выше было упомянуто, что загрузку виджетов можно сделать и из текстовой строки программного модуля, в котором находится базовый класс. Проверим это, напишем программный код для загрузки кода на языке KV из текстовой строки программного модуля с базовым классом (листинг 2.18).

Листинг 2.18. Демонстрация метода Builder (загрузка кода на KV из текстовой строки) модуль Metod_Builder1.py

```
# модуль Metod_Builder1.py
from kivy. app import App # импорт класса – приложения
from kivy.lang import Builder # импорт метода Builder

# создание текстовой строки
my_str = «»»
Label:
…… text: («Загрузка метки из текстовой строки»)
…… font_size: ’16pt’
«»»

# загрузка кода из текстовой строки
kv_str = Builder. load_string (my_str)

class Basic_Class (App): # определение базового класса
…… def build (self):
…… … … return kv_str

My_App = Basic_Class () # приложение на основе базового класса
My_App.run () # запуск приложения
```

Здесь мы создали текстовую строку my_str и поместили в нее программный код на языке KV. Затем в переменную kv_str с использованием метода Builder. load_string, загрузили код из строковой переменной my_str. Затем в базовом классе создали функцию def build (self), которая возвращает значение переменной kv_str. Результат работы этого приложения представлен на рис.2.16.

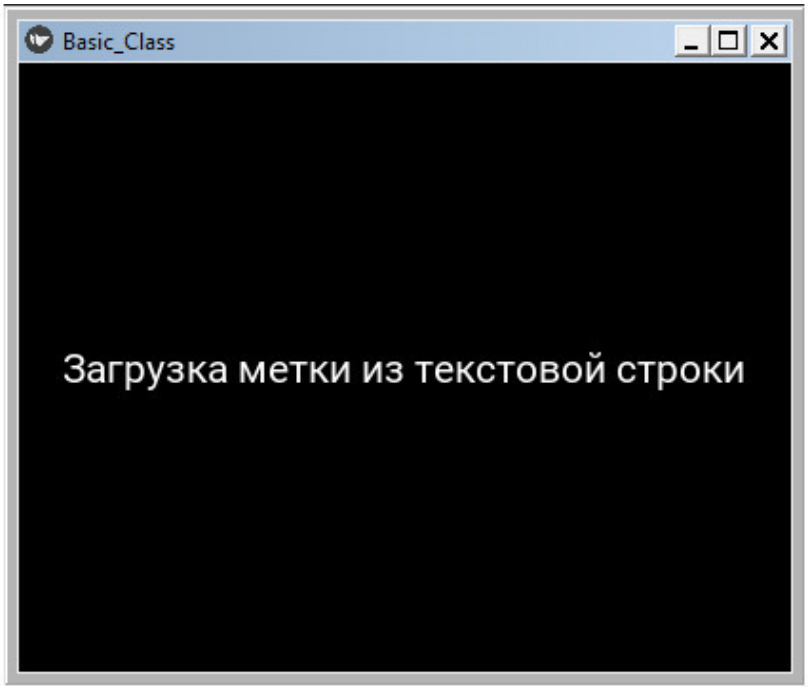


Рис. 2.16. Окно приложения Basic_Class при загрузки метки из текстовой строки

Таким образом, в Kivy реализована возможность не только отделять код с логикой работы приложения от кода с описанием интерфейса, но и совмещать их в рамках одного программного модуля.

Итак, к данному моменту мы установили, что Kivy позволяет создавать элементы интерфейса приложения (виждеты) на языке KV, и либо разделять, либо совмещать программные коды, описывающие интерфейс и логику работы приложения. Еще выяснили, что по умолчанию виджеты располагаются в центре окна приложения и, при изменении размеров окна, перерисовываются автоматически, сохраняя свои пропорции и положение. Это очень важная особенность Kivy, которая обеспечивает адаптацию приложения под размер экрана того устройства, на котором оно запущено.

2.4. Виджеты в Kivy

Интерфейс пользователя в приложениях на Kivy строится на основе виджетов. Виджеты Kivy можно классифицировать следующим образом.

– UX-виджеты, или видимые виджеты (они отображаются в окне приложения, и пользователь взаимодействует с ними);

– виджеты контейнеры или «макеты» (они не отображаются в окне приложения и служат контейнерами для размещения в них видимых виджетов);

– сложные UX-виджеты (комбинация нескольких виджетов, обеспечивающая совмещение их функций);

– виджеты поведения (контейнеры, которые обеспечивают изменение поведения находящихся в них элементов);

– диспетчер экрана (особый виджет, который управляет сменой экранов приложения).

Видимые виджеты служат для создания элементов пользовательского интерфейса. Типичными представителями таких виджетов являются кнопки, выпадающие списки, вкладки и т. п. Невидимые виджеты используются для позиционирования видимых элементов в окне приложения. Любой инструментарий графического интерфейса пользователя поставляется с набором виджетов. Фреймворк Kivy и библиотека KivyMD имеет множество встроенных виджетов.

Работа со свойствами виджетов в Kivy имеет свои особенности. Например, с параметрами свойств по умолчанию они располагаются в центре элемента, к которому привязаны (например, к главному окну приложения) и занимают всю его площадь. Однако при разработке интерфейса пользователя необходимо размещать виджеты в разных частях экрана, менять их размеры, цвет и ряд других свойств. Для этого каждый виджет имеет набор свойств и методов, которые разработчик может устанавливать по своему усмотрению. Кроме того, разработчик имеет возможность вкладывать один виджет в другой и таким образом строить «дерево виджетов». Теперь можно на простых примерах познакомиться с основными видимыми виджетами Kivy.

2.5. Виджеты пользовательского интерфейса (UX-виджеты)

У фреймворка Kivy имеется небольшой набор видимых виджетов:
– Label – метка или этикетка (текстовая надпись в окне приложения);
– Button – кнопка;
– Checkbox – флажок;
– Image – изображение;
– Slider – слайдер;
– ProgressBar – индикатор;
– TextInput – поле для ввода текста;
– ToggleButton – кнопка «переключатель»;
– Switch – выключатель;
– Video – окно для демонстрации видео из файла;
– Widget – пустая рамка (поле).

Это достаточно скромный, базовый набор визуальных элементов, но мы именно с них начнем знакомство с Kivy.

Примечание.

Более богатый набор визуальных элементов реализован в библиотеке KivyMD. О них будет подробно рассказано в последующих главах.

2.5.1. Виджет Label – метка

Виджет Label используется для отображения текста в окне приложения. Он поддерживает вывод символов как в кодировке ascii, так и в кодировке unicode. Покажем на простом примере, как можно использовать виджет Label в приложении. Для этого создадим файл с именем K_Label_1.py и напишем в нем следующий код (листинг 2.19).

Листинг 2.19. Пример использования виджета Label (модуль K_Label_1.py)

```
# модуль K_Label_1.py
from kivy. app import App
from kivy.uix.label import Label

class MainApp (App):
…… def build (self):
…… … … L = Label (text=«Это текст», font_size=50)
…… … … return L

MainApp().run ()
```

В этом модуле мы создали объект-метку L на основе базового класса Label. Для метки в свойстве text задали значение, который нужно вывести на экран – «Это текст», а в свойстве font_size задали размер шрифта -50. После запуска данного приложения получим следующий результат (рис.2.17).

Рис. 2.17. Результаты выполнения приложения из модуля K_Label_1.py

В данном примере объект Label был создан в коде на языке Python. Реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_Label_2.py и напишем в нем следующий код (листинг 2.20).

Листинг 2.20. Пример использования виджета Label (модуль K_Label_2.py)

```
# модуль K_Label_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Label:
…… text: «Это текст»
…… font_size: 50
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект Label был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Метка Label имеет ряд свойств, которые позволяют задать выводимый текст и параметры шрифта:

– text – текст, выводимый в видимую часть виджета (текстовое содержимое метки, надпись на кнопке и т.п.);

– font_size – размер шрифта;

– color – цвет шрифта;

– font_name – имя файла с пользовательским шрифтом (.ttf).

2.5.2. Виджет Button – кнопка

Кнопка Button – это элемент интерфейса, при касании которого будут выполнены запрограммированные действия. Виджет Button имеет те же свойства, что и виджет Label. Покажем на простом примере, как можно использовать виджет Button в приложении. Для этого создадим файл с именем K_Button_1.py и напишем в нем следующий код (листинг 2.21).

Листинг 2.21. Пример использования виджета Button (модуль K_Button_1.py)

```
# модуль K_Button_1.py
from kivy. app import App
from kivy. uix. button import Button

class MainApp (App):
…… def build (self):
…… … … btn = Button (text=«Это кнопка», font_size=50)
…… … … return btn

MainApp().run ()
```

В этом модуле мы создали объект-кнопку btn на основе базового класса Button. Для кнопки в свойстве text задали надпись, которую нужно вывести на кнопку – «Это кнопка», а в свойстве font_size задали размер шрифта -50. После запуска данного приложения получим следующий результат (рис.2.18).

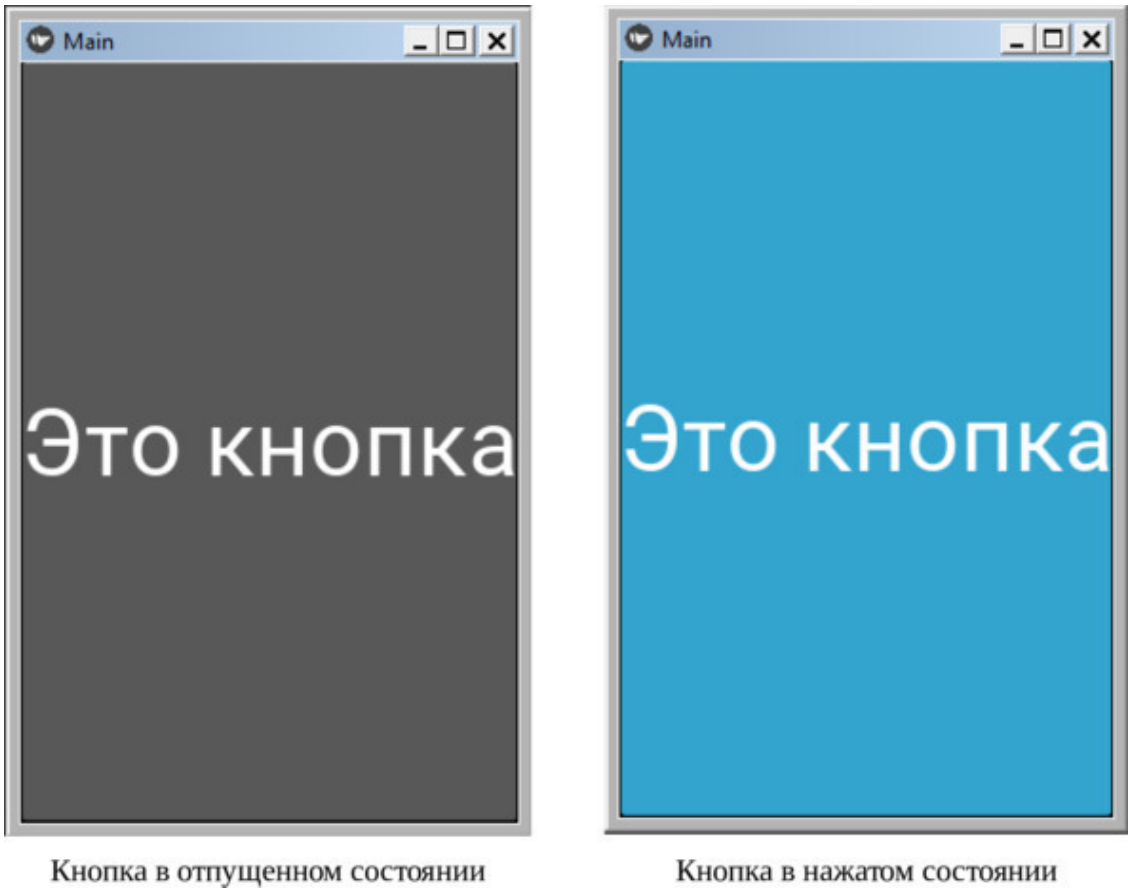


Рис. 2.18. Результаты выполнения приложения из модуля K_Button _1.py

В данном примере объект Button был создан в коде на языке Python. Как видно из данного рисунка, в нажатом и отпущенном состоянии кнопка будет иметь разный вид. В данном примере мы не программировали действия, которые будут выполнены при касании кнопки, этот вопрос будет освящен в разделе, касающемся обработки событий. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_Button_2.py и напишем в нем следующий код (листинг 2.22).

Листинг 2.22. Пример использования виджета Button (модуль K_Button _2.py)

```
# модуль K_Button_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Button:
…… text: «Это кнопка»
…… font_size: 50
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект Button был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Кнопка Button имеет ряд свойств, которые позволяют задать надпись на кнопке, параметры шрифта, и запустить реакцию на события или изменение состояния:

– text – надпись на кнопке;
– font_size – размер шрифта;
– color – цвет шрифта;
– font_name – имя файла с пользовательским шрифтом (.ttf).
– on_press – событие, возникает, когда кнопка нажата;
– on_release – событие, возникает, когда кнопка отпущена;
– on_state – состояние (изменяется тогда, когда кнопка нажата или отпущена).

2.5.3. Виджет CheckBox – флажок

Виджет CheckBox (флажок) это элемент в виде мини-кнопки с двумя состояниями. Флажок в данном элементе можно либо поставить, либо снять. Покажем на простом примере, как можно использовать виджет CheckBox в приложении. Для этого создадим файл с именем K_CheckBox_1.py и напишем в нем следующий код (листинг 2.23).

Листинг 2.23. Пример использования виджета CheckBox (модуль K_CheckBox_1.py)

```
# модуль K_CheckBox_1.py
from kivy. app import App
from kivy.uix.checkbox import CheckBox

class MainApp (App):
…… def build (self):
…… …… checkbox = CheckBox ()
…… …… return checkbox

MainApp().run ()
```

В этом модуле мы создали объект-флажок checkbox на основе базового класса CheckBox. После запуска данного приложения получим следующий результат (рис.2.19).

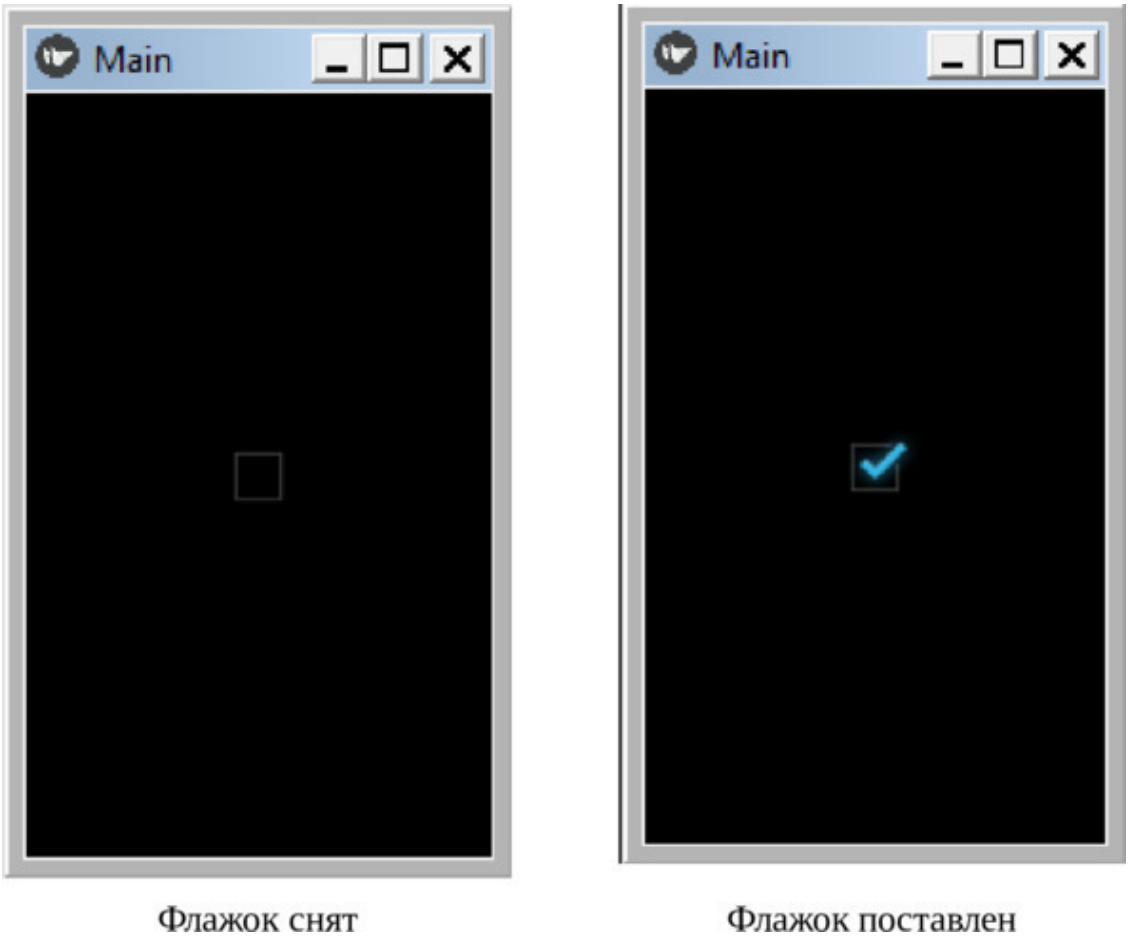


Рис. 2.19. Результаты выполнения приложения из модуля K_CheckBox _1.py

В данном примере объект CheckBox был создан в коде на языке Python. Как видно из данного рисунка, при установке и снятии флажка виджет будет иметь разный вид. В данном примере мы не программировали действия, которые будут выполнены при изменении состояния флажка, этот вопрос будет освящен в разделе, касающемся обработки событий. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_CheckBox_2.py и напишем в нем следующий код (листинг 2.24).

Листинг 2.24. Пример использования виджета CheckBox (модуль K_CheckBox _2.py)

```
# модуль K_CheckBox_2.py
from kivy. app import App
from kivy.lang import Builder
KV = «»»
CheckBox:
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект CheckBox был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Флажок CheckBox имеет ряд свойств, которые позволяют задать цвет и запустить реакцию на изменение состояния:

– color – цвет флажка (в формате r, g,b,a);

– active – состояние в виде логического значения (True – когда флажок поставлен, False – когда флажок снят).

2.5.4. Виджет Image – рисунок

Виджет Image (рисунок) служит для вывода в окно приложения изображения. Покажем на простом примере, как можно использовать виджет Image в приложении. Для этого создадим файл с именем K_Image_1.py и напишем в нем следующий код (листинг 2.25).

Листинг 2.25. Пример использования виджета Image (модуль K_Image_1.py)

```
# модуль K_ Image _1.py
from kivy. app import App
from kivy.uix.image import Image

class MainApp (App):
…… def build (self):
…… … … img = Image(source="./Images/Fon2.jpg»)
…… … … return img

MainApp().run ()
```

В этом модуле мы создали объект-изображение img на основе базового класса Image. Для изображения в свойстве source задали путь к файлу с изображением. После запуска данного приложения получим следующий результат (рис.2.20).

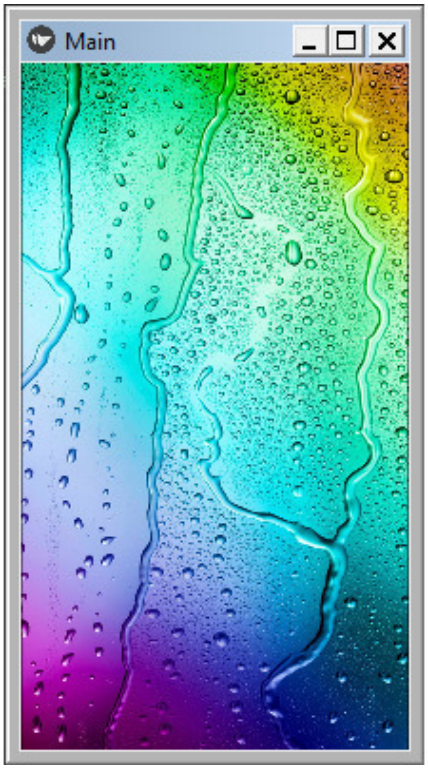


Рис. 2.20. Результаты выполнения приложения из модуля K_Image _1.py

В данном примере объект Image был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_Image_2.py и напишем в нем следующий код (листинг 2.26).

Листинг 2.26. Пример использования виджета Image (модуль K_ mage_2.py)

```
# модуль K_Image_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Image:
…… source: "./Images/Fon2.jpg»
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект Image был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Виджет Image имеет свойства, которые позволяют загрузить изображение и придать ему оттенок:

– source – источник (путь к файлу для загрузки изображения);

– color – цвет изображения (в формате r, g, b, a), можно использовать для «подкрашивания» изображения.

Этот виджет имеет подкласс AsyncImage, который позволяет загрузить изображение асинхронно (например, из интернета с веб-сайта). Это может быть полезно, поскольку не блокирует приложение в ожидании загрузки изображения (оно загружается в фоновом потоке).

2.5.5. Виджет Slider – слайдер (бегунок)

Виджет Slider (слайдер) это бегунок, который поддерживает горизонтальную и вертикальную ориентацию и используется в качестве полосы прокрутки. Покажем на простом примере, как можно использовать виджет Slider в приложении. Для этого создадим файл с именем K_Slider_1.py и напишем в нем следующий код (листинг 2.27).

Листинг 2.27. Пример использования виджета Slider (модуль K_Slider_1.py)

```
# модуль K_Slider_1.py
from kivy. app import App
from kivy.uix.slider import Slider

class MainApp (App):
…… def build (self):
…… … … slide = Slider (orientation=’vertical’,
…… … … value_track=True,
…… … … value_track_color= (1, 0, 0, 1))
…… … … return slide

MainApp().run ()
```

В этом модуле мы создали объект-бегунок slide на основе базового класса Slider. Для бегунка задали следующие свойства:

– orientation=’vertical’ – вертикальная ориентация;
– value_track=True – показать след бегунка;
– value_track_color= (1, 0, 0, 1) – задан цвет следа бегунка (красный).

После запуска данного приложения получим следующий результат (рис.2.21).

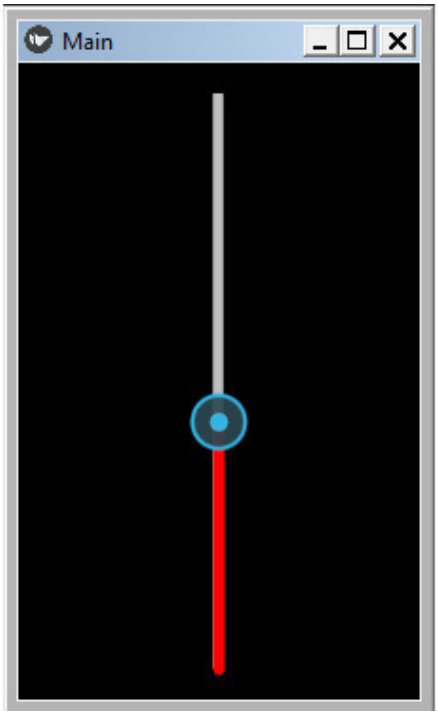


Рис. 2.21. Результаты выполнения приложения из модуля K_Slider_1.py (при вертикальном расположении бегунка)

В данном примере объект Slider был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_Slider_2.py и напишем в нем следующий код (листинг 2.28).

Листинг 2.28. Пример использования виджета Slider (модуль K_Slider_2.py)

```
# модуль K_Slider_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Slider:
…… orientation: ’horizontal’
…… value_track: True
…… value_track_color: 1, 0, 0, 1
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект Slider был создан в коде на языке KV, и было изменено одно свойство – ориентация. В данном коде задана горизонтальная ориентация бегунка. После запуска данного приложения получим следующий результат (рис.2.22).

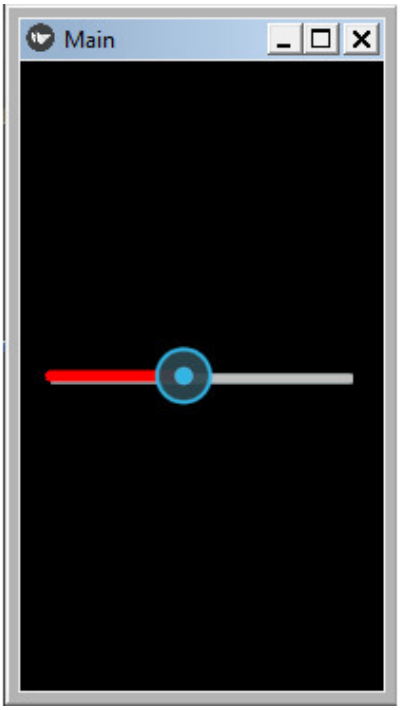


Рис. 2.22. Результаты выполнения приложения из модуля K_Slider_2.py (при горизонтальном расположении бегунка)

Бегунок Slider имеет ряд свойств, которые позволяют задать некоторые параметры, запустить реакцию на события или изменение состояния:

– min – минимальное значение (например – 0);

– max – максимальное значение (например – 500);

– value – текущее (начальное) значение (например – 50);

– step – шаг изменения значения (например – 1);

– value_track_color – цвет следа бегунка (в формате r, g, b, a);

– value_track – показывать след бегунка (True – да, False – нет)

– orientation – ориентация бегунка ('vertical' – вертикальная, 'horizontal' – горизонтальная);

– on_touch_down – событие (касание бегунка);

– on_touch_up – событие (бегунок отпущен);

– on_touch_move – событие (касание бегунка с перемещением).

2.5.6. Виджет ProgressBar – индикатор

Виджет ProgressBar (индикатор) используется для отслеживания хода выполнения любой задачи. Покажем на простом примере, как можно использовать виджет ProgressBar в приложении. Для этого создадим файл с именем K_ProgressBar_1.py и напишем в нем следующий код (листинг 2.29).

Листинг 2.29. Пример использования виджета ProgressBar (модуль K_ProgressBar_1.py)

```
# модуль K_ProgressBar_1,py
from kivy. app import App
from kivy.uix.progressbar import ProgressBar

class MainApp (App):
…… def build (self):
…… …… Progress = ProgressBar (max=1000)
…… …… Progress.value = 650
…… …… return Progress

MainApp().run ()
```

В этом модуле мы создали объект-индикатор Progress на основе базового класса ProgressBar. Для индикатора задали следующие свойства:

– max=1000 – максимальное значение шкалы бегунка;

– value = 650 – текущее положение на шкале бегунка.

После запуска данного приложения получим следующий результат (рис.2.23).

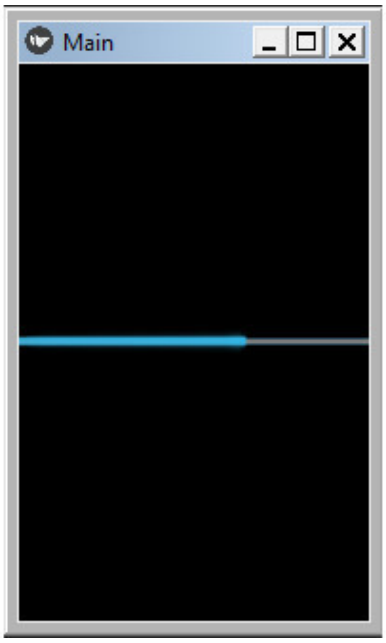


Рис. 2.23. Результаты выполнения приложения из модуля ProgressBar _2.py

В данном примере объект ProgressBar был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_ProgressBar_2.py и напишем в нем следующий код (листинг 2.30).

Листинг 2.30. Пример использования виджета ProgressBar (модуль K_ProgressBar_2.py)

```
# модуль K_ProgressBar_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
ProgressBar:
…… max: 1000
…… value: 650
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект ProgressBar был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Индикатор ProgressBar имеет ряд свойств, которые позволяют задать и получить некоторые параметры:

– max – максимальное значение;

– value – текущее значение;

2.5.7. Виджет TextInput – поле для ввода текста

Виджет TextInpu (текстовое поле) используется для ввода и редактирования текста. Покажем на простом примере, как можно использовать виджет TextInput в приложении. Для этого создадим файл с именем K_TextInput_1.py и напишем в нем следующий код (листинг 2.31).

Листинг 2.31. Пример использования виджета TextInput (модуль K_TextInput_1.py)

```
# модуль K_TextInput_1.py
from kivy. app import App
from kivy. uix. textinput import TextInput

class MainApp (App):
…… def build (self):
…… … … my_text = TextInput (font_size=30)
…… … … return my_text

MainApp().run ()
```

В этом модуле мы создали объект my_text – поле для ввода текста на основе базового класса TextInput. В свойстве font_size=30 задан размер шрифта. После запуска данного приложения получим следующий результат (рис.2.24).

Рис. 2.24. Результаты выполнения приложения из модуля K_TextInput_1.py

В данном примере объект TextInput был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_TextInput_2.py и напишем в нем следующий код (листинг 2.31).

Листинг 2.31. Пример использования виджета TextInput (модуль K_TextInput_2.py)

```
# модуль K_TextInput_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
…… TextInput:
…… font_size: 30
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект TextInput был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Виджет TextInput имеет ряд свойств, которые позволяют задать вводимому тексту параметры шрифта:

– text – текст (текстовое содержимое поля ввода.);
– font_size – размер шрифта;
– color – цвет шрифта;
– font_name – имя файла с пользовательским шрифтом (.ttf);
– password – скрывать вводимые символы (при значении свойства True);
– password_mask – маска символа (символ, который скрывает вводимый текст).

2.5.8. Виджет ToggleButton – кнопка «с залипанием»

Виджет ToggleButton действует как кнопка с эффектом залипания. Когда пользователь касается кнопки, она нажимается и остается в нажатом состоянии, после второго касания кнопка возвращается в исходное состояние. Покажем на простом примере, как можно использовать виджет ToggleButton в приложении. Для этого создадим файл с именем K_ToggleButton_1.py и напишем в нем следующий код (листинг 2.33).

Листинг 2.33. Пример использования виджета ToggleButton (модуль K_ToggleButton_1.py)

```
# модуль K_ToggleButton_1.py
from kivy. app import App
from kivy.uix.togglebutton import ToggleButton

class MainApp (App):
…… def build (self):
…… …… t_but = ToggleButton (text=«Кнопка»,
…… …… font_size=50)
…… …… return t_but

MainApp().run ()
```

В этом модуле мы создали объект t_but – кнопка с эффектом залипания на основе базового класса ToggleButton. Свойству text присвоили значение «Кнопка» и задали размер шрифта font_size=50. После запуска данного приложения получим следующий результат (рис.2.25).

Кнопка в отжатом состоянии

Кнопка в нажатом состоянии

Рис. 2.25. Результаты выполнения приложения из модуля K_ToggleButton_1.py

В данном примере объект ToggleButton был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_ToggleButton_2.py и напишем в нем следующий код (листинг 2.34).

Листинг 2.34. Пример использования виджета ToggleButton (модуль K_ToggleButton_2.py)

```
# модуль K_ToggleButton_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
ToggleButton:
…… text: «Кнопка»
…… font_size: 50
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект ToggleButton был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Кнопки ToggleButton могут быть сгруппированы для создания группы переключателей. В этом случае только одна кнопка в группе может находиться в «нажатом» состоянии. Имя группы может быть строкой с произвольным содержанием. Для примера создадим файл с именем K_ToggleButton_3.py и напишем в нем следующий код (листинг 2.35).

Листинг 2.35. Пример использования виджета ToggleButton в группе (модуль K_ToggleButton_3.py)

```
# модуль K_ToggleButton_3.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
BoxLayout:
…… orientation: «vertical»
…… ToggleButton:
…… …… text: «Москва»
…… …… group:’city’
…… …… state: ’down’
…… ToggleButton:
…… …… text: «Воронеж»
…… …… group: ’city’
…… ToggleButton:
…… …… text: «Сочи»
…… …… group: ’city’
«»»
```

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

MainApp().run ()
```

В этом модуле создано 3 кнопки, которые собраны в одну группу city. Первая кнопка переведена в нажатое состояние – state: 'down'. После запуска приложения получим следующий результат (рис.2.26).

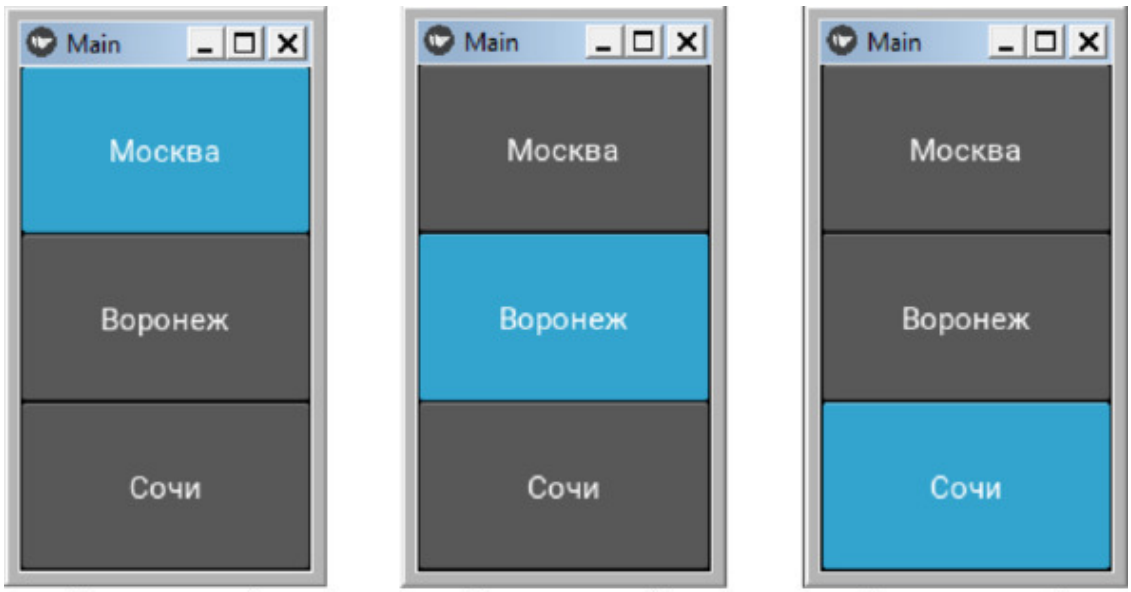


Рис. 2.26. Результаты выполнения приложения из модуля K_ToggleButton_3.py

Как видно из данного рисунка, в нажатом состоянии может находиться только одна кнопка из группы.

Кнопка ToggleButton имеет ряд свойств, которые позволяют задать надпись на кнопке, параметры шрифта, и запустить реакцию на события или изменение состояния:

– text – надпись на кнопке;
– font_size – размер шрифта;
– color – цвет шрифта;
– font_name – имя файла с пользовательским шрифтом (.ttf).
– on_press – событие, возникает, когда кнопка нажата;
– on_release – событие, возникает, когда кнопка отпущена;
– on_state – состояние (изменяется тогда, когда кнопка нажата или отпущена);
– group – задание имени группы (текстовая строка, например 'city');
– state – задание состояние кнопки ('down' – нажата).

2.5.9. Виджет Switch – выключатель

Виджет Switch действует как кнопка – выключатель. При этом имитируется механический выключатель, который либо включается, либо выключается. Виджет Switch имеет два положения включено (on) – выключено (off). Когда пользователь касается кнопки, она переходит из одного положения в другое. Покажем на простом примере, как можно использовать виджет Switch в приложении. Для этого создадим файл с именем K_Switch1.py и напишем в нем следующий код (листинг 2.36).

Листинг 2.36. Пример использования виджета Switch (модуль K_Switch_1.py)

```
# модуль K_Switch1.py
from kivy. app import App
from kivy. uix. switch import Switch

class MainApp (App):
…… def build (self):
…… …… sw = Switch (active=True)
…… …… return sw

MainApp().run ()
```

В этом модуле мы создали объект sw (кнопка выключатель) на основе базового класса Switch. Свойству active (состояние) присвоили значение True (включено). После запуска данного приложения получим следующий результат (рис.2.27).

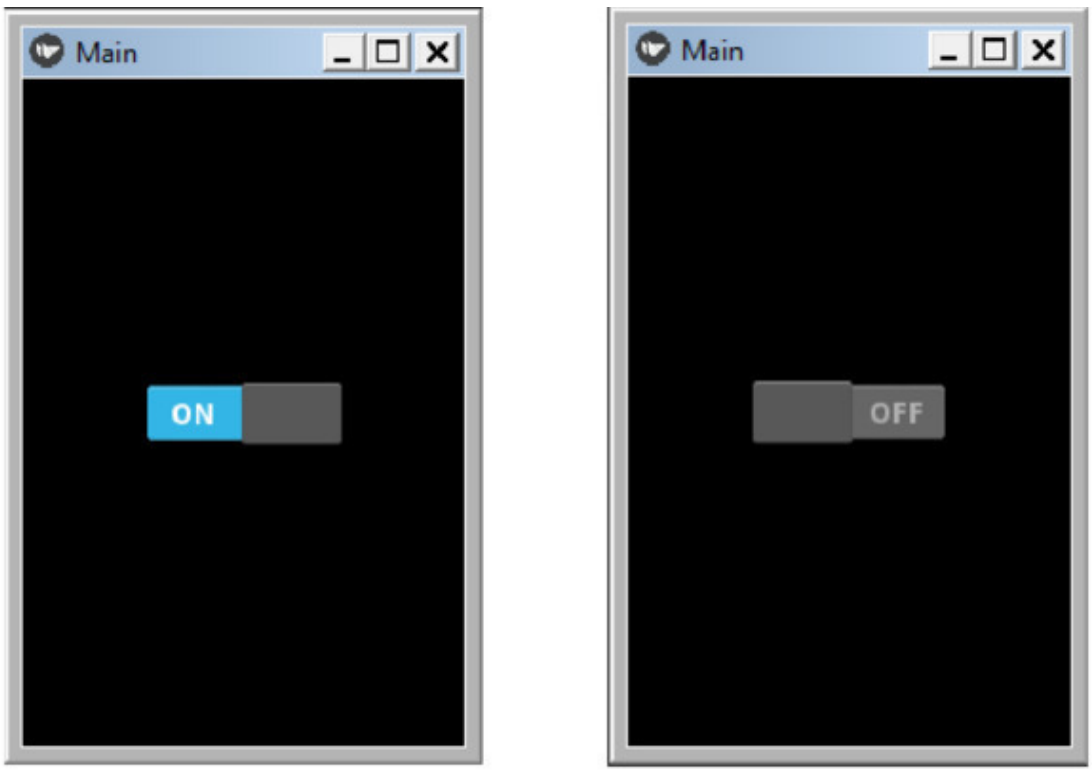


Рис. 2.27. Результаты выполнения приложения из модуля K_Switch1.py

В данном примере объект Switch был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_ Switch_2.py и напишем в нем следующий код (листинг 2.36_1).

Листинг 2.36_1. Пример использования виджета Switch (модуль K_Switch_1.py)

```
# модуль K_Switch2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Switch:
…… active: True
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект Switch был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

По умолчанию виджет является статичным с минимальным размером 83x32 пикселя. Выключатель Switch имеет ряд свойств, которые позволяют задать и получить некоторые параметры:

– active – состояние выключателя (по умолчанию имеет значение False)
– on_touch_down – событие (касание выключателя);
– on_touch_up – событие (выключатель отпущен);
– on_touch_move – событие (касание выключателя с перемещением).

К сожалению, данный элемент не имеет свойства text, поэтому для размещения поясняющих надписей нужно в паре использовать метку Label.

2.5.10. Виджет Video – окно для демонстрации видео

Виджет Video создает окно для демонстрации видео из видео файла и видео потока. Виджет Video имеет свойство play (проигрывать), которое может принимать два значения: True – начать проигрывание, False – остановить проигрывание.

Примечание.

В зависимости от вашей платформы и установленных плагинов вы сможете воспроизводить видео разных форматов. Например, поставщик видео pygame поддерживает только формат MPEG1 в Linux и OSX, GStreamer более универсален и может читать многие кодеки, такие как MKV, OGV, AVI, MOV, FLV (если установлены правильные плагины gstreamer).

Покажем на простом примере, как можно использовать виджет Video в приложении. Для этого создадим файл с именем K_Video1.py и напишем в нем следующий код (листинг 2.37).

Листинг 2.37. Пример использования виджета Video (модуль K_Video_1.py)

```
# модуль K_Video1.py
from kivy. app import App
from kivy.uix.video import Video

class MainApp (App):
…… def build (self):
…… … … video = Video(source="./Video/My_video.mp4», play=True)
…… … … return video

MainApp().run ()
```

В этом модуле мы создали объект video (окно для показа кадров) на основе базового класса Video. Свойству play (проигрывать) присвоили значение True (включено). После запуска данного приложения получим следующий результат (рис.2.28).

Рис. 2.28. Результаты выполнения приложения из модуля K_Video1.py

Примечание.

Если на вашем компьютере не воспроизводится видео, то, скорее всего это происходит из-за отсутствия нужных кодеков. Для воспроизведения видеофайлов разных форматов нужно в инструментальную среду дополнительно установить модуль ffpyplayer. Для этого необходимо в терминале Pycharm выполнить команду: pip install ffpyplayer

В данном примере объект video был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_Video_2.py и напишем в нем следующий код (листинг 2.37_1).

Листинг 2.37_1. Пример использования виджета Video (модуль K_Video_2.py)

```
# модуль K_Video2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Video:
…… source: "./Video/My_video.mp4»
…… … … play: True
«»»
```

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

MainApp().run ()
```

В данном примере объект Video был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

Объект Video имеет ряд свойств, которые позволяют задать и получить некоторые параметры:

– source – источник (путь к файлу и имя видео файла)

– play – проигрывать (по умолчанию False, для запуска проигрывания установить -True);

– state – состояние (имеет три значения: play – проигрывать, pause – поставить на паузу, stop – остановить);

– volume – громкость звука, значение в диапазоне 0—1 (1 – полная громкость, 0 – отключение звука).

2.5.11. Виджет Widget – базовый класс (пустая поверхность)

Класс Widget является своеобразным базовым классом, необходимым для создания пустой поверхности, которая по умолчанию имеет черный цвет. Это некая основа, или базовый строительный блок интерфейсов GUI в Kivy. Кроме того, эта поверхность может использоваться как холст для рисования.

Покажем на простом примере, как можно использовать виджет Widget в приложении. Для этого создадим файл с именем K_Widget_1.py и напишем в нем следующий код (листинг 2.38).

Листинг 2.38. Пример использования виджета Widget (модуль K_Widget_1.py)

```
# модуль K_Widget_1.py
from kivy. app import App
from kivy. uix. widget import Widget

class MainApp (App):
…… def build (self):
…… … … wid = Widget ()
…… … … return wid

MainApp().run ()
```

В этом модуле мы создали объект wid (пустая поверхность) на основе базового класса Widget. После запуска данного приложения получим следующий результат (рис.2.29).

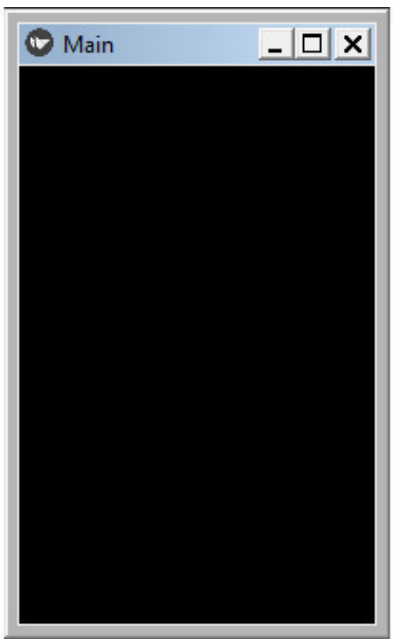


Рис. 2.29. Результаты выполнения приложения из модуля K_ Widget _1.py

Как видно из данного рисунка, класс Widget отобразил пустую поверхность. В данном примере объект wid был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем K_Widget_2.py и напишем в нем следующий код (листинг 2.39).

Листинг 2.39. Пример использования виджета Widget (модуль K_Widget_2.py)

```
# модуль K_Widget_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Widget:
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В данном примере объект Widget был создан в коде на языке KV, а результат работы приложения будет таким же, как представлено на предыдущем рисунке.

У данного класса есть встроенный объект canvas, который можно использовать для рисования на экране. Данный объект может принимать события и реагировать на них. Кроме того, у данного встроенного объекта есть две важные инструкции: Color (цвет) и Rectangle (прямоугольник, рамка). С использованием данных инструкций для созданной поверхности можно задать цвет, или поместить на нее изображение.

Для демонстрации этой возможности создадим файл с именем K_Widget_3.py и напишем в нем следующий код (листинг 2.40).

Листинг 2.40. Пример использования виджета Widget (модуль K_Widget_3.py)

```
# модуль K_Widget_3.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Widget:
…… canvas:
…… … … Color:
…… … … … … … rgba: 0, 1, 0, 1
…… … … Rectangle:
…… … … … … … pos: self. pos
…… … … … … … size: self.size
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В этом модуле мы создали объект Widget, а для объекта canvas в инструкциях задали следующие параметры:

– Color (цвет) – зеленый (rgba: 0, 1, 0, 1);

– Rectangle (рамка) – принимать позицию и размер такими, как у родительского элемента. После запуска данного приложения получим следующий результат (рис.2.30).

Рис. 2.30. Результаты выполнения приложения из модуля K_ Widget _3.py

Как видно из данного рисунка, вся поверхность приобрела зеленый цвет. При изменении размеров окна приложения, рамка виджета автоматически перерисовывается, и продолжает занимать всю поверхность экрана.

Теперь попробуем вставить в рамку изображение. Для демонстрации этой возможности создадим файл с именем K_Widget_4.py и напишем в нем следующий код (листинг 2.41).

Листинг 2.41. Пример использования виджета Widget (модуль K_Widget_4.py)

```
# модуль K_Widget_4.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Widget:
…… canvas:
…… … … #Color:
…… … … #rgba: 1, 0, 0, 1
…… Rectangle:
…… … … source: './Images/Fon.jpg'
…… … … pos: self. pos
…… … … size: self.size
«»»

class MainApp (App):
…… def build (self):
…… … … return Builder. load_string (KV)

MainApp().run ()
```

В этом модуле мы создали объект Widget, а для объекта canvas в инструкцию Rectangle (рамка) загрузили изображение из файла './Images/Fon.jpg'. Инструкция Color (цвет) заком-

ментирована, поэтому изображение будет показано в оригинальном цвете. Если снять комментарии с этих строк, то изображение пример красный оттенок. После запуска данного приложения получим следующий результат (рис.2.31).

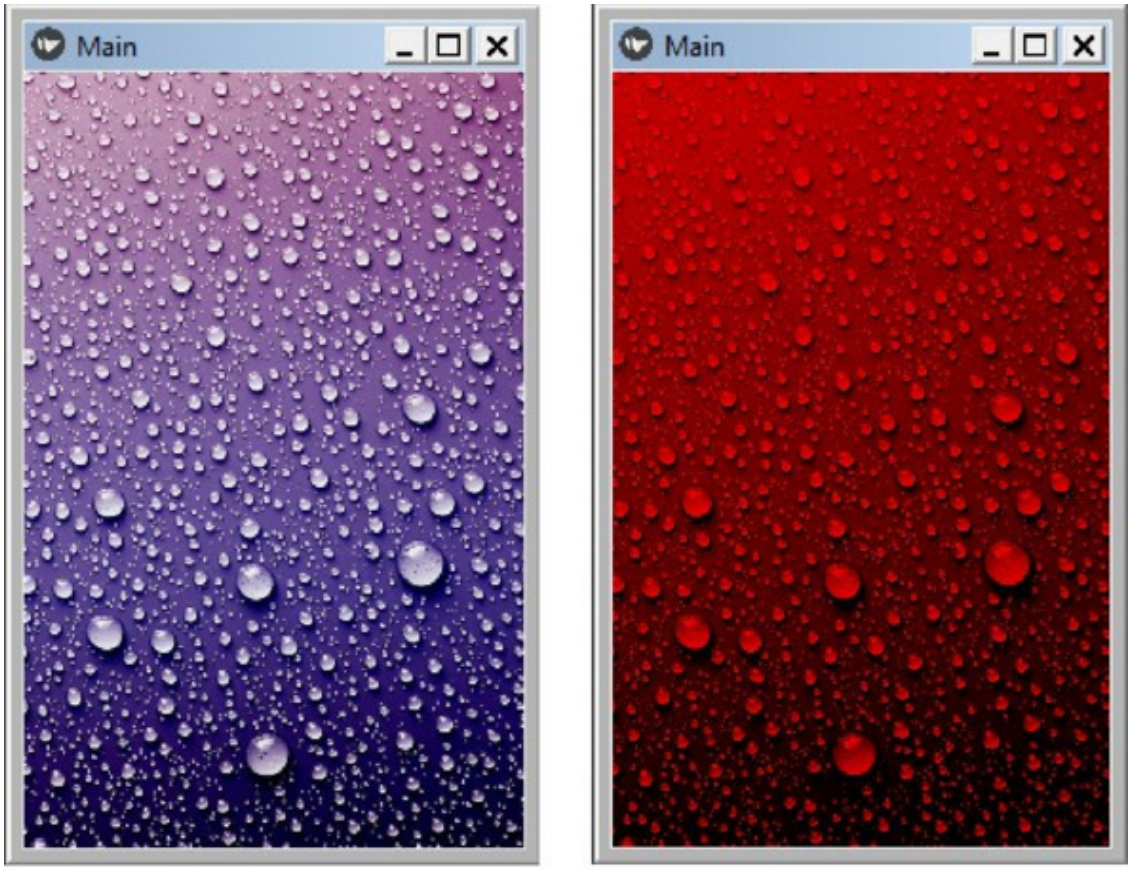


Рис. 2.31. Результаты выполнения приложения из модуля K_Widget _4.py

Как видно из данного рисунка, инструкции объекта color распространяется на все изображение.

Объект Widget имеет ряд свойств, которые позволяют задать и получить некоторые параметры:

– canvas – встроенный объект, имеющий инструкции (**важно** – пишется с маленькой буквы);

– Color – инструкция для задания цвета виджета (**важно** – пишется с большой буквы);

– rgba – свойство (цвет виджета), задается в формате r, g, b, a;

– Rectangle – инструкция для задания свойств рамки виджета (**важно** – пишется с большой буквы);

– source – источник (путь и имя файла с изображением, которое можно поместить в рамку);

– size – размер (указывается – self.size, иметь размер рамки, как у родительского виджета);

– pos – положение (указывается – self. pos, иметь положение рамки, как у родительского виджета).

Итак, в данном разделе мы познакомились с основными виджетами фреймворка Kivy. Реализовали простейшие примеры, в которых показано, как можно создать визуальный эле-

мент интерфейса, используя только Python, и с совместным использованием Python и языка KV. В этих примерах не были показаны ни особенности синтаксиса языка KV, ни способы формирования самого интерфейса из набора виджетов. Для того, чтобы поместить тот или иной визуальный виджет в определенную область окна приложения используется набор специальных виджетов, обеспечивающих позиционирование элементов интерфейса. Имена этих виджетов начинаются с префикса Layout (размещение, расположение, расстановка). Эти виджеты не видны в окне приложения, это просто контейнеры, в которых определенным образом размещаются видимые виджеты. Виджеты – контейнеры позволяют стоить дерево виджетов. Поэтому прежде чем перейти к знакомству с виджетами – контейнерами, разберемся со способами и особенностями формирования дерева виджетов.

2.6. Правила работы с виджетами в Kivy

2.6.1. Задание размеров и положения виджетов в окне приложения

Виджеты в Kivy по умолчанию заполняют все окно приложения и располагаются в его центре. Однако они имеют еще ряд свойств, благодаря которым, виджету можно задать размер и поместить в разные области окна приложения.

Рассмотрим на примере кнопки Button, как можно задать ей размер и расположить в разных местах главного экрана. Создадим файл с именем Button1.py и напишем в нем следующий код (листинг 2.42).

Листинг 2.42. Задание параметров виджету Button – размер и положение (модуль Button1.py)

```
# модуль Button1.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Button:
…… text: «Это кнопка»
…… size_hint:.5,.5
…… # – — – — – — – — – —
…… #size_hint:.8,.5
…… #size_hint:.5,.8

…… pos_hint: {’center_x’:.5, ’center_y’:.5}
…… # – — – — – — – — – — – — – — – — – — – — – —
…… #size_hint:.2,.1
…… #pos_hint: {’center_x’:.15, ’center_y’:.5}
…… #pos_hint: {’center_x’:.85, ’center_y’:.5}
…… #pos_hint: {’center_x’:.5, ’center_y’:.15}
…… #pos_hint: {’center_x’:.5, ’center_y’:.85}
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

Здесь в текстовой строке KV создан виджет – Button (кнопка). Для данного виджета заданы следующие свойства:

– text – надпись на кнопке

– size_hint – размер кнопки;

– pos_hint – положение кнопки в окне приложения.

Если с надписью на кнопке все понятно (свойству text присваивается значение «Это кнопка»). То какой смысл имею следующие два свойства кнопки и их параметры (size_hint и pos_hint). Разберемся с этим вопросом.

Пока рассмотрим две рабочие строки (на которых нет знака комментария «#»):

– size_hint:.5,.5;

– pos_hint: {'center_x':.5, 'center_y':.5}.

Свойство кнопки size_hint определяет ее размер по горизонтали (ширина – x) и вертикали (высота -y). Но это не абсолютный, а относительный размер. Если принять размер окна приложения за единицу – 1 (или за 100%), то размер кнопки в нашем случае будет составлять 0.5 (или 50%) от размера окна по ширине и по высоте.

Свойство кнопки pos_hint определяет ее положение в окне приложения, но так же не в абсолютных, а в относительных единицах. По аналогии, если принять размер окна приложения за единицу – 1 (или за 100%), то в этом примере положение центра кнопки будет расположено в точке, составляющей 0.5 (или 50%) от размера окна по горизонтали (по оси «x»), и 0.5 (или 50%) от размера окна по вертикали (по оси «y»).

После запуска данного приложения получим следующий результат (рис.2.32).

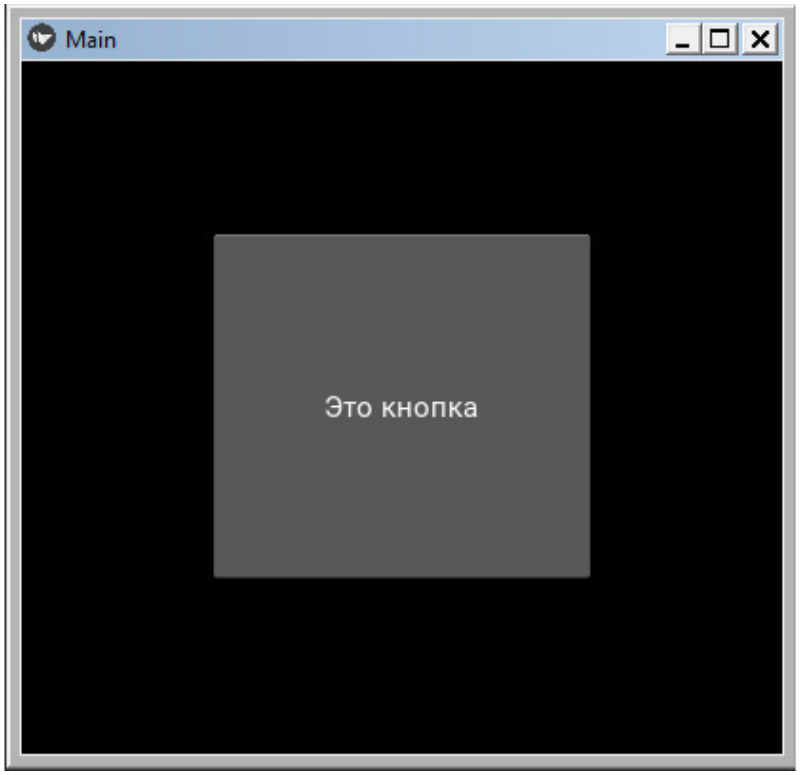


Рис. 2.32. Окно приложения Button1.py с кнопкой в центре окна

Вместо положения центра элемента, можно указать положение его левого нижнего угла. Для этого вместо параметров {'center_x':.5, 'center_y':.5}, нужно указать {'x':.5, 'y':.5}.

Создадим файл с именем Button1_1.py и напишем в нем следующий код (листинг 2.43).

Листинг 2.43. Задание параметров виджету Button – размер и положение (модуль Button1_1.py)

```
# модуль Button1_1.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Button:
…… text: «Это кнопка»
…… size_hint:.5,.5
…… pos_hint: {'x':.5, 'y':.5}
```

```
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

В результате его выполнения получим следующий результат (рис.2.33).

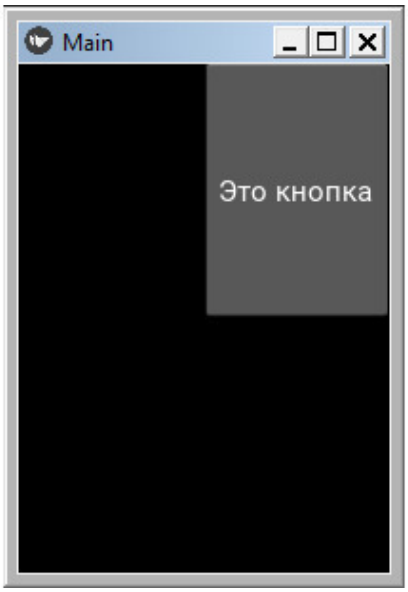


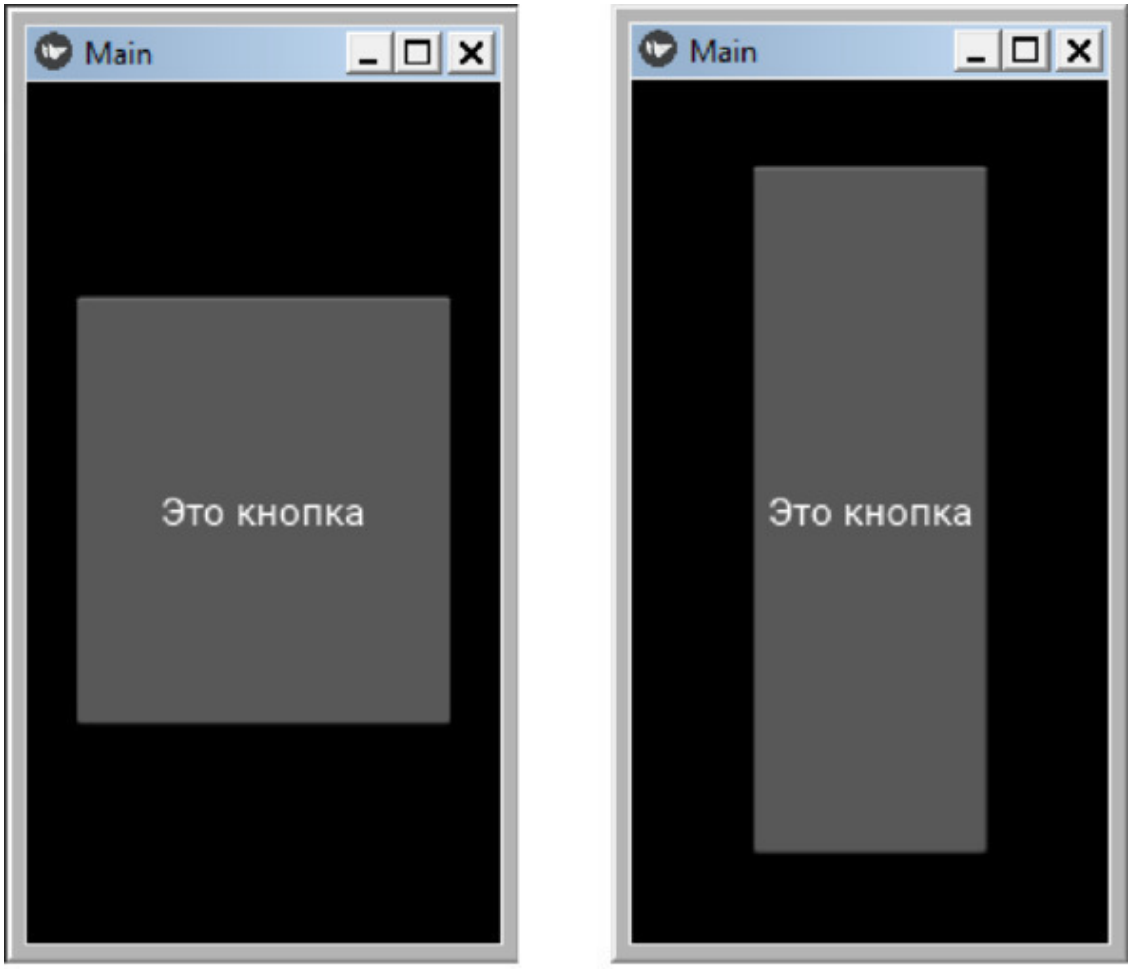
Рис. 2.33. Окно приложения Button1_1.py с кнопкой (левый нижний угол в центре окна)

Почему в Kivy задаются не абсолютные, а относительные размеры и положение виджетов? Это обеспечивает автоматическую расстановку виджетов в окне приложения, при его запуске на разных устройствах с различным размером и разрешением экранов. При этом будут сохранены все пропорции между размерами и расположением виджетов. Таким образом, программисту не нужно адаптировать приложение для различных устройств. Интерфейс будет корректно выглядеть и на смартфоне, и на планшете, и на настольном компьютере. Однако, если мы планируем создавать приложения для мобильных устройств, то интерфейс пользователя необходимо строить с учетом пропорции и размеров экранов мобильных устройств.

Теперь поэкспериментируем с закомментированными строками. Попробуем изменить размеры кнопки, для этого достаточно переназначить значения свойства size_hint (закомментированные строки):

```
#size_hint:.8,.5
#size_hint:.5,.8
```

В первой задали размер кнопки по горизонтали – 0.8, во второй размер кнопки по вертикали – 0.8. Запусти приложение, поочередно меняя комментарии в этих строках. Результаты работы программы представлены на рис. 2.34.



size_hint: .8, .5 size_hint: .5, .8

Рис. 2.34. Окно приложения Button1.py при разных параметрах размера кнопки

Итак, на примере кнопки (Button) мы показали, как в Kivy с помощью относительных параметров можно задавать размеры виджета.

Теперь путем настройки свойств кнопки изменим ее положение в окне приложения. Для этого достаточно изменить свойство pos_hint.

Следует иметь в виду, что в KV началом координат (x, y) является левый нижний угол окна приложения. Уменьшим размер кнопки (size_hint:.2,.1) поместим ее в разные места окна приложения, для чего будем снимать комментарии со следующих строк программы:

```
#size_hint:.2,.1
#pos_hint: {'center_x':.15, 'center_y':.5}
#pos_hint: {'center_x':.85, 'center_y':.5}
#pos_hint: {'center_x':.5, 'center_y':.15}
#pos_hint: {'center_x':.5, 'center_y':.85}
```

Запустим приложение несколько раз, поочередно меняя комментарии в этих строках, и посмотрим на результаты (рис.2.35):

Рис. 2.35. Положение кнопки в различных частях окна приложения

При этом в Kivy имеется возможность задавать не только относительные, но и абсолютные значения параметров. Для этого используются следующие свойства:

– size_hint: None, None – отменить использование автоматической перерисовки элемента (подгонку под размер родительского виджета);

– size – абсолютный размер элемента в пикселах, например, 150, 50 (150 – ширина элемента, 50 – высота элемента);

– pos – абсолютная позиция элемента в окне приложения в пикселах, например, 140, 40 (140 – координата по оси x, 40 – координата по оси y).

Рассмотрим на примере кнопки Button, как можно задать ей абсолютный размер и расположить в указанное место экрана. Создадим файл с именем Button2.py и напишем в нем следующий код (листинг 2.44).

Листинг 2.44. Задание абсолютных параметров виджету Button – размер и положение (модуль Button2.py)

```
# модуль Button2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Button:
……text: «Кнопка»
…… size_hint: None, None
…… size: 150, 50
…… pos: 100, 50
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

В этой программе мы создали кнопку Button и задали ей абсолютные размеры ширина – 150 пикселей, высота – 50 пикселей, и поместили ее в следующие координаты окна (x= 100, y=50). Кроме того, в строке «size_hint: None, None» мы отменили автоматическое растягивание кнопки в размеры окна приложения.

Примечание.

В приложениях на Kivy нулевой координатой окна приложения (x=0, y=0) является левый нижний угол.

После запуска приложения получим следующий результат (рис.2.36).

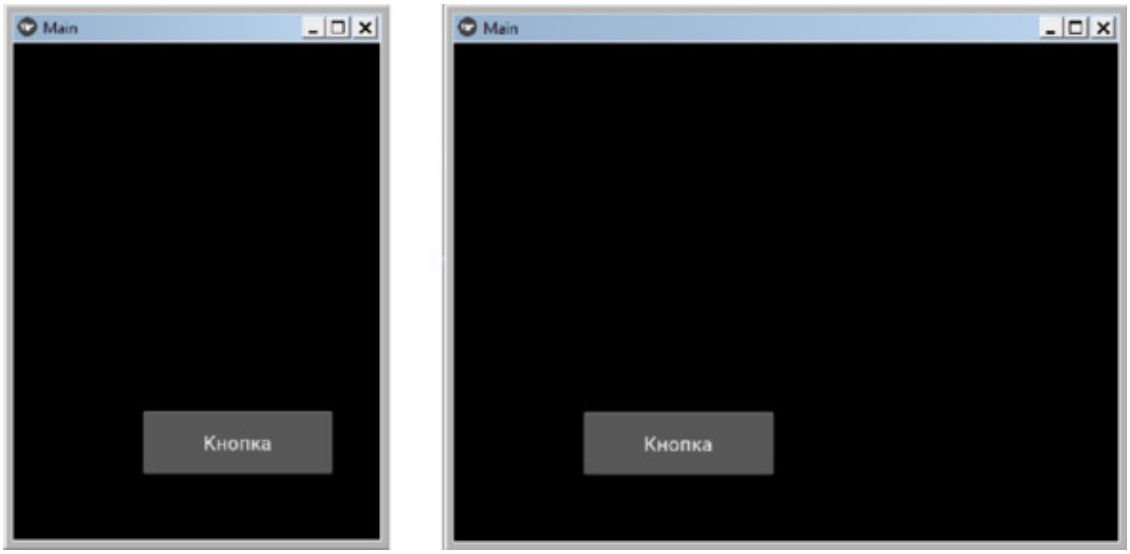


Рис. 2.36. Использование абсолютные значений параметров для задания размера и положения элемента в окне приложения

Итак, мы разобрались с очень важной частью использования Kivy для разработки интерфейса – заданием размеров и позиционирования визуальных элементов на основе относительных и абсолютных параметров.

Подводя итог, напомним, какие свойства используются для задания размеров и положения виджетов:

– text – надпись на элементе;

– size_hint – относительный размер элемента (например, size_hint:.5,.5);

– pos_hint – относительное положение элемента в окне приложения (например, центра – pos_hint: {'center_x':.5, 'center_y':.5} или левого нижнего угла – pos_hint: {'x':.5, 'y':.5}).

– size_hint: None, None – отменить использование автоматической перерисовки элемента (подгонку под размер родительского виджета);

– size – абсолютный размер элемента в пикселах, например, size: 150, 50 (150 – ширина элемента, 50 – высота элемента);

– pos – абсолютная позиция элемента в окне приложения в пикселах, например, pos: 140, 40 (140 – координата по оси x, 40 – координата по оси y).

2.6.2. Задание виджетам цвета фона

В этом разделе мы узнаем, как изменить цвет фона на примере кнопки. В Kivy существует свойство с именем background_color. Это свойство определяет одно цветовое значение, которое используется для изменения цвета фона элемента.

По умолчанию цвет кнопки серый, если необходимо его изменить, то используется это свойство. Для получения чистого цвета RGB (красный, зеленый, синий) параметры этого свойства должны принимать значение от 0 до 1 (например, background_color:1,0,0,1 – красный цвет, 0,1,0,1 – зеленый цвет, 0,0,1,1 – синий цвет).

В интернете на ряде сайтов можно найти информацию, что эти параметры могут принимать только значение от 0 до 1, и любое другое значение приведет к некорректному поведению программы. Это, скорее всего, имеет отношение к одной из старых версий документации. К настоящему моменту разработчики внесли некоторые изменения в программный код своих функций, и эти величины могут принимать значения, отличные от 0 и 1, что обеспечивает возможность получать весь спектр цветов.

Рассмотрим это на примере изменения цвета кнопок. Создадим файл Button_Color.py и напишем там следующий программный код (листинг 2.45).

Листинг 2.45. Задание цвета кнопкам через свойство background_color (модуль Button_Color.py)

```
# модуль Button_Color.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
GridLayout:
…… cols: 3
…… rows: 2

…… Button:
…… …… text: «Красный»
…… …… background_color: 1, 0, 0, 1
…… Button:
…… …… text: «Зеленый»
…… …… background_color: 0, 1, 0, 1
…… Button:
…… …… text: «Синий»
…… …… background_color: 0, 0, 1, 1
…… Button:
…… …… text: «Черный»
…… …… background_color: 0, 0, 0, 1
…… Button:
…… …… text: «Белый»
…… …… color: 0,0,0,1
…… …… background_normal:»»
…… Button:
…… …… text: «Бирюзовый»
…… …… background_color: 102/255, 255/255, 255/255, 1
«»»
```

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

MainApp().run ()
```

Здесь мы создали таблицу из трех колонок и двух строк, в которую разместили 6 кнопок. Каждой кнопке задали свой цвет.

Примечание.

Обратите внимание, что для задания белого цвета фона используется другое свойство – «background_normal:».

Поскольку на белом фоне не будет видна надпись белого цвета, то для текста, который выводится на этой кнопке, был задан черный цвет (color: 0,0,0,1). Для задания бирюзового цвета использовалось значение параметра «102/255, 255/255, 255/255, 1». Дело в том, что в таблице цветов RGB бирюзовый цвет имеет значение «102, 255, 255». В текущей версии Kivy параметры этого цвета можно задать простым деление этих значений на число 255.

Для всех цветов последнее (четвертое) значение параметра цвета равно 1. Это, по сути, значение альфа маски (слоя прозрачности) для четырехканальных изображений (четыре канала используются в файлах «.png» для хранения изображений). Значение альфа маски всегда находится в пределах 0—100% (или от 0 до 1). При значении 1 будет получен «чистый» цвет (маска прозрачная), 0 – черный цвет (маска не прозрачная), промежуточные значения между 0—1 (полупрозрачная маска) будут давать заданный цвет с разной степенью яркости (затененности). Здесь мы задали значение данного параметра 1. Необходимо следить за очередными версиями Kivy, поскольку в документации может появиться информация об изменениях способов задания цвета.

Результаты работы этой программы представлены на рис. 2.37.

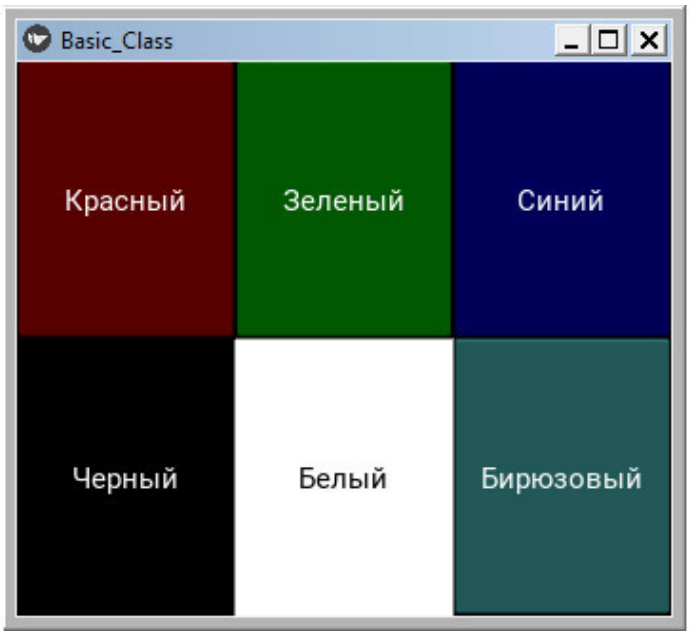


Рис. 2.37. Изменение цвета кнопок с использованием свойства background_color

Итак, мы познакомились с возможностью задавать цвета визуальному виджету с использованием свойства background_color: r, g, b, a (например, background_color: 1, 0, 0, 1).

2.6.3. Обработка событий виджетов

Как и многие другие инструменты для разработки пользовательского интерфейса, Kivy полагается на события. С использованием данного фреймворка можно реализовать отклик на касание клавиш, на касание кнопок мыши или прикосновения к сенсорному экрану. В Kivy задействован концепт часов (Clock), что дает возможность создать отложенный вызов функций через заданное время.

В Kivy реализовано два способа реагирования на события:

– явное связывание визуального элемента с заданной функцией;

– неявное связывание визуального элемента с заданной функцией.

Рассмотрим обе эти возможности. Для явного связывания визуального элемента с заданной функцией создадим новый файл Button_Otklik1.py и внесем в него следующий код (листинг 2.46).

Листинг 2.46. Явное связывание визуального элемента с функцией отклика на действия пользователя (модуль Button_Otklik1.py)

```
# модуль Button_Otklik1.py
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 ()
```

Здесь в базовом классе мы реализовали две функции:

– в первой (def build) мы создали кнопку, поместили ее в центре окна приложения и связали событие нажатие кнопки (on_press) с функцией – press_button;

– во второй функции (def press_button) мы прописали действия, которые необходимо выполнить при касании кнопки (в качестве такого действия задан вывод в терминальное окно сообщения ««Вы нажали на кнопку!»).

После запуска данного приложения мы получим следующее окно (рис.2.38).

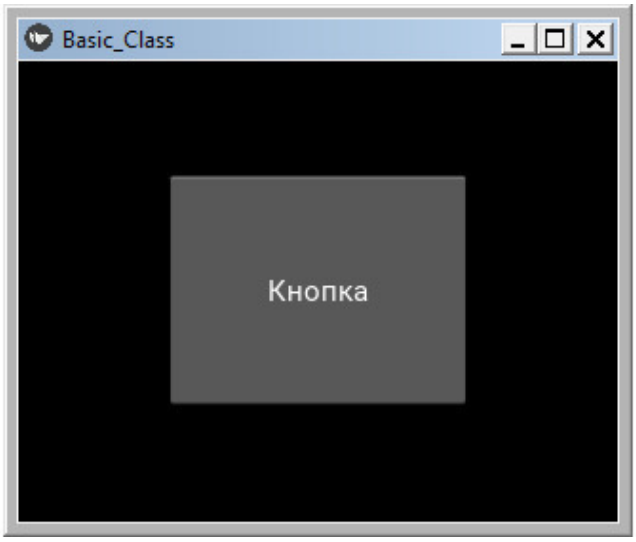


Рис. 2.38. Окно приложения с кнопкой, выполняющей запрограммированные действия

Теперь каждый раз, когда пользователь будет нажимать кнопку (касаться кнопки), в окне терминала будет появляться сообщение – «Вы нажали на кнопку!» (рис.2.39).

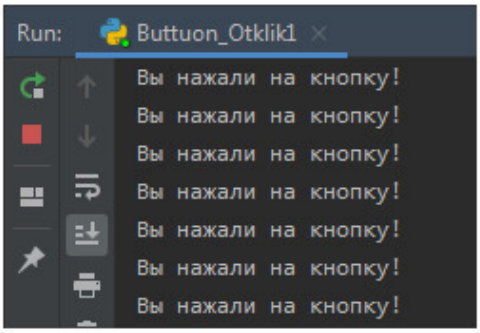


Рис. 2.39. Окно терминала с результатами действия при нажатии на кнопку

В данном примере модуль был создан в коде на языке Python. А сейчас реализуем тот же пример с использованием языка KV. Для этого создадим файл с именем Button_Otklik11.py и напишем в нем следующий код (листинг 2.47).

Листинг 2.47. Явное связывание визуального элемента с функцией отклика на действия пользователя (модуль Button_Otklik11.py)

```
# модуль Button_Otklik11.py
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 ()
```

Здесь в строковой переменной KV обрабатывается событие нажатия кнопки (on_press). При возникновении данного события выполняется обращение к функции приложения press_button, которая находится в корневом модуле (root). Результаты работы приложения будут такими же, как представлено на двух рисунках выше.

На языке Kivy достаточно просто организована обработка событий:

```
«событие: функция обработки события»
```

Например, у кнопки имеются зарезервированное событие – on_press (касание кнопки). Если обработка этого события реализуется непосредственно в коде на KV, то это делается следующим образом:

```
Button:
…… on_press: print («Кнопка нажата»)
```

Если обработка события реализуется в разделе приложении, написанном на Python, то можно использовать следующий код:

```
# это код на KV
Button:
on_press: app.press_button (args)

# это код на Python
def press_button (self):
print («Вы нажали на кнопку!»)
```

Для неявного связывания визуального элемента с заданной функцией создадим новый файл Button_Otklik2.py и внесем в него следующий код (листинг 2.48).

Листинг 2.48. Неявное связывание визуального элемента с функцией отклика на действия пользователя (модуль Button_Otklik2.py)

```
# модуль Button_Otklik2.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 () # приложение на основе базового класса
My_App.run () # запуск приложения
```

В данном коде создана кнопка button на основе базового класса Button, но эта кнопка не имеет связи с функцией обработки события ее нажатия, хотя сама функция press_button присутствует.

С первого взгляда данный код может показаться странным, так как кнопка button не связана с функций реакции на событие нажатия кнопки. Такую связку можно реализовать на уровне языка KV. Вспомним, что при запуске головного модуля Kivy автоматически ищет файл с таким же названием, что и у базового класса (в данном случае файл – basic_class1.kv), и выполняет запрограммированные там действия. Найдем в своем проекте (или создадим) файл с именем basic_class1.kv и внесем в него следующий программный код (листинг 2.49).

Листинг 2.49. Содержание файла basic_class1.kv (модуль basic_class1.kv)

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

Иными словами мы связь отклика на нажатия кнопки перенесли из основного модуля, в связанный модуль на языке KV. Если теперь запустить программу на выполнение, то мы получим тот же результат, что и в предыдущем программном модуле.

2.7. Дерево виджетов – как основа пользовательского интерфейса

В приложениях на Kivy – ***пользовательский интерфейс строится на основе дерева виджетов***. Принципиальная структура дерева виджетов приведена на рис.2.40.

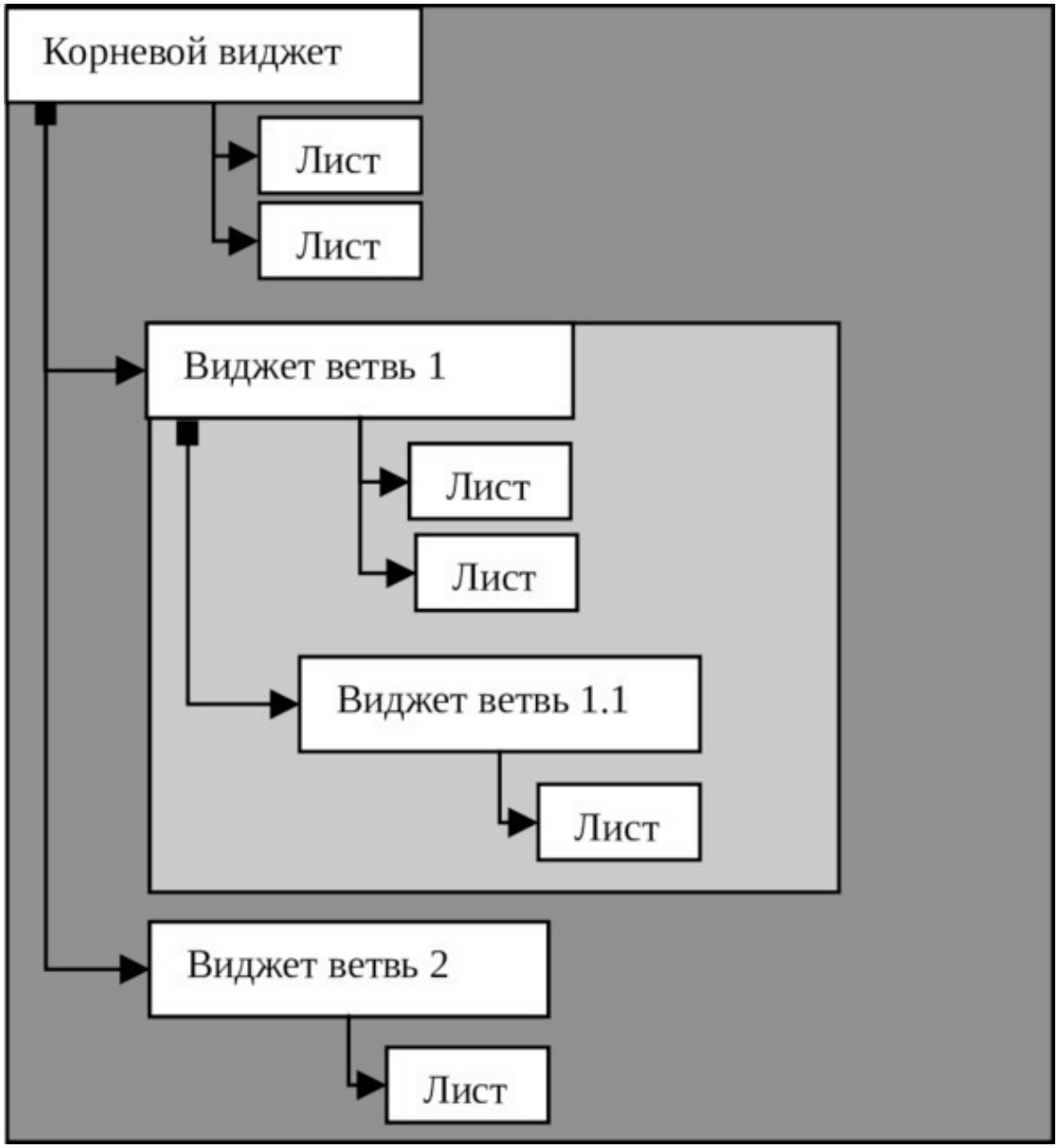


Рис. 2.40. Структура дерева виджетов

Основой дерева виджетов является «Корневой виджет». Это, по сути, контейнер, в котором находятся дочерние элементы, или виджеты ветки. На приведенном выше рисунке, в корневом виджете – контейнере имеется две ветки («Виджет ветвь 1» и «Виджет ветвь 2»). В свою очередь, каждая из этих веток может иметь свои ответвления. В частности «Виджет ветвь 1», так же является контейнером, в котором находится «Виджет ветвь 1.1». Каждая ветка дерева может иметь «листья». Лист – это конечный элемент в дереве виджетов. Каждый лист – это свойство виджета, которое имеет параметр с заданным значением. Кроме свойства «лист»

может содержать и метод, связанный с обработкой события (ссылка на функцию, в которой будет обработано событие, касающееся данного виджета). Структура дерева виджетов может быть сформирована как с использованием языка KV, так и на языке Python.

В приложении может быть только один корневой виджет и любое количество веток, или дочерних виджетов. Виджет представляет собой объект, созданный на базе одного из базовых классов фрейморка Kivy. Базовые классы фрейморка Kivy, на которых можно построить пользовательский объект, можно разделить на две категории:

– классы для создания видимых виджетов (они отображаются в окне приложения);

– классы для создания невидимых виджетов (они указывают положение видимых виджетов в окне приложения).

В литературе можно встретить различное наименование невидимых виджетов: контейнер, макет, виджет позиционирования, виджет Layout.

При построении дерева виджетов на языке KV каждая последующая ветка в программном коде отделяется от предыдущей ветки с помощью отступов. Корневой виджет всегда начинается с первого символа в редакторе программного кода. Каждая последующая ветвь дерева виджетов имеет отступ в 4 символа и начинается с пятого символа в редакторе программного кода. Например:

Корневой виджет:
…… Дочерний виджет 1:
…… …… Дочерний виджет 1.1:
…… Дочерний виджет 2:
…… …… Дочерний виджет 2.1:
…… …… …… Дочерний виджет 2.1.1:
…… …… …… Дочерний виджет 2.1.2:

В редакторе программного кода такой отступ можно создать с использованием клавиши «Tab».

Виджеты в Kivy организованы в виде деревьев. В любом приложении должен быть один корневой виджет, который обычно имеет дочерние виджеты. Дерево виджетов для приложения можно построить и на языке KV, и на языке Python.

На языке Python дерево виджетов можно формировать с помощью следующих методов:

– add_widget (): добавить виджет к родительскому виджету в качестве дочернего;

– remove_widget (): удалить виджет из списка дочерних элементов;

– clear_widgets (): удалить все дочерние элементы из родительского виджета.

Например, если необходимо добавить кнопку в контейнер BoxLayout, то это можно сделать последовательностью следующих команд:

```
layout = BoxLayout (padding=10) # Создать контейнер
button = Button (text=«Кнопка») # создать кнопку
layout.add_widget (button) # положить кнопку в контейнер
```

Для демонстрации этой возможности создадим файл с именем K_Treet_1.py и напишем в нем следующий код (листинг 2.50).

Листинг 2.50. Пример создания дерева виждетов на Python (модуль K_Tree_1.py)

```
# модуль K_Tree_1.py
from kivy. app import App
from kivy.uix.boxlayout import BoxLayout
from kivy. uix. button import Button
from kivy.uix.screenmanager import Screen
```

```
class MainApp (App):
…… def build (self):
…… …… scr = Screen () # корневой виджет (экран)
…… …… box = BoxLayout () # контейнер box
…… …… but1 = Button (text=«Кнопка 1») # кнопка 1
…… …… but2 = Button (text=«Кнопка 2») # кнопка 2
…… …… box.add_widget (but1) # положить кнопку 1 в контейнер
…… …… box.add_widget (but2) # положить кнопку 2 в контейнер
…… …… scr.add_widget (box) # положить контейнер в корневой виджет
…… return scr

MainApp().run ()
```

В этом модуле мы создали корневой виджет scr (экран) на основе базового класса Screen. Затем создали контейнер box на основе базового класса BoxLayout. После этого создали две кнопки but1 и but2 на основе базового класса Button. На следующем этапе эти кнопки положили в контейнер, а сам контейнер поместили в корневой виджет. После запуска данного приложения получим следующий результат (рис.2.41).

Рис. 2.41. Результаты выполнения приложения из модуля K_Treet_1.py

Программный код получился достаточно длинным, поскольку для каждого базового класса приходится делать импорт соответствующего модуля.

Аналогичный код на языке KV будет выглядеть гораздо проще и понятней:

```
Screen: # создание корневого виджета (экран)
…… BoxLayout: # создание контейнера BoxLayout
…… Button: # добавление в контейнер виджета Button (кнопка)
…… Button: # добавление в контейнер виджета Button (кнопка)
```

Для демонстрации этого создадим файл с именем K_Treet_2.py и напишем в нем следующий код (листинг 2.51).

Листинг 2.51. Пример создания дерева виждетов на Python (модуль K_Tree_2.py)

```
# модуль K_Tree_2.py
from kivy. app import App
from kivy.lang import Builder

KV = «»»
Screen: # создание корневого виджета (экран)
……BoxLayout: # создание контейнера BoxLayout
…… …… Button: # добавление в контейнер виджета Button (кнопка)
…… …… …… text: «Кнопка 1»
…… …… Button: # добавление в контейнер виджета Button (кнопка)
…… …… …… text: «Кнопка 2»
«»»

class MainApp (App):
…… def build (self):
…… …… return Builder. load_string (KV)

MainApp().run ()
```

После запуска приложения мы получим тот же результат, что и на предыдущем рисунке. При этом сам код стал компактней, поскольку нет необходимости явным образом импортировать модули с базовыми классами (они подгружаются автоматически).

При использовании языка Python при создании элемента можно сразу задать и его свойства. Например, в предыдущих примерах это было сделано следующим образом:

```
but1 = Button (text=«Кнопка 1»)
```

Аналогичный код на языке KV выглядит иначе:

```
Button:
…… text: «Кнопка 1»
```

Примечание.

На языке KV имена виджетов должны начинаться с заглавных букв, а имена свойств – со строчных букв.

Теперь можно более детально познакомиться с виджетами – контейнерами, которые отвечают за размещение видимых элементов интерфейса на экране, и используется для построения дерева виджетов.

2.8. Виджеты для позиционирования элементов интерфейса в приложениях на Kivy

В Kivy имеется набор так называемых «layout» виджетов, или виджетов позиционирования. Это особый вид виджетов, которые контролируют размер и положение своих дочерних элементов. Ниже приводятся краткие характеристики этих виджетов.

AnchorLayout. Это простой макет, заботящийся только о позициях дочерних виджетов. Он позволяет размещать дочерние элементы в позиции относительно границы макета (при этом значение size_hint не соблюдается).

BoxLayout. Размещает дочерние виджеты смежным образом (вертикально или горизонтально), то есть рядом друг с другом, заполняя при этом все свое пространство. Свойство дочерних элементов size_hint (указание размера) можно использовать для изменения пропорций, разрешенных для каждого дочернего элемента, или для установки фиксированного размера для некоторых из них.

FloatLayout. Позволяет размещать дочерние элементы с произвольным расположением и размером (как с абсолютными значениями параметров, так и относительно размера макета).

GridLayout. Размещает дочерние виджеты в Grid (таблица, решетка). Необходимо указать хотя бы одно измерение таблицы (количество строк или столбцов), чтобы kivy мог вычислить размер элементов и их расположение.

PageLayout. Позволяет создать набор страниц с возможностью размещения на них визуальных элементов и организовать смену страниц скроллингом.

RelativeLayout. Ведет себя так же, как FloatLayout, за исключением того, что позиции дочерних элементов относятся к положению внутри контейнера, а не к экрану.

Scatter. Используется для создания интерактивных контейнеров. Элементы, размещенные в данном контейнере можно перемещать, поворачивать и масштабировать двумя пальцами на устройствах с сенсорным экраном. При масштабировании самого виджета элементы, находящиеся в нем, не меняют своих размеров.

ScatterLayoyt. Используется для создания интерактивных контейнеров. Элементы, размещенные в данном контейнере можно перемещать, поворачивать и масштабировать двумя пальцами на устройствах с сенсорным экраном. При масштабировании самого виджета элементы, находящиеся в нем, меняют свои размеры вместе с родительским контейнером.

StackLayout. Размещает дочерние виджеты рядом друг с другом, но с заданным размером элемента в одном из измерений, не пытаясь уместить их во всем пространстве родительского контейнера. Это полезно для отображения дочерних элементов одного и того же заданного размера.

Конец ознакомительного фрагмента.