KIVY FRAMEWORK

TWORZENIE APLIKACJI MOBILNYCH W JĘZYKU PYTHON

WPROWADZENIE

Nazywam się Rafał Kaczor i obecnie jestem testerem oprogramowania w firmie Milo Solutions. W branży IT działam od jakichś 3 lat – wcześniej byłem redaktorem portali technologicznych takich jak Android.com.pl czy Tabletowo.pl.

WPROWADZENIE

Moja przygoda z Pythonem i Kivy rozpoczęła się ponad rok temu. Staram się promować Kivy wśród polskiej społeczności i zachęcać do nauki Pythona. Tworzę też moduły przydatne podczas programowania w Kivy.

Żywię nadzieję, że nie zanudzę Państwa moją prezentacją ;)

DLACZEGO PLATFORMY MOBILNE?

- Każdego dnia aktywowanych jest ponad milion nowych urządzeń z Androidem
- Każdego miesiąca użytkownicy urządzeń z Androidem z ponad 190 krajów dokonują ponad 1,5 miliarda pobrań aplikacji

Google play

DLACZEGO PLATFORMY MOBILNE?

- Apple dotąd sprzedało ponad miliard swoich urządzeń z systemem iOS
- Do stycznia 2015 użytkownicy Apple App Store dokonali ponad 75 miliardów pobrań aplikacji

DLACZEGO PLATFORMY MOBILNE?

- W 2014 roku Apple wypłaciło programistom aplikacji i gier ponad 10 miliardów dolarów, zaś Google – ponad 7 miliardów
- Rynek aplikacji mobilnych cały czas się rozwija, a użytkownicy nieustannie pobierają gry i programy
- Android oraz iOS mają ugruntowaną na rynku pozycję, dzięki czemu jest to stabilny grunt pod inwestycję swojego czasu spędzanego na tworzeniu aplikacji i gier
- Wszystkie te przykłady potwierdzają, że warto tworzyć oprogramowanie dla urządzeń mobilnych, ponieważ jest to szansa na dotarcie do dużej bazy użytkowników i w związku z tym monetyzację projektu

APLIKACJE NATYWNE – ROZTERKI PROGRAMISTY

- By tworzyć wydajne programy dla platform Apple i Google musimy podjąć naukę specyficznego środowiska API oraz języków Java / Objective C / Swift
- Tworząc natywną aplikację dla Androida oraz iOS musimy programować te same rozwiązania dwukrotnie, przez co tracimy cenny czas

APLIKACJE NATYWNE – ROZTERKI PROGRAMISTY

- Projektując design aplikacji na platformy Android oraz iOS jesteśmy zmuszeni do nauki i używania nieintuicyjnych systemów znaczników XML / Storyboard
- Pisząc kod aplikacji częściej zastanawiamy się, jak coś wykonać, żeby w ogóle działało, aniżeli jak wykonać to tak, by użytkownik odczuwał przyjemność korzystania z aplikacji

PYTHON A RYNEK MOBILNY

- Obecnie żadna z wymienionych platform – ani Android, ani iOS – nie posiada preinstalowanego interpretera języka Python
- Pythona uważa się za język wolny i nadający się jedynie do rozwiązań webowych

PYTHON A RYNEK MOBILNY

- Pythonowi zarzuca się problemy wydajnościowe podczas skomplikowanych obliczeń matematycznych i graficznych
- Python nie posiada wsparcia dużych firm takich jak Google, Apple, Microsoft w kontekście wykorzystania go do produkcji aplikacji mobilnych

PYTHON A RYNEK MOBILNY

Sadząc po powyższych argumentach można zadać sobie pytanie: „dlaczego mielibyśmy programować mobilne aplikacje właśnie w Pythonie?”

PYTHON A RYNEK MOBILNY

- Nie zawsze różnice w wydajności poszczególnych komponentów wykonanych w językach niskiego i wysokiego są tak duże, że musimy wykorzystywać jedynie języki niskiego poziomu
- Python potrafi wykorzystywać wydajne rozszerzenia napisane w języku C

PYTHON A RYNEK MOBILNY

- Czasem zachodzi potrzeba szybszego napisania aplikacji i ewentualnej jej optymalizacji, a tutaj Python sprawdzi się dużo lepiej niż np. C++ czy Java
- Python pozwala rozwiązywać pewne problemy w relatywnie krótszym czasie niż języki niższego poziomu

PYTHON A RYNEK MOBILNY

- Część programistów Pythona mogłaby tworzyć wspaniałe i wydajne aplikacje w języku który doskonale znają, zamiast uczyć się latami nowych rozwiązań, w których czasem trudno się odnaleźć
- Z punktu widzenia użytkowników im więcej aplikacji jest dostępnych do wyboru tym lepiej. Zwykłego użytkownika nie obchodzi w jakim języku jest napisany program, jeśli pomaga on mu rozwiązywać problemy życia codziennego

PYTHON A RYNEK MOBILNY

- Argument najbliższy naszej społeczności: Python jest po prostu intuicyjnym i wygodnym językiem programowania...
- ...no dobrze, najlepszym ;)

python™

KIVY FRAMEWORK

KIVY FRAMEWORK

Kivy jest frameworkiem służącym do tworzenia multiplatformowych aplikacji korzystających z naturalnych interfejsów użytkownika (NUI) oraz różnych metod wprowadzania danych i interakcji z użytkownikiem. Narzędzie zostało napisane w całości w Pythonie i pozwala na produkcję oprogramowania z użyciem tego języka.

KIVY FRAMEWORK

Kivy korzysta z licencji MIT i jest oprogramowaniem typu Open Source. Kod całego projektu można znaleźć na GitHubie.

W związku z tym Kivy pozwala na wszelkie komercyjne zastosowania bez konieczności dokupywania specjalnych licencji i zezwoleń.

KIVY FRAMEWORK

Kivy Framework jest przede wszystkim narzędziem do tworzenia aplikacji multiplatformowych. Raz napisany program może zostać uruchomiony na:

- platformach desktop: Microsoft Windows, Apple Mac OS X, dystrybucjach Linuksa (np. Ubuntu)
- platformach mobilnych: Google Android, Apple iOS
- platformach embedded: Raspberry Pi

CECHY KIVY FRAMEWORK

- Multiplatformowość (Windows, Linux, OS X, Android, iOS, Raspberry Pi)
- Natywne wsparcie różnych metod interakcji i wprowadzania danych (ekrany dotykowe z multidotykiem, klawiatury, myszy, touchpady, kontrolery gier, zewnętrzne rysiki i tablety graficzne, TUIO)
- Wsparcie dla akceleracji graficznej z użyciem OpenGL ES 2.0

CECHY KIVY FRAMEWORK

- Dedykowany intuicyjny język „Kivy Language” służący opisywaniu interfejsu użytkownika
- Łatwość tworzenia aplikacji i szybkość działania
- Darmowość oraz otwarte źródła
- Bogata dokumentacja API, dostępność przykładowych programów oraz rozwinięta społeczność

CECHY KIVY FRAMEWORK

- Stabilność oraz profesjonalne zaplecze programistyczne prowadzone przez Kivy Organization
- Gotowość do użytku w projektach komercyjnych bez dodatkowych opłat
- Możliwość korzystania z bogatych zasobów bibliotek Pythona przy tworzeniu oprogramowania (PIL, urllib)

CECHY KIVY FRAMEWORK

- Moduły Kivy wymagające dużej wydajności zostały stworzone w języku C z użyciem Cythona
- Integracja z innymi projektami Kivy Organization, takimi jak: Python-for-Android, Buildozer, Kivy-iOS, PyJNIus, PyOBJus, Plyer, Kivy Designer, pozwalającymi usprawniać produkcję aplikacji na konkretne platformy

CECHY KIVY FRAMEWORK

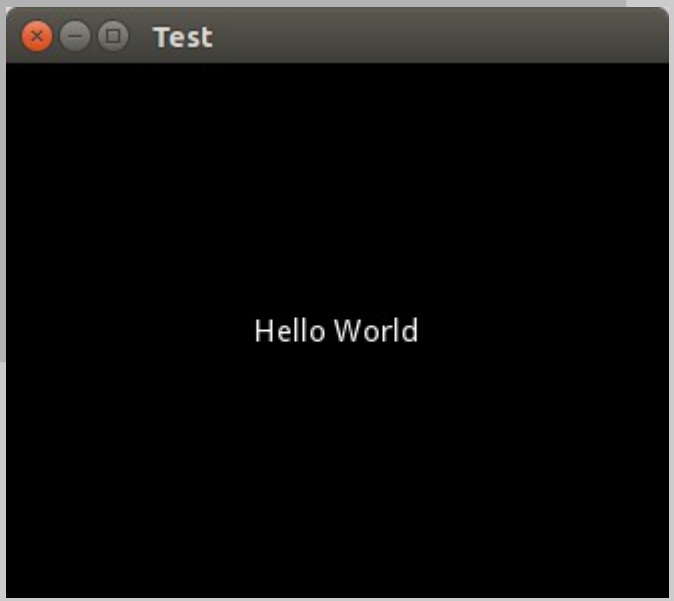
- Kivy Framework obecnie jest dystrybuowane w stabilnej wersji 1.9.0. Kivy wspiera zarówno Pythona w odsłonie 2.7 jak i 3.4. Jeśli chcemy tworzyć aplikacje mobilne, czyli paczkować nasze programy na Androida oraz iOS, jesteśmy zmuszeni używać Pythona 2.7.

SCHEMAT APLIKACJI KIVY

- Poszczególne elementy interfejsu użytkownika (np.. przyciski, etykiety, checkboxy, listy) tworzą tak zwane Widgety
- Widgety mogą być grupowane i automatycznie układane w interfejsie dzięki tak zwanym Layoutom (np. GridLayout tworzy siatkę widgetów)
- Dane w aplikacji są zapisywane w tak zwanych Properties, czyli polach klas które są cały czas obserwowane przez program i ich zmiana może wywołać automatyczną interakcję

„HELLO WORLD”

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

class TestApp(App):
    def build(self):
        return Label(text='Hello World')

TestApp().run()
```



„HELLO WORLD”

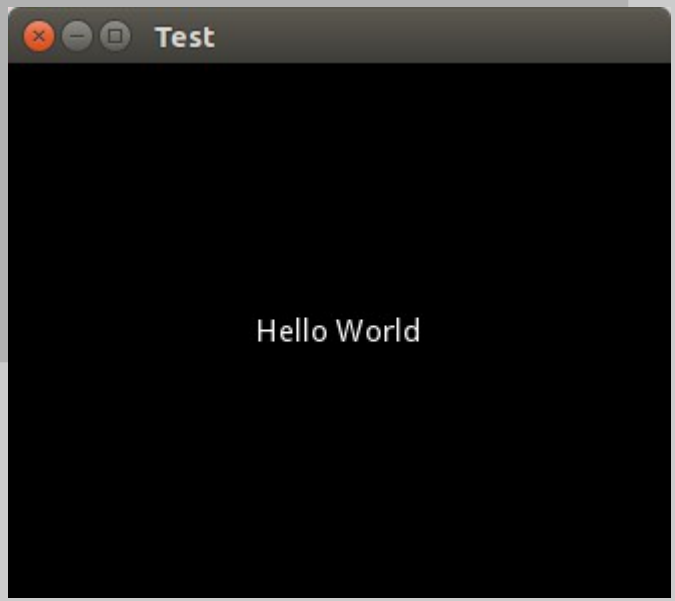
- Z modułu ***kivy.app*** importujemy klasę ***App***
- Z modułu ***kivy.uix.label*** importujemy klasę ***Label***
- Tworzymy klasę naszej aplikacji dziedziczącą z klasy ***App***
- Przeciążamy metodę ***build()*** klasy ***App*** zwracając w niej instancję klasy ***Label*** z przekazanym argumentem ***text*** o wartości ***'Hello World'***
- Uruchamiamy program tworząc instancję klasy aplikacji i wywołując z niej metodę ***run()***

„HELLO WORLD”

Gdy tworzymy aplikację w Kivy Framework musimy pamiętać o dwóch ważnych rzeczach. Mianowicie, zarówno logikę jak i prezentację aplikacji tworzymy w formie drzewa. Wierzchołkiem interfejsu użytkownika jest klasa zwracana we wspomnianej metodzie ***build()*** aplikacji. Wierzchołkiem części logicznej aplikacji jest zaś instancja klasy aplikacji dziedziczącej z klasy ***App***. Odwołania do tych elementów przydadzą się nam w dalszej części tworzenia aplikacji w Kivy.

„HELLO WORLD” Z UŻYCIEM KIVY LANGUAGE

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

root = Builder.load_string('''

Label:
    text: 'Hello World!'


''')

class TestApp(App):

    def build(self):
        return root

TestApp().run()
```



„HELLO WORLD” Z UŻYCIEM KIVY LANGUAGE

Kivy Framework posiada swój dedykowany język służący definiowaniu wyglądu interfejsu użytkownika. Twórcy projektu nadali mu nazwę „Kivy Language”. Język ten zasadą działania przypomina nieco CSS, zaś strukturą kodu samego Pythona. Dzięki Kivy Language bardzo łatwo możemy definiować wygląd elementów aplikacji oraz działanie programu w zależności od interakcji ze strony użytkownika.

„HELLO WORLD” Z UŻYCIEM KIVY LANGUAGE

W podanym przykładzie aplikacja działa i wygląda identycznie jak „Hello World”, które stworzyliśmy stricte programistycznie. Za pomocą klasy ***Builder*** załadowaliśmy do zmiennej ***root*** arkusz stylów naszej aplikacji. W tym przypadku jest to etykieta (***Label***) z polem ***text*** o wartości „Hello World”. Etykieta jest ustawiona jako wierzchołek drzewa widgetów, więc program po starcie prezentuje właśnie ją.

„HELLO WORLD” Z UŻYCIEM KIVY LANGUAGE

Arkusze stylów Kivy Language mogą być załadowane przez aplikację na 3 sposoby:

- Jeśli metoda build() klasy aplikacji nie jest przeciążona, aplikacja stara się załadować arkusz z pliku „nazwaklasyaplikacji.kv”. Jeśli klasę aplikacji nazwiemy TestKivyApp, tak będzie ona próbowała wczytać interfejs z pliku testkivy.kv (nazwa klasy pozbawiana jest członu „App” oraz wszystkie litery są małe)

„HELLO WORLD” Z UŻYCIEM KIVY LANGUAGE

- Możemy arkusz stylów wczytać do zmiennej z pliku o innej nazwie. Wykorzystujemy do tego klasę ***Builder*** i jej metodę ***load_file()***. Otrzymaną zmienną zwracamy w przeciążonej metodzie ***build()***
- Możemy też arkusz stylów podać jako zwykły łańcuch znaków, który metoda ***load_string()*** klasy ***Builder*** przeparsuje i przypisze do jakiejś zmiennej, którą to zwracamy w przeciążonej metodzie ***build()***. Tę właśnie metodę zaprezentowałem w poprzednim przykładzie

MOŻLIWOŚCI KIVY LANGUAGE

- Definiowanie drzewa widgetów, w tym widgetu głównego (root-widgetu)
- Definiowanie klas dynamicznych oraz ich wyglądu
- Przypisywanie działań programu do zmiany wartości zmiennych lub interakcji ze strony użytkownika (bindowanie)
- Separacja interfejsu od logiki

SYNTAX KIVY LANGUAGE

```
# definicja root-widgetu aplikacji
RootWidget:


# definicja stylu klasy dynamicznej RootWidget
# dziedziczącej z klasy GridLayout

<RootWidget@GridLayout>:
    # umieszczenie w RootWidget Widgetu MyButton
    MyButton:
        # definicja akcji wyzwalanej po naciśnięciu przycisku
        on_release: print 'Hello World!'


# definicja stylu klasy dynamicznej MyButton
# dziedziczącej z klasy Button
<MyButton@Button>:
    # przypisanie wartości zmiennych klasy
    size_hint: 1, None
    height: '50dp'
    # ...

<SomeOtherClass>:
    # ...
```

SYNTAX KIVY LANGUAGE

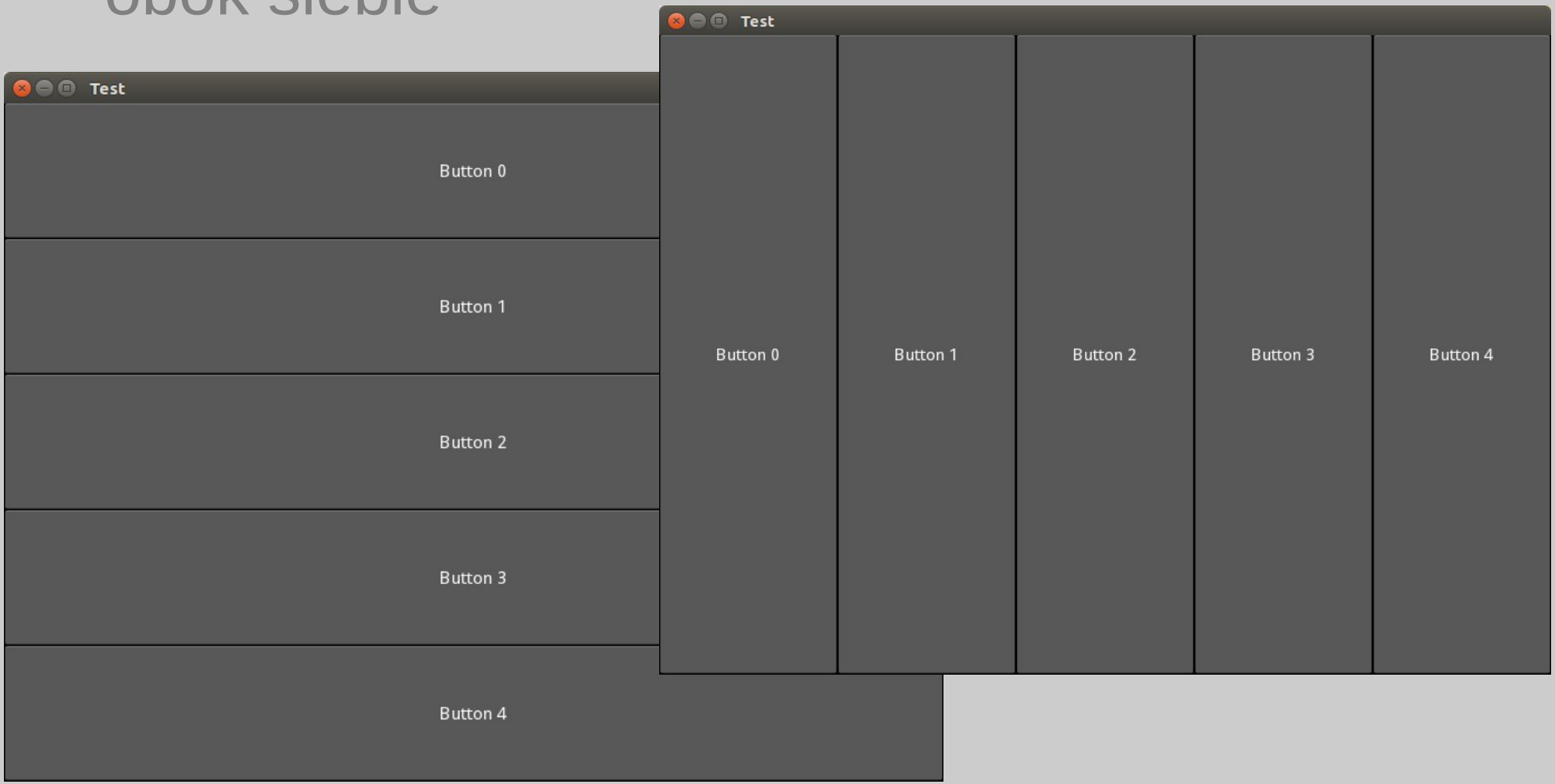
- Kivy Language tak jak Python bazuje na indentacji
- Umieszczenie nazwy klasy w znacznikach „<>” oznacza że definiujemy uniwersalny schemat wyglądu tej klasy, który możemy później wielokrotnie wykorzystać
- Umieszczenie nazwy klasy bez znaczników dodaje Widget do wskazanego drzewa
- Użycie znacznika „@” oznacza dziedziczenie podczas tworzenia klasy dynamicznej. Znak „+” umożliwia dziedziczenie z kilku klas

LAYOUTY W KIVY

- Kivy używa Layoutów do grupowania widgetów lub dokładnego rozmieszczania ich w określonych miejscach okna aplikacji
- Layout także jest Widgetem, więc może być umieszczany w innych Layoutach
- Dzięki Layoutom możemy łatwo tworzyć rozbudowane i responsywne interfejsy użytkownika
- Najpopularniejsze Layouty to: BoxLayout, GridLayout, AnchorLayout, FloatLayout

LAYOUTY W KIVY

- BoxLayout grupuje swoje dzieci pod sobą lub obok siebie

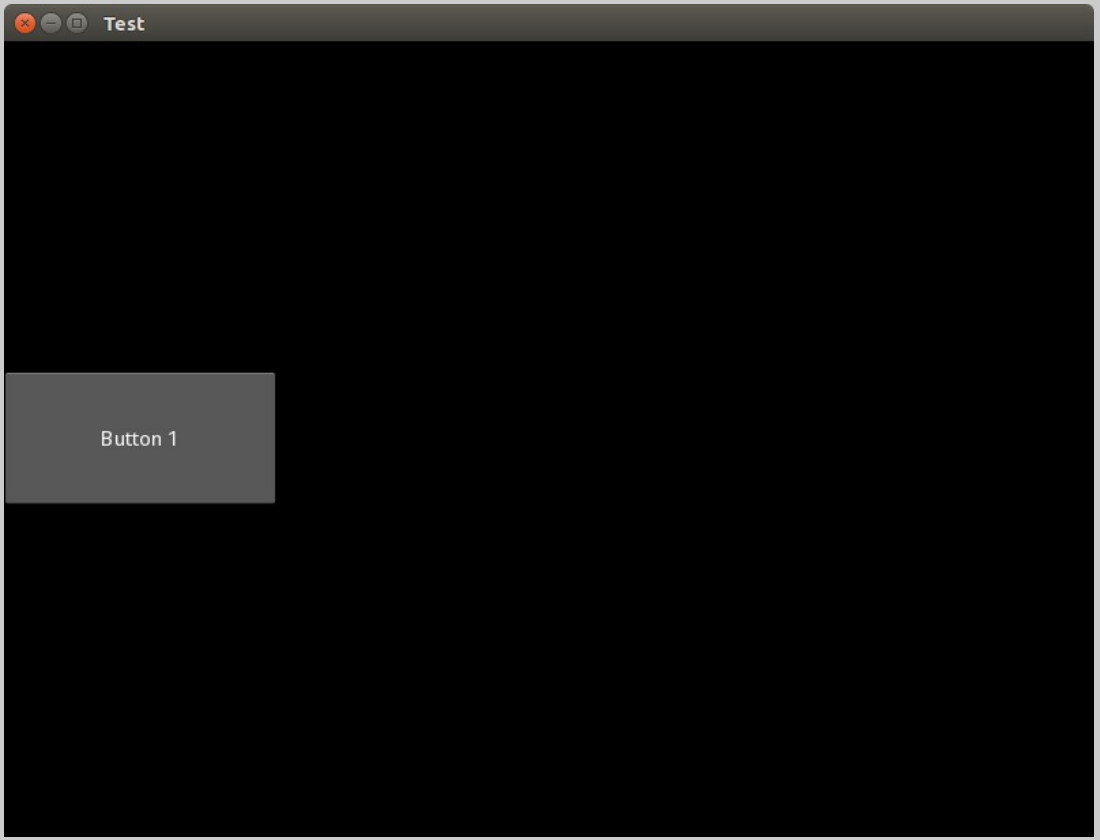


LAYOUTY W KIVY

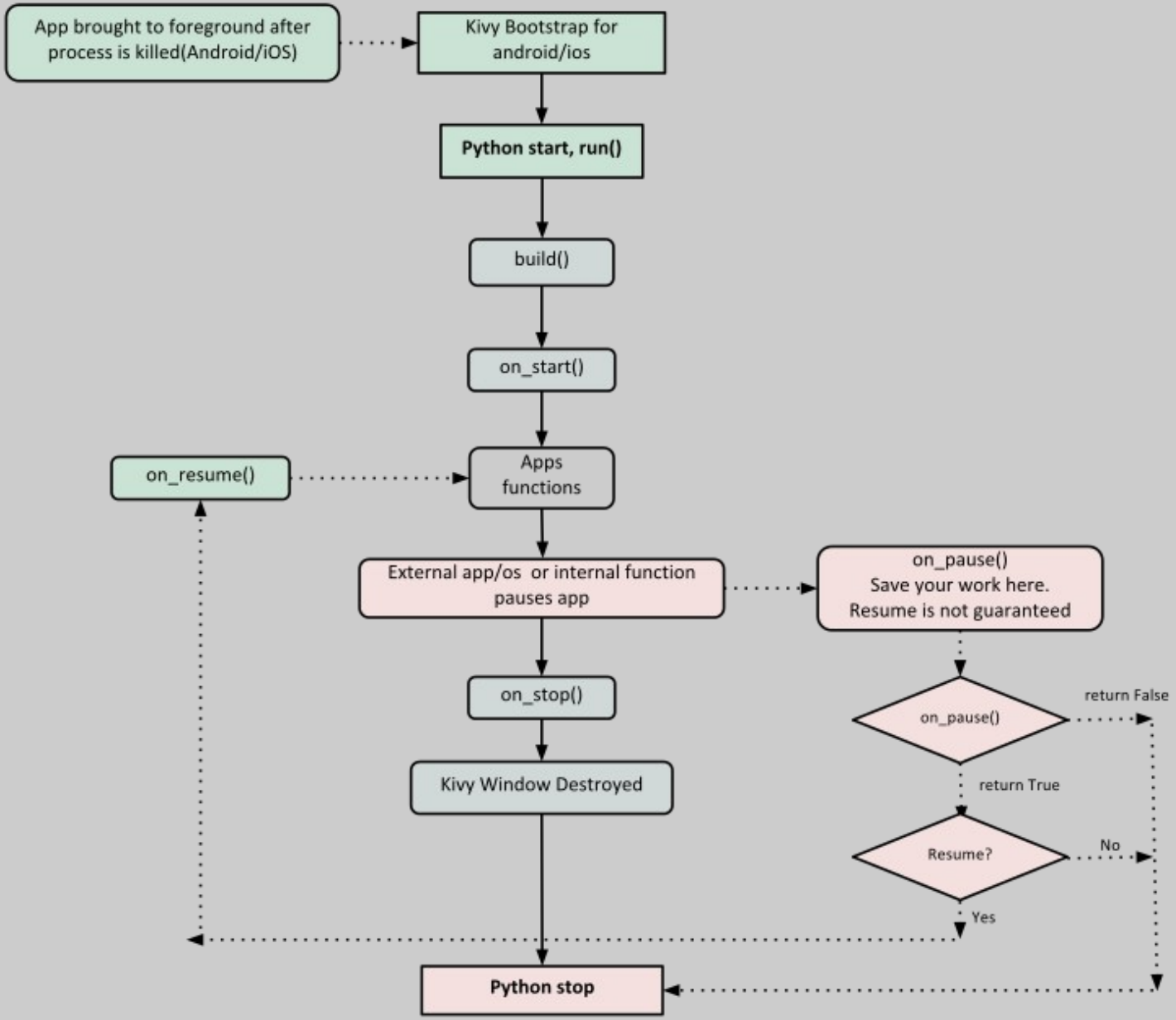
- GridLayout tworzy ze swoich dzieci siatkę

LAYOUTY W KIVY

- AnchorLayout układa elementy na współrzędnych X (lewo, środek, prawo) oraz Y (góra, środek, dół) względem rodzica



CYKL ŻYCIA APLIKACJI KIVY



CYKL ŻYCIA APLIKACJI KIVY

- Aplikacje napisane w Kivy Framework obsługują natywnie zachowania Androida oraz iOS w zakresie pauzowania aplikacji w tle
- Poprzez przeciążenie funkcji ***on_start()***, ***on_stop()***, ***on_pause()*** oraz ***on_resume()*** w klasie aplikacji jesteśmy w stanie kontrolować zachowanie programu podczas uruchamiania czy minimalizacji
- Kivy jest w stanie obsługiwać proces w tle na Androidzie by wykonywać działania nawet wówczas, gdy aplikacja główna jest wyłączona

KIVY PROPERTIES

- Aplikacje w Kivy działają w formie pętli, która cały czas analizuje zachowanie użytkownika, pozwala na cykliczne wykonywanie funkcji programu oraz dba o renderowanie interfejsu
- Główna pętla aplikacji Kivy jest w stanie obserwować niektóre zmienne i w razie zmiany ich wartości wykonywać określone czynności
- Dzięki Kivy Language bardzo łatwo możemy bindować czynności podejmowane przez Kivy Framework do określonych partii interfejsu użytkownika oraz ustalać, z jakich zmiennych korzystać mają Widgety do wyświetlania danych

KIVY PROPERTIES

- Kivy Framework do obsługi określonych typów zmiennych korzysta ze swoich klas, tak zwanych ***Properties***, z pakietu ***kivy.properties***
- Do obsługi typów liczbowych służy klasa ***NumericProperty***, do obsługi łańcuchów znaków ***StringProperty***, do obsługi list ***ListProperty***, do obsługi obiektów ***ObjectProperty***
- Dzięki używaniu Kivy Properties zamiast zwykłych zmiennych języka Python aplikacja może automatycznie odświeżać elementy interfejsu i w czasie rzeczywistym reagować na zmiany czynione przez użytkownika
- Kivy Properties pomagają także ograniczać dostępne zakresy danych, by uniemożliwić użytkownikowi wprowadzenie niepoprawnych wartości

KIVY PROPERTIES

- Aby klasa mogła korzystać z Kivy Properties, powinna w pierwszej kolejności dziedziczyć z klasy ***EventDispatcher***. Wszystkie Widgety w Kivy Framework domyślnie dziedziczą po klasie ***EventDispatcher*** i aktywnie korzystają z Properties
- Zmienne klasy będące obiektami Kivy Properties musimy zadeklarować jako pola klasy, możemy im też wówczas nadać domyślne wartości
- Przy zmianie wartości obiektów Properties traktujemy je jako zwykłe zmienne Pythona

KIVY CLOCK

- Kivy używa klasy ***Clock*** z pakietu ***kivy.clock*** by rejestrować wykonanie pewnych funkcji w przyszłości lub anulować wykonywanie wcześniej zaplanowanych działań
- Klasa ***Clock*** pozwala planować wykonanie funkcji raz oraz cyklicznie w jednakowych odstępach czasu
- Użycie klasy ***Clock*** do planowania działań wymagających czasu gwarantuje że główny wątek nigdy nie zostanie zablokowany, dzięki czemu aplikacja przez cały czas będzie responsywna

```
from kivy.app import App
from kivy.uix.label import Label
from kivy.properties import NumericProperty
from kivy.lang import Builder
from kivy.clock import Clock

class AutoUpdateLabel(Label):

    some_number = NumericProperty(0)

    def __init__(self, **kwargs):
        super(AutoUpdateLabel, self).__init__(**kwargs)
        Clock.schedule_interval(self.update, 1)

    def update(self, *args):
        self.some_number += 5

root = Builder.load_string('''

AutoUpdateLabel:
    text: str(self.some_number)
    font_size: '100dp'

''')

class TestApp(App):

    def build(self):
        return root

TestApp().run()
```

CLOCK I PROPERTIES W DZIAŁANIU

- Importujemy odpowiednie klasy: ***Clock***, ***NumberProperty***, ***Label***
- Tworzymy klasę ***AutoUpdateLabel***, deklarujemy w niej pole ***some_number***, będące obiektem typu ***NumericProperty*** o domyślnej wartości 0
- Tworzymy metodę ***update()***, która z każdym wykonaniem wykona inkrementację wartości zmiennej ***some_number*** o 5
- Używając klasy ***Clock*** planujemy cykliczne wywołanie metody ***update()*** co 1 sekundę

CLOCK I PROPERTIES W DZIAŁANIU

- Za pomocą Kivy Language ustawiamy klasę ***AutoUpdateLabel*** jako root widget naszej aplikacji, zmieniamy wielkość czcionki etykiety na ***„100dp”*** oraz bindujemy wartość jej pola ***text*** do wartości pola ***some_number***
- Za każdym razem gdy wykona się metoda ***update()***, wartość zmiennej ***some_number*** zwiększy się o 5
- Ponieważ Kivy obserwuje zmienną ***some_number***, to gdy jej wartość ulega zmianie, Framework automatycznie zmienia wartość zmiennej ***text*** etykiety i aktualizuje interfejs użytkownika

KIVY NA ANDROIDZIE I IOS

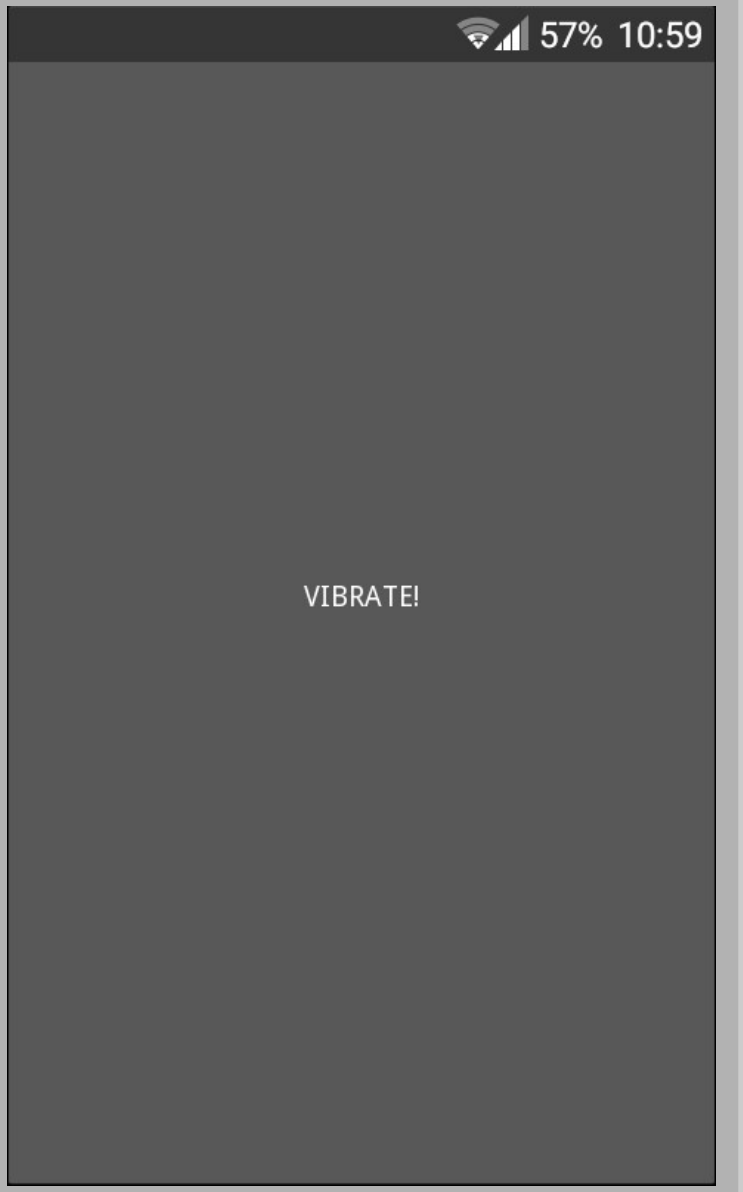
- Kivy jest frameworkiem multiplatformowym, jednak doskonale nadaje się do tworzenia aplikacji mobilnych: interfejs programu domyślnie jest responsywny, cykl życia aplikacji wspiera pauzowanie i wznawianie, możemy też korzystać z natywnego systemowego API
- Dostęp do natywnego API platformy zapewnia albo wieloplatformowa biblioteka ***plyer***, albo też moduł ***pyJNIus*** na Androidzie oraz ***pyOBJus*** na iOS
- Na Androidzie możemy też korzystać z dobrodziejstw projektu Python-for-Android autorstwa Kivy Organization

KIVY NA ANDROIDZIE – PLYER

Kivy Organization stworzyło moduł do obsługi sprzętu i natywnego API niezależnie od platformy, na jakiej jest on używany. Nazywa się on ***plyer*** i pozwala na napisanie jednego kodu, który działa na Androidzie, iOS i platformach desktop, korzystając z możliwości każdej z nich „pod maską”. Dzięki niemu możemy korzystać m.in. z GPS, akcelerometru, kompasu, silnika wibracji czy statusu baterii wywołując metody Pythona, bez potrzeby martwienia się o utrudniony dostęp do natywnego API.

KIVY NA ANDROIDZIE - PLYER

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

try:
    from plyer import vibrator
except:
    print 'Unable to import proper module!'

class VibratingButton(Button):

    def start_vibration(self, time):
        try:
            vibrator.vibrate(time)
        except:
            print 'Unable to start vibrations!'

root = Builder.load_string('''

VibratingButton:
    text: 'VIBRATE!'
    on_release: self.start_vibration(3)

''')

class TestApp(App):

    def build(self):
        return root

TestApp().run()
```



KIVY NA ANDROIDZIE – PLYER

- Importujemy potrzebne moduły Kivy – ***App***, ***Label***, ***Builder***
- Wykonujemy próbę importu modułu ***vibrator*** z pakietu ***plyer***, przechwytując wyjątek w razie niepowodzenia
- Tworzymy klasę ***VibratingButton*** dziedzicząc z klasy ***Button***
- Tworzymy w tej klasie metodę ***start_vibration()***, przyjmującą jako argument czas w sekundach. Metoda ta spowoduje wibrowanie urządzenia przez podany czas

KIVY NA ANDROIDZIE – PLYER

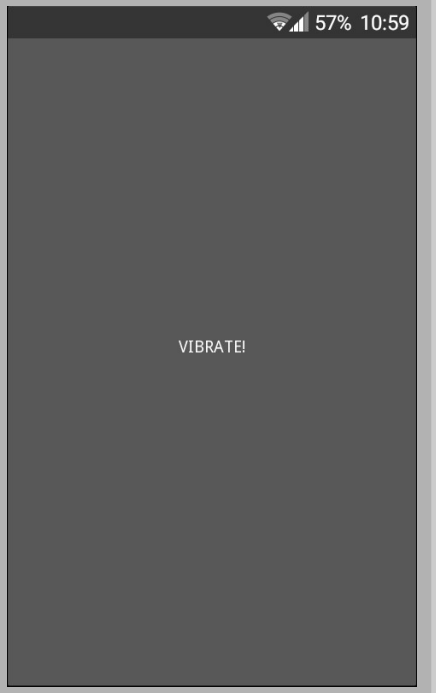
- Dodajemy naszą nowo utworzoną klasę jako root-widget aplikacji. Do pola text przypisujemy wartość „VIBRATE”
- Do sygnału ***on_release*** emitowanego po puszczeniu przycisku ***VibratingButton*** bindujemy metodę ***start_vibration()***
- Aplikacja uruchomiona na telefonie z Androidem po kliknięciu przycisku spowoduje, że urządzenie będzie wibrowało przez 3 sekundy

KIVY NA ANDROIDZIE – PYJNIUS

Kivy Organization stoi także za stworzeniem modułu pyJNIus, który umożliwia mapowanie klas Javy do zmiennych Pythona poprzez interfejsy JNI (Java Native Interface). Dzięki użyciu tego narzędzia jesteśmy w stanie wykonać praktycznie każde zadanie związane z obsługą Android API, niemniej wymaga to nieco znajomości samej Javy i mechanizmów pozyskiwania danych na platformie Google.

KIVY NA ANDROIDZIE - PYJNIUS

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

try:
    from jnius import autoclass
    Context = autoclass('android.content.Context')
    PythonActivity = autoclass('org.renpy.android.PythonActivity')
except:
    print 'Unable to load proper module!'

class VibratingButton(Button):

    def start_vibration(self, time):
        try:
            activity = PythonActivity.mActivity
            vibrator = activity.getSystemService(Context.VIBRATOR_SERVICE)
            vibrator.vibrate(time)
        except:
            print 'Unable to start vibrations!'

root = Builder.load_string('''

VibratingButton:
    text: "VIBRATE!"
    on_release: root.start_vibration(3000)

''')

class TestApp(App):

    def build(self):
        return root

TestApp().run()
```



KIVY NA ANDROIDZIE – PYJNIUS

- Importujemy potrzebne moduły Kivy – ***App***, ***Label***, ***Builder***
- Próbujemy wykonać import modułu ***autoclass*** z pakietu ***jnius*** wyłapując wyjątek w przypadku błędu
- Za pomocą narzędzia ***autoclass*** mapujemy klasy Javy do zmiennych Pythona i w razie błędu łapiemy wyjątek

KIVY NA ANDROIDZIE – PYJNIUS

- Zmienna ***Context*** odzwierciedla dzięki temu klasę ***Context***, a zmienna ***PythonActivity*** klasę naszej aplikacji Kivy uruchomionej na systemie Android
- Dzięki ***pyJNIusowi*** możemy dla zadeklarowanych klas Javy wykonywać ich metody prosto z poziomu Pythona tak, jakbyśmy to robili pisząc natywnie
- Tworzymy klasę ***VibratingButton*** dziedzicząc z klasy Button

KIVY NA ANDROIDZIE – PYJNIUS

- Tworzymy w tej klasie metodę ***start_vibration()***, przyjmującą jako argument czas w sekundach. Metoda ta spowoduje wibrowanie urządzenia przez podany w argumencie czas
- W metodzie tej uzyskujemy dostęp do naszej aktywności oraz systemowej obsługi wibratora wywołując odpowiednie metody z Android API

KIVY NA ANDROIDZIE – PYJNIUS

- Dodajemy naszą nowo utworzoną klasę jako root-widget aplikacji. Do pola text przypisujemy wartość „VIBRATE”
- Do sygnału ***on_release*** emitowanego po puszczeniu przycisku ***VibratingButton*** bindujemy metodę ***start_vibration()***
- Aplikacja uruchomiona na telefonie z Androidem po kliknięciu przycisku spowoduje, że urządzenie będzie wibrowało przez 3 sekundy

ANDROID – TWORZENIE PAKIETÓW APK

- Aplikacje używające Kivy mogą zostać w łatwy sposób spakowane do postaci pakietów APK, które następnie można zainstalować na smartfonie lub tablecie
- Aby stworzyć pakiet APK mamy do wyboru dwie opcje: albo ręcznie kompilować źródła z użyciem projektu Python-for-Android, albo też wybrać kompilację z użyciem narzędzia Buildozer
- Kompilacja aplikacji na Androida możliwa jest tylko na systemie linuksowym, np. Ubuntu
- Używanie Buildozera znacznie usprawnia proces testowania i produkcji ostatecznej wersji aplikacji

ANDROID – TWORZENIE PAKIETÓW APK

- Buildozer potrafi także paczkować programy dla systemu iOS, jednak jako że wymagany jest do tego sprzęt z Mac OS X oraz licencja developerska Apple skupimy się wyłącznie na Androidzie
- Buildozer pobiera za nas wszystkie zależności, potrzebne do zbudowania paczki, w tym Android SDK, NDK. Bardzo przyspiesza to cały proces
- Buildozer kompiluje bootstrap aplikacji napisany natywnie w Javie, który z kolei uruchamia interpreter Pythona z pakietu oraz uruchamia nasz skrypt

ANDROID – TWORZENIE PAKIETÓW APK

- Jako że jesteśmy w stanie dostać się do plików źródłowych Javy w projekcie, możemy pewne rzeczy edytować, np. dodać tam zewnętrzne Activity, np. reklamowe, z paczek SDK
- Pakiet APK z aplikacją Kivy zawiera nasz projekt, interpreter Pythona oraz źródła Kivy. Minimalna paczka zajmuje około 6-7 MB. Po rozpakowaniu na urządzeniu z Androidem, całość zajmuje około 27 MB
- Buildozera obsługujemy z poziomu konsoli

BUILDOZER

```
buildozer init

"""
Tym poleceniem inicjujemy projekt Buildozera w aktualnym katalogu.
"""

buildozer android debug

"""
Tym poleceniem zlecamy kompilację aplikacji w trybie debugowania.
"""

buildozer android debug deploy run logcat

"""
Tym pleceniem zlecamy kompilację aplikacji, przesłanie jej na urządzenie,
uruchomienie oraz otworzenie w konsoli podglądu na Android logcat, dzięki czemu
możemy na bieżąco wyłapywać błędy programu
"""
```

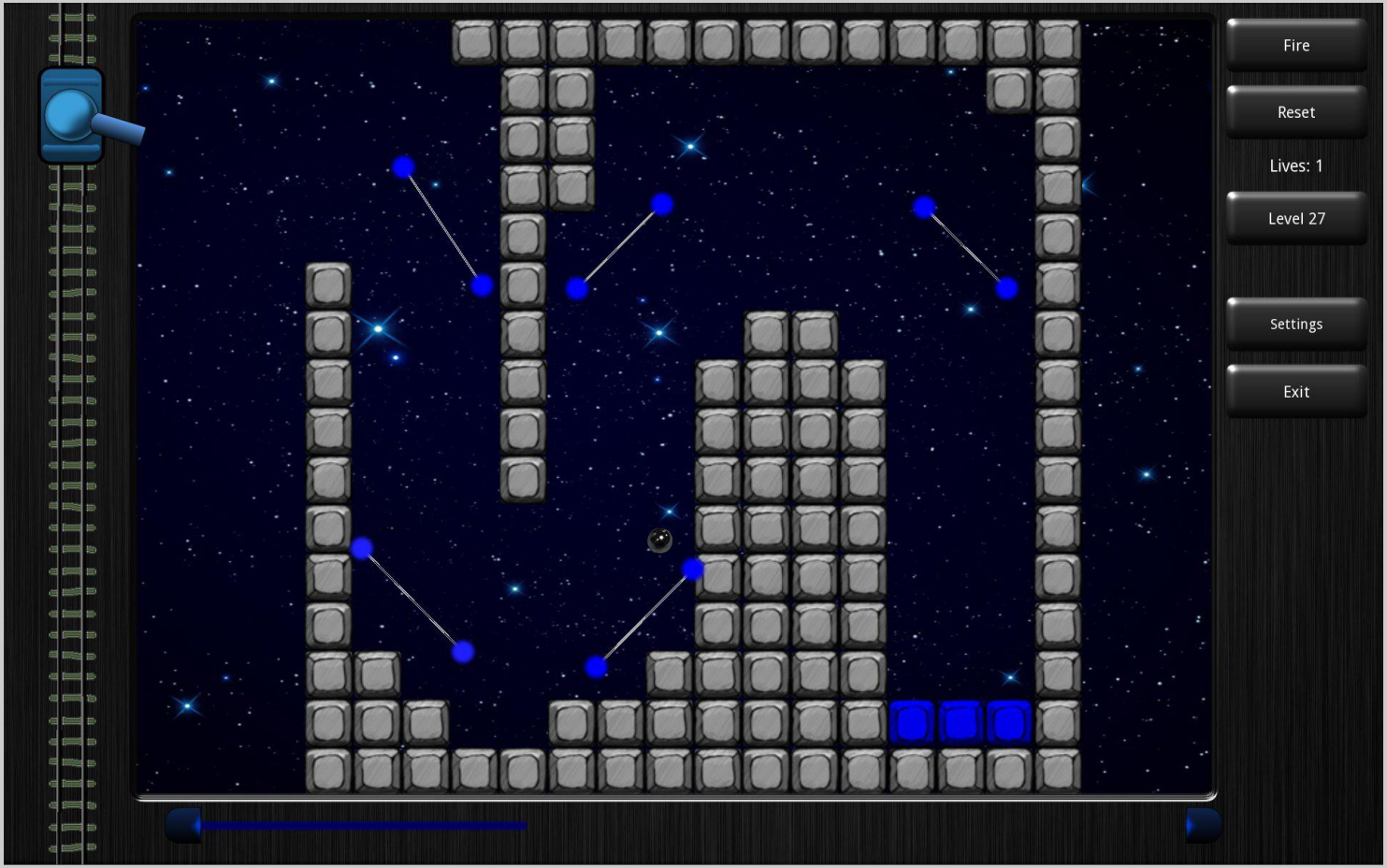
ANDROID – TWORZENIE PAKIETÓW APK

- Kompilację przy użyciu Buildozera możemy kontrolować poprzez plik „buildozer.spec”, który tworzy się nam w katalogu projektu po wykonaniu polecenia „buildozer init”
- Poprzez plik buildozer.spec możemy zmienić nazwę aplikacji, domenę, ikonę, ekran ładowania, numer wersji pakietu, pozwolenia, a także podpowiedzieć kompilatorowi czy ma spakować także dodatkowe moduły Pythona
- Używanie Kivy i Buildozera drastycznie skraca czas potrzebny na zaprogramowanie i kompilację aplikacji dla Androida

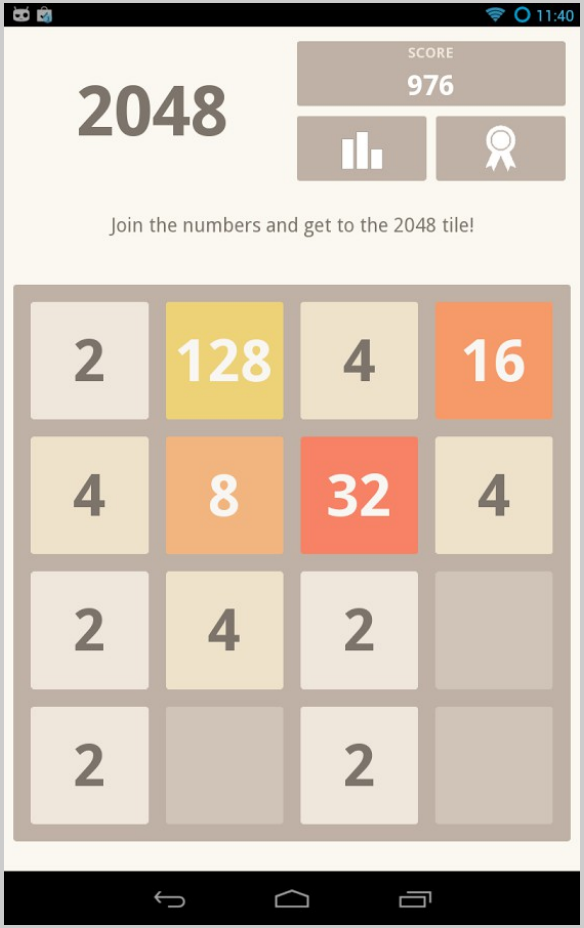
PRZYKŁADOWE APLIKACJE

- Skompilowana do postaci pakietu APK lub IPA aplikacja może być podpisana i udostępniona użytkownikom poprzez sklepy z aplikacjami takie jak np. Google Play
- Obecnie w Google Play oraz App Store jest kilkadziesiąt projektów, stworzonych w całości przy użyciu Kivy Framework

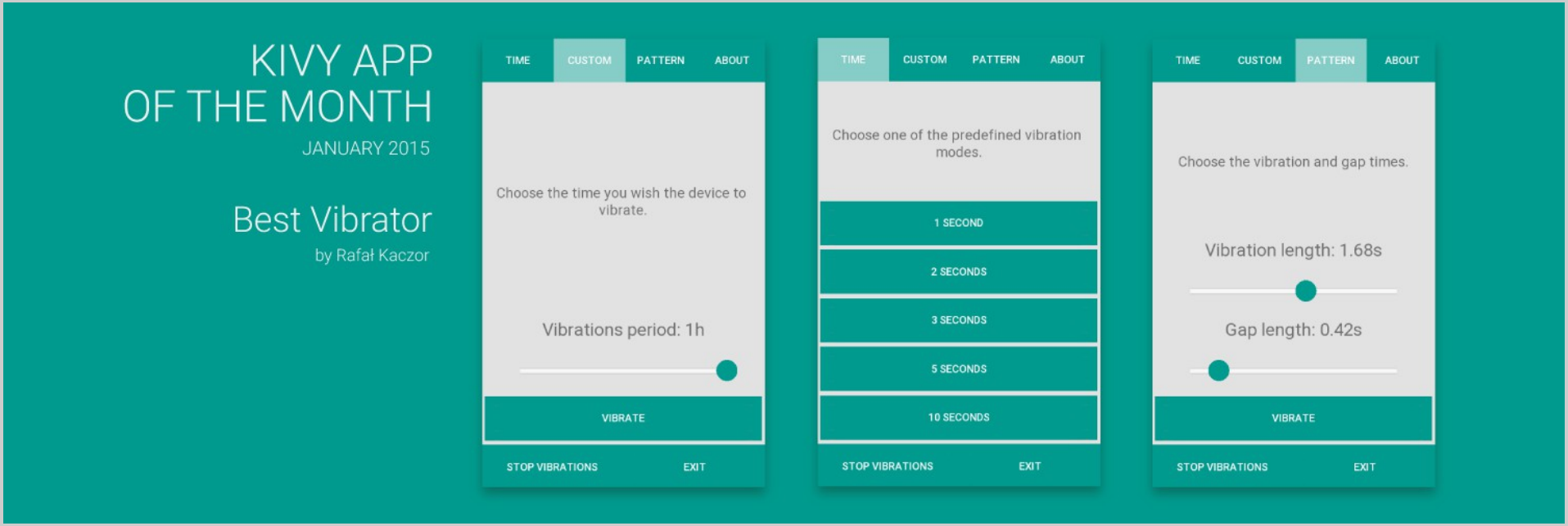
DEFLECTTOUCH



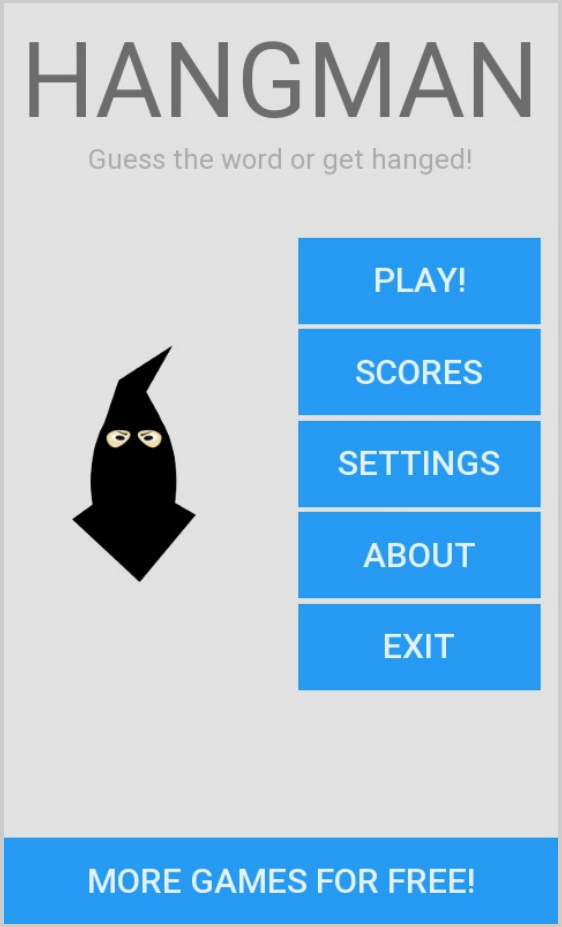
2048 USING KIVY



BEST VIBRATOR (MATERIAL)

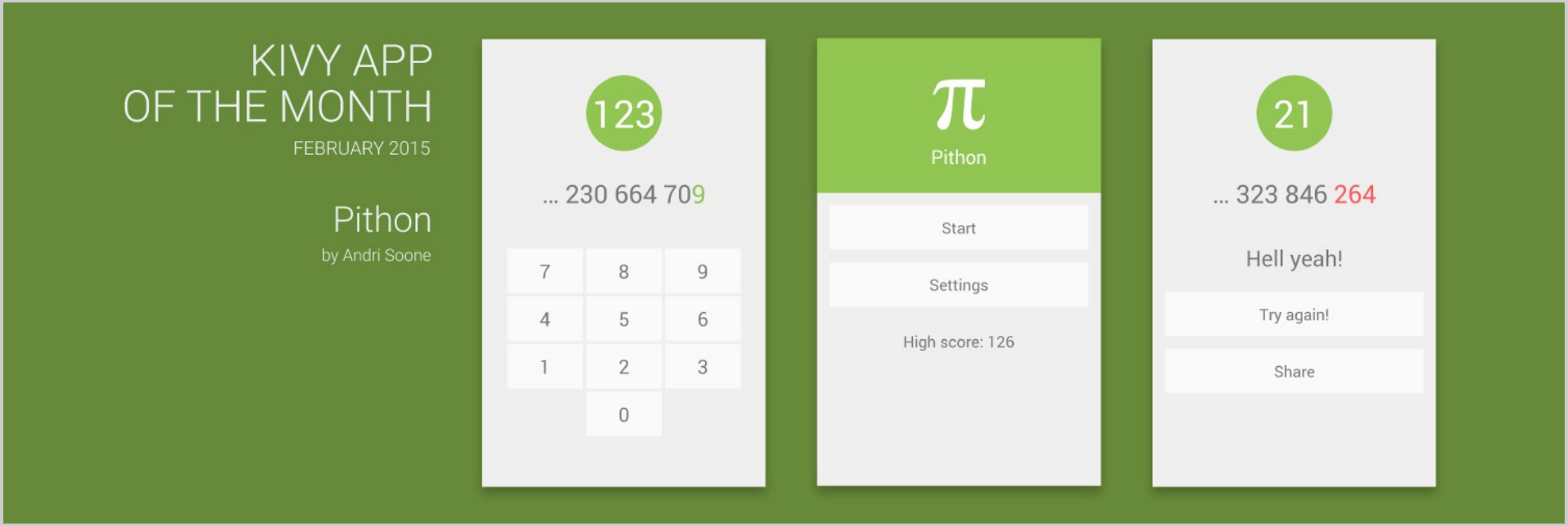


HANGMAN


HANGMAN
Guess the word or get hanged!
PLAY!
SCORES
SETTINGS
ABOUT
EXIT
MORE GAMES FOR FREE!

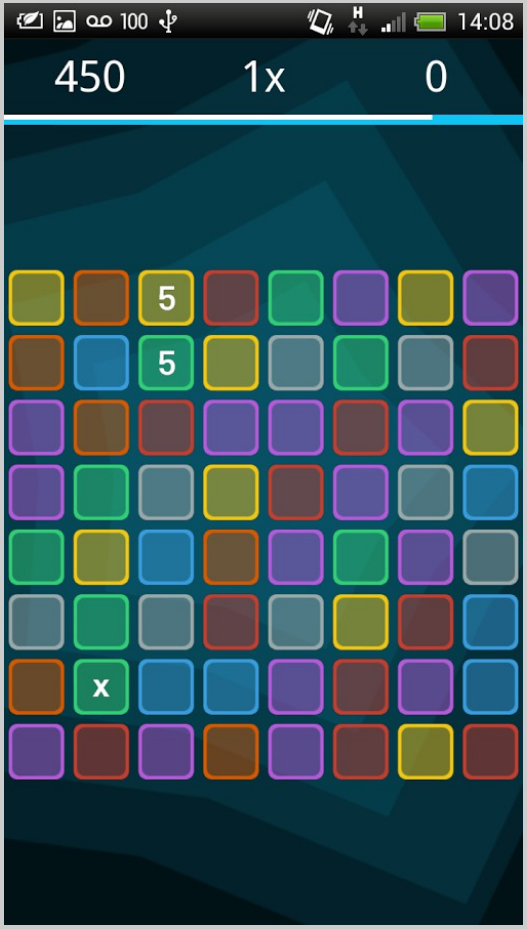

Errors: 2 Time: 25
_PINI_T_ET_
A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z

PITHON

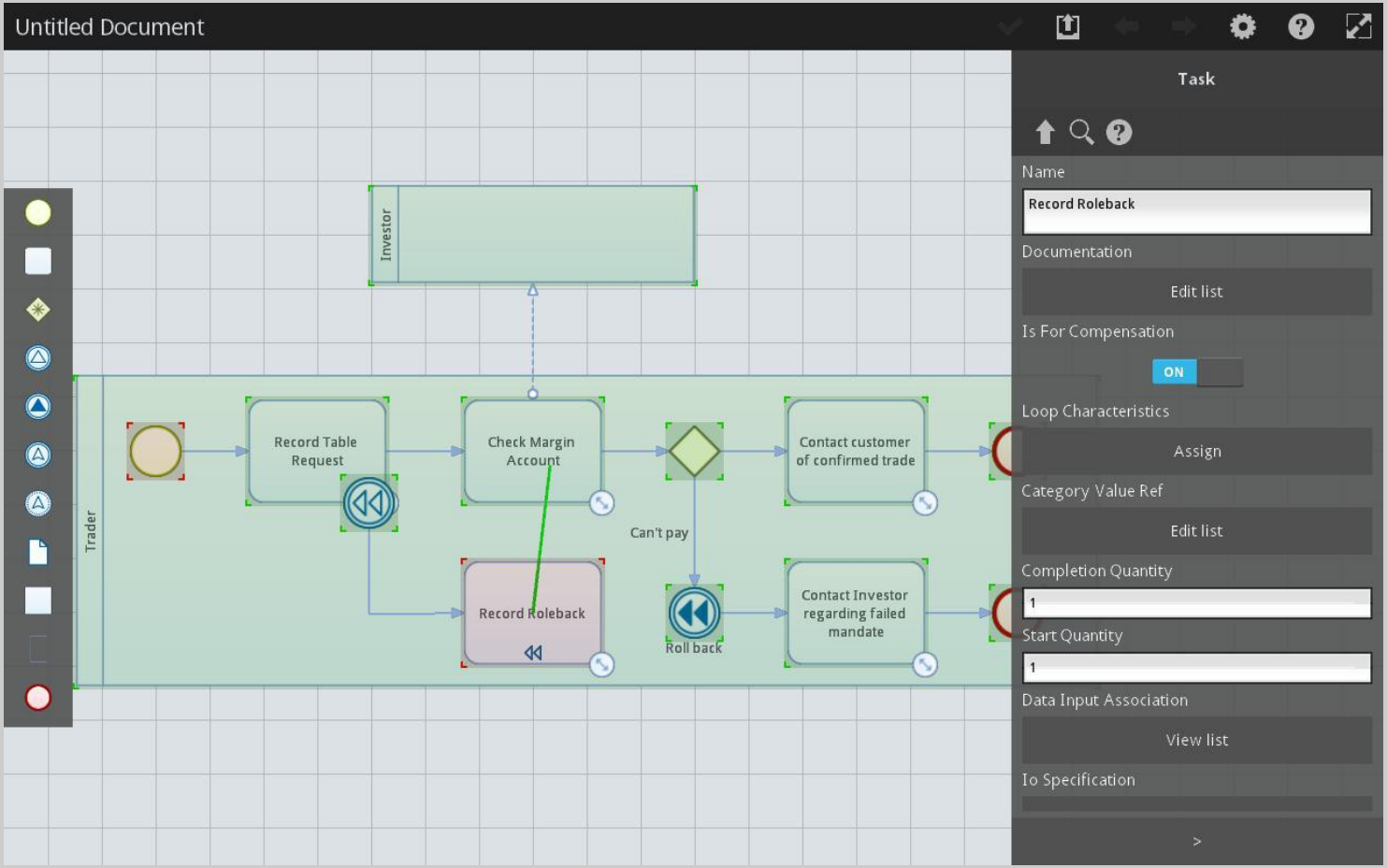

KIVY APP
OF THE MONTH
FEBRUARY 2015
Pithon
by Andri Soone
123
... 230 664 709
7
8
9
4
5
6
1
2
3
0
π
Pithon
Start
Settings
High score: 126
21
... 323 846 264
Hell yeah!
Try again!
Share

PLAYING CARDS

Flat Jewels



PROCESSCRAFT



PRZYDATNE LINKI

- Strona projektu: http://kivy.org/
- Dokumentacja Kivy: http://kivy.org/docs/
- Kanał IRC projektu: #kivy na irc.freenode.net
- Wprowadzenie do Kivy: http://kivy.org/docs/gettingstarted/intro.html
- Polska grupa Kivy na Facebooku: https://www.facebook.com/groups/KivyPolska/
- Kivy Crash Course: http://inclem.net/pages/kivy-crash-course/

PRZYDATNE LINKI

- Dokumentacja dot. plyer: https://readthedocs.org/projects/plyer/
- Dokumentacja dot. Buildozer: https://readthedocs.org/projects/buildozer/
- Integracja reklam AdBuddiz w aplikacjach Kivy na Androida: https://github.com/kivy/kivy/wiki/AdBuddiz-Android-advertisements-integration-for-Kivy-apps
- Moduł AdBuddizController do wygodnej obsługi tych reklam: https://github.com/rafalo1333/AdBuddizController

PODSUMOWANIE

Plusy:

- Bardzo szybkie i wygodne tworzenie aplikacji
- Natywne wsparcie multidotyku
- Łatwe tworzenie interfejsów użytkownika dzięki Kivy Language
- Intuicyjna architektura aplikacji
- Wsparcie dla wielu platform i ich API
- Rozbudowana dokumentacja

PODSUMOWANIE

Minusy:

- Nieco gorsza wydajność na starych smartfonach i tabletach
- Społeczność dopiero się rozrasta
- Dostęp do API Androida wymaga nieco znajomości Javy