# Создание настольных Python приложений

## с графическим интерфейсом пользователя

Тимур Машнин

12+

# Создание настольных Python приложений

## с графическим интерфейсом пользователя

Тимур Машнин

12+

# Тимур Машнин
# Создание настольных Python приложений с графическим интерфейсом пользователя

# Исходный код

Исходный код к примерам можно скачать по адресу https://github.com/novts/python-gui.

# Введение

**Создание настольных Python приложений с GUI**

Введение

Де факто Python является наиболее популярным объектно-ориентированным языком программирования, который используется для веб-разработки и анализа больших данных.

И конечно, Python – это интерактивный язык программирования, который предоставляет широкий спектр возможностей для создания графического интерфейса пользователя.

Riverbank Computing News Software Other Stuff Support Commercial About

## What is PyQt?

PyQt is a set of Python v2 and v3 bindings for The Qt Company's Qt application framework and runs on all platforms supported by Qt including Windows, macOS, Linux, iOS and Android. PyQt5 supports Qt v5. PyQt4 supports Qt v4 and will build against Qt v5. The bindings are implemented as a set of Python modules and contain over 1,000 classes.

PyQt4 and Qt v4 are no longer supported and no new releases will be made. PyQt5 and Qt v5 are strongly recommended for all new development.

### License

PyQt is dual licensed on all supported platforms under the GNU GPL v3 and the Riverbank Commercial License. Unlike Qt, PyQt is not available under the LGPL. You can purchase the commercial version of PyQt here. More information about licensing can be found in the License FAQ.

PyQt does not include a copy of Qt. You must obtain a correctly licensed copy of Qt yourself. However, binary wheels of the GPL version of PyQt5 are provided and these include a copy of the LGPL version of Qt.

### PyQt Components

A description of the components of PyQt5 can be found in the PyQt5 Reference Guide.

A description of the components of PyQt4 can be found in the PyQt4 Reference Guide.

PyQt – это библиотека графического фреймворка Qt для языка программирования Python.

А Qt кью-ти – это кроссплатформенный инструментарий для разработки программного обеспечения на языке программирования C++, такого как графические интерфейсы, работа с сетью, базами данных и XML.

PyQt работает на всех платформах, поддерживаемых Qt – Linux и другие UNIX-подобные ОС, Mac OS и Windows.

И существуют 2 версии: PyQt5, поддерживающий Qt 5, и PyQt4, поддерживающий Qt 4.

PyQt практически полностью реализует возможности Qt, включая набор виджетов графического интерфейса, доступ к базам данных с помощью SQL, парсер XML и так далее.

PyQt также включает в себя Qt Designer— дизайнер графического интерфейса пользователя с генерацией Python кода из файлов, созданных в Qt Designer.

Project description

Release history

Download files

Project links

Homepage

Download

Statistics

View statistics for this project via Libraries.io, or by using our public dataset on Google BigQuery

PySide2

Introduction

PySide2 is the official Python module from the Qt for Python project, which provides access to the complete Qt 5.12+ framework.

The Qt for Python project is developed in the open, with all facilities you'd expect from any modern OSS project such as all code in a git repository and an open design process. We welcome any contribution conforming to the Qt Contribution Agreement.

Installation

Since the release of the Technical Preview it is possible to install via `pip`, both from Qt's servers and PyPi:

```
pip install PySide2
```

PySide – это также библиотека графического фреймворка Qt для языка программирования Python.

Основное отличие PySide от PyQt – это лицензии под которыми распространяются эти две обёртки Qt.

PyQt5 распространяется под GPL и коммерческой лицензией.

А PySide2 распространяется как Qt под GPL, LGPL и коммерческой лицензией.

То есть если вы пишите открытое ПО – можно использовать как PyQt5, так и PySide2.

Но если вы пишите закрытое/коммерческое ПО – бесплатно можно использовать только PySide2, а для использования PyQt5 потребуется покупать коммерческую лицензию.

Python » English 3.9.1rc1 Documentation » The Python Standard Library » Graphical User Interfaces with Tk » previous | next | modules | index

Quick search Go |

Table of Contents

- tkinter — Python interface to Tcl/Tk
  - Tkinter Modules
  - Tkinter Life Preserver
    - How To Use This Section
    - A Simple Hello World Program
  - A (Very) Quick Look at Tcl/Tk
  - Mapping Basic Tk into Tkinter
  - How Tk and Tkinter are Related
  - Handy Reference
    - Setting Options
    - The Packer
    - Packer Options
    - Coupling Widget Variables

# tkinter — Python interface to Tcl/Tk

**Source code:** Lib/tkinter/__init__.py

The tkinter package ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running python -m tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

**See also:** Tkinter documentation:

**Python Tkinter Resources**
The Python Tkinter Topic Guide provides a great deal of information on using Tk from Python and links to other sources of information on Tk.

Tkinter – это самая популярная библиотека для создания графического интерфейса пользователя или настольных приложений.

Tkinter – это комбинация стандартного графического интерфейса пользователя Tk и Python.

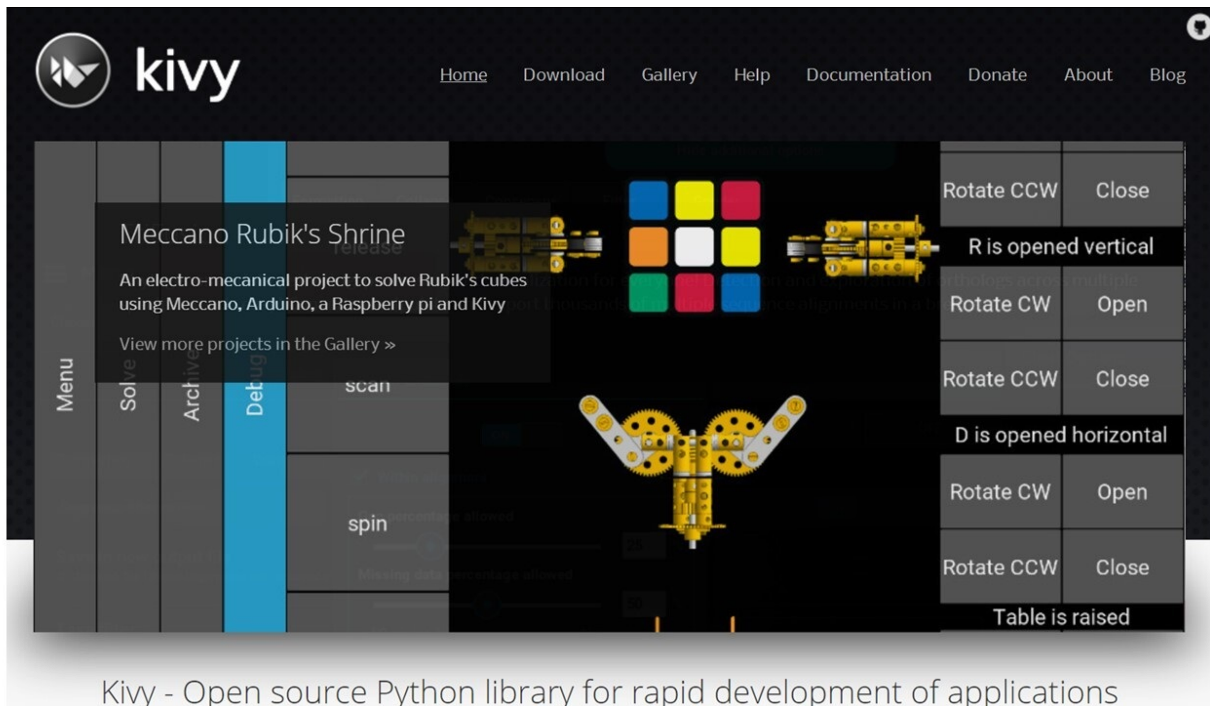
А Tk – это кроссплатформенная библиотека графического интерфейса с открытым исходным кодом.

Tkinter входит в стандартную библиотеку Python.

И Tkinter – это свободное программное обеспечение, распространяемое под Python-лицензией.

TKinter поставляется с хорошей документацией, что является основным ее достоинством.

И получить ответы на свои вопросы здесь легко, так как у Tkinter тысячи пользователей, потому что эта библиотека используется в течение очень долгого времени.




Kivy – это бесплатная среда Python с открытым исходным кодом для разработки кросс-платформенных приложений с поддержкой мультитач с пользовательским интерфейсом.

Kivy создана поверх OpenGL и для создания пользовательских интерфейсов дает возможность один раз написать код и запустить его на разных платформах Windows, MacOSX, Linux, Android, iOS и Raspberry.

wxPython
The GUI Toolkit for Python

About News Downloads Documentation Support Developers Blog

Welcome to wxPython!

This website is all about wxPython, the cross-platform GUI toolkit for the Python language. With wxPython software developers can create truly native user interfaces for their Python applications, that run with little or no modifications on Windows, Macs and Linux or other unix-like systems.

Learn more

WxPython – это обёртка библиотеки кроссплатформенного графического интерфейса пользователя wxWidgets, написанной на языке программирования C++.

Это еще одна из альтернатив Tkinter, которая поставляется вместе с Python.

И WxPython реализована в виде модуля расширения Python.

(This library is available under a free and permissive license however, if you Enjoy *Dear PyGui* please consider becoming a Sponsor)

python 3.6 | 3.7 | 3.8 | 3.9 pypi v0.6.403 downloads 364k

Embedded Build passing static-analysis passing build pending

**Dear PyGui** is a **simple to use (but powerful) Python GUI framework**. *Dear PyGui* is **NOT** a wrapping of **Dear ImGui** in the normal sense. It is a library built with *Dear ImGui* which simulates a traditional retained mode GUI (as opposed to *Dear ImGui*'s immediate mode paradigm).

Dear
PyGUI – это простая и легкая библиотека графического интерфейса пользователя, так как она полностью связана с языком программирования Python.

Dear PyGui предоставляет оболочку библиотеки C++ Dear ImGui, которая имитирует традиционный графический интерфейс.

Это кроссплатформенная среда приложений с графическим интерфейсом пользователя, которая отображает естественный графический интерфейс платформы.

Здесь мы перечислили наиболее широко используемые и лучшие доступные фреймворки графического пользовательского интерфейса Python.

И вы можете выбрать наиболее подходящую вам среду для разработки графического интерфейса Python.

Далее мы более подробно разберем каждую библиотеку.

# Библиотека PyQt

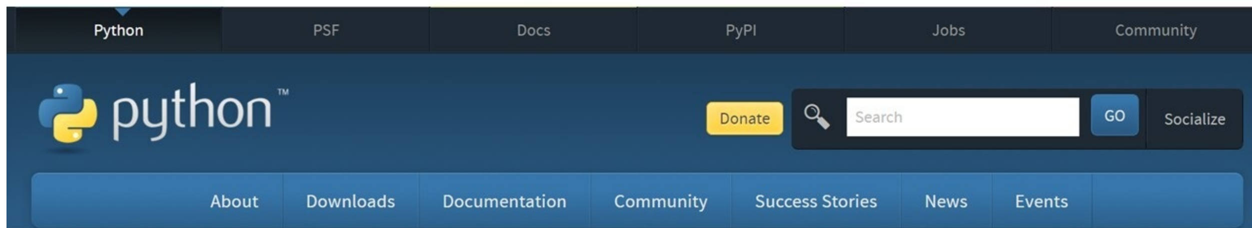
Создание настольных Python приложений с GUI

Библиотека PyQt

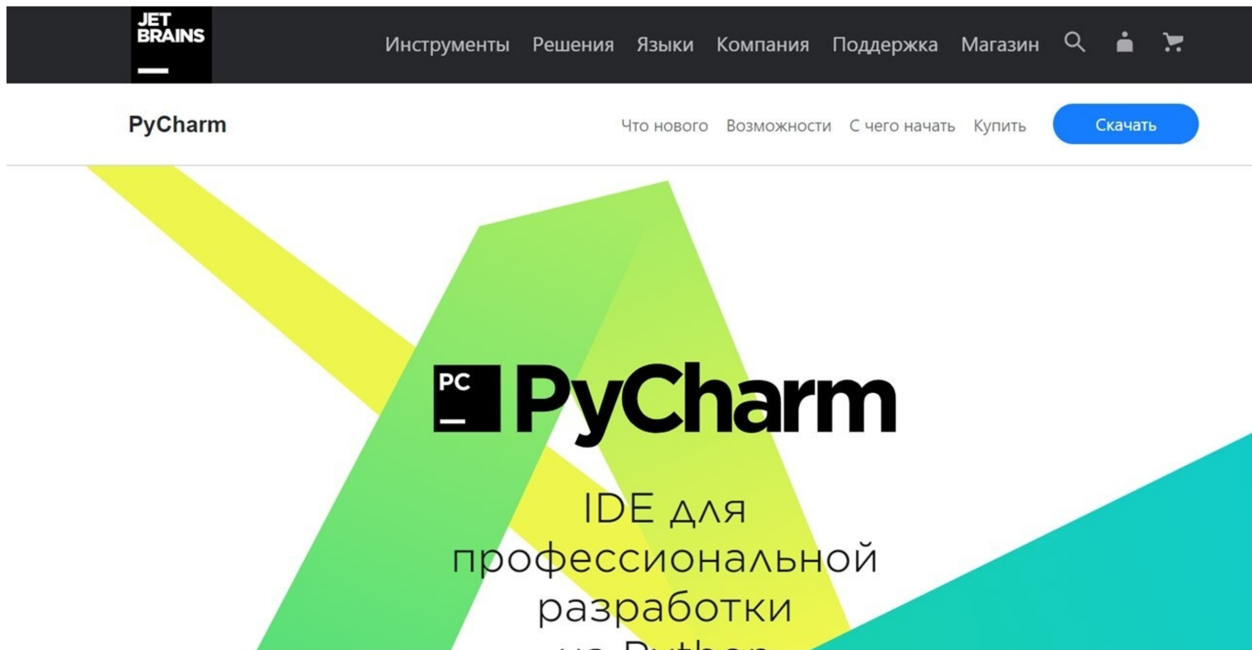
PyQt – это библиотека, которая позволяет использовать библиотеку графического интерфейса Qt в Python.

Сама библиотека Qt написана на C ++.

Самая последняя версия библиотеки – это PyQt5, и она поддерживает последнюю версию Qt5.



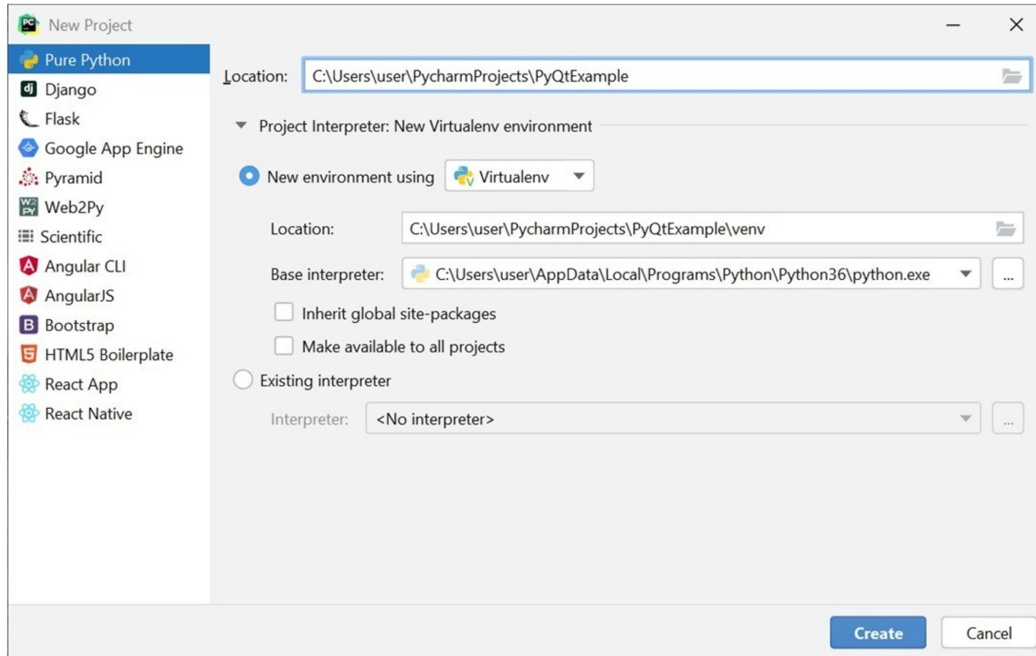

Для работы с библиотекой PyQt5, установим питон 3.6.




Для разработки приложений питон с графическим интерфейсом пользователя мы будем пользоваться средой разработки

PyCharm

.



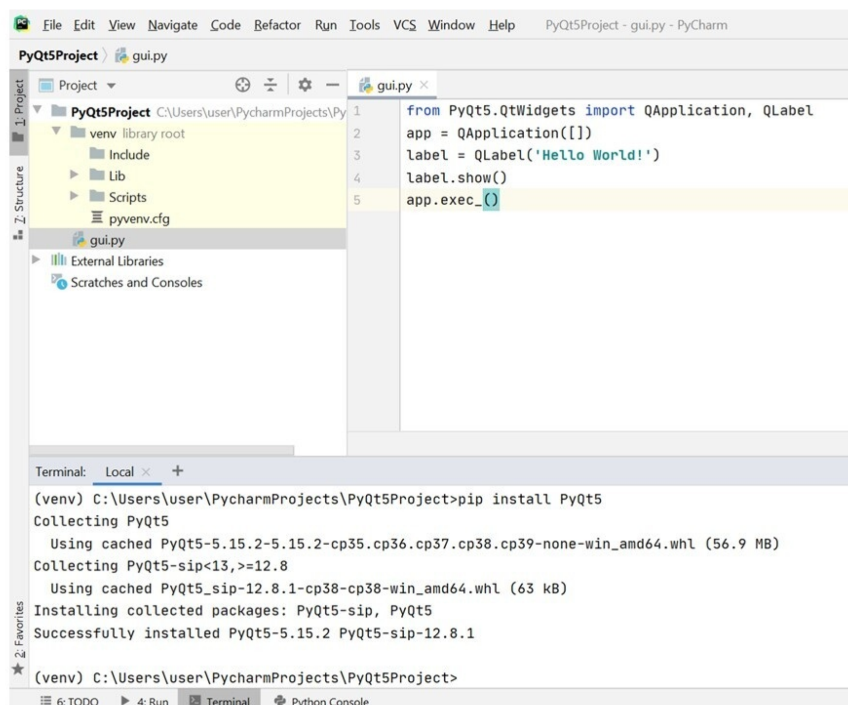

Создадим питон проект. При этом будет автоматически создана виртуальная среда.

Виртуальная среда – это просто локальный каталог, содержащий библиотеки для конкретного проекта.

pip install PyQt5

```
Terminal: Local
(venv) C:\Users\user\PycharmProjects\PyQt5Project>pip install PyQt5
Collecting PyQt5
  Using cached PyQt5-5.15.2-5.15.2-cp35.cp36.cp37.cp38.cp39-none-win_amd64.whl (56.9 MB)
Collecting PyQt5-sip<13,>=12.8
  Using cached PyQt5_sip-12.8.1-cp38-cp38-win_amd64.whl (63 kB)
Installing collected packages: PyQt5-sip, PyQt5
Successfully installed PyQt5-5.15.2 PyQt5-sip-12.8.1

(venv) C:\Users\user\PycharmProjects\PyQt5Project>
6: TODO   4: Run   Terminal   Python Console
```

И для установки библиотеки PyQt просто наберите в окне терминала, в командной строке pip install PyQt5.



Далее в проекте создадим питон файл и наберем в нем код.

```
from PyQt5.QtWidgets import QApplication, Qlabel

app = QApplication([])

label = QLabel('Hello World!')

label.show()

app.exec_()
```

Сначала мы загружаем PyQt с помощью оператора импорта.

И из PyQt5 виджетов импортируем QApplication, QLabel.

Класс QApplication управляет потоком управления и основными настройками приложения с графическим интерфейсом.

Виджет QLabel обеспечивает отображение текста или изображения.

Затем мы создаем QApplication с помощью команды:

app = QApplication

Это требование Qt – каждое приложение с графическим интерфейсом должно иметь ровно один экземпляр QApplication.

Здесь квадратные скобки представляют аргументы командной строки, переданные приложению.

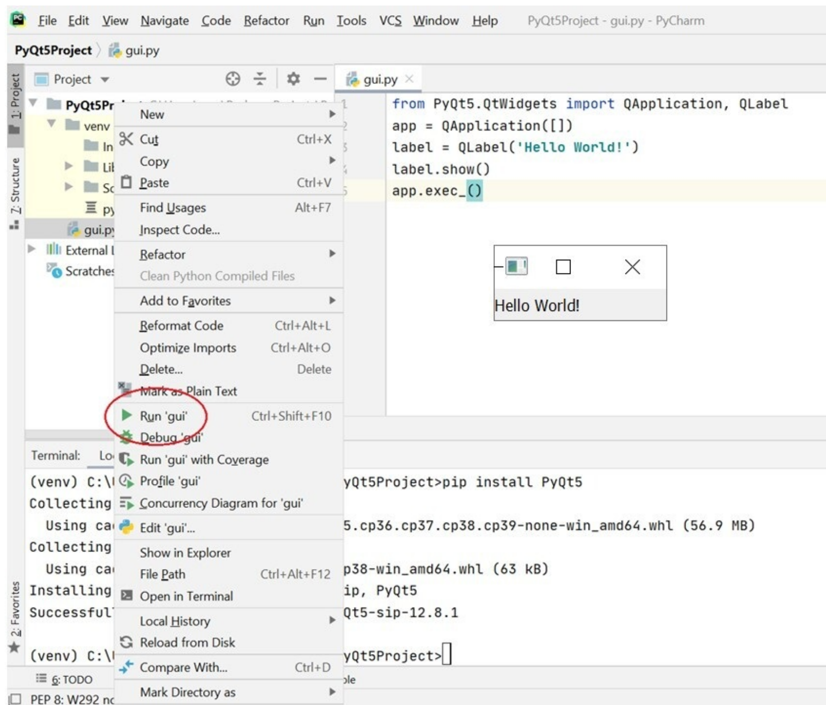
Так как наше приложение не использует никаких параметров, мы оставляем скобки пустыми.

Далее мы создаем простую метку 'Привет, мир!'.

И затем мы говорим Qt показать метку на экране с помощью команды show.

И последний шаг – это передать управление среде Qt и попросить ее «запустить приложение, пока пользователь не закроет его».

Это делается с помощью команды exec.



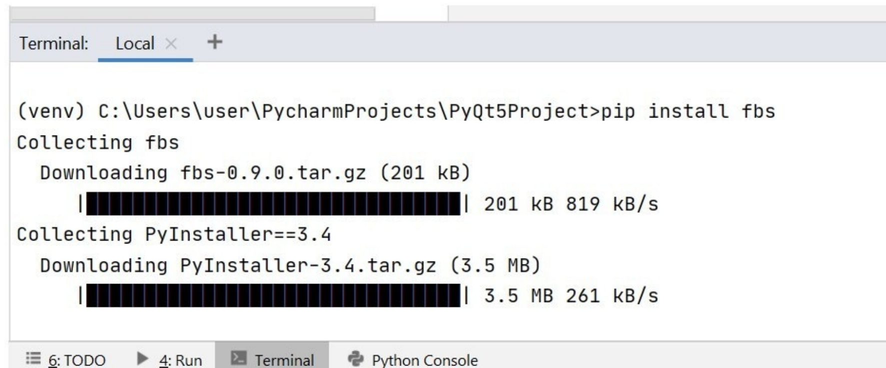

Далее нажмем правой кнопкой мыши на созданном питон файле и выберем команду

run

.

В результате будет запущено приложение и откроется окно с меткой.

pip install fbs




Теперь у нас есть приложение с графическим интерфейсом пользователя.

И оно работает на вашем компьютере.

Вопрос – как его передать другим людям, чтобы они тоже могли его запустить?

Вы можете попросить пользователей вашего приложения установить Python и PyQt, а затем предоставить им свой исходный код.

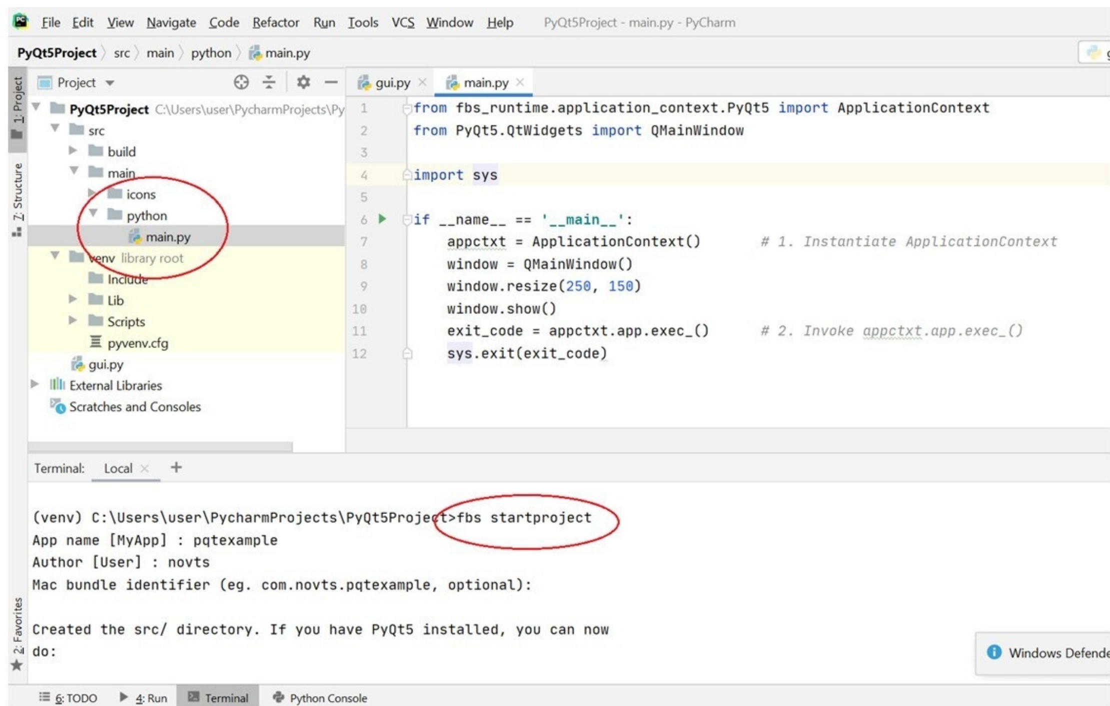
Но это очень неудобно.

Вместо этого нам нужен исполняемый файл, который другие люди могут запускать в своих системах, ничего не устанавливая.

В Python процесс превращения исходного кода в автономный исполняемый файл называется замораживанием.

Хотя существует множество библиотек, которые решают эту проблему, например PyInstaller, py2exe и так далее, здесь мы будем использовать библиотеку под названием fbs, которая позволяет создавать автономные исполняемые файлы для приложений PyQt.

Поэтому для начала установим библиотеку fbs.




Далее мы в терминале запускаем команду fbs startproject.

В результате выполнения которой будет создана папка src/main/python/c файлом main.py.

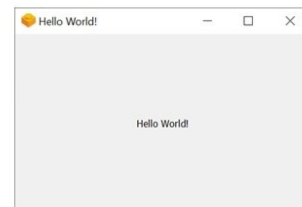
Команда startproject создает необходимую структуру папок для приложения fbs.

Если мы наберем в терминале команду fbs run, откроется просто пустое окно.

Теперь, как нам вставить в это окно нашу метку.

```python
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QMainWindow, QLabel
from PyQt5.QtCore import Qt

import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QMainWindow()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)
    label = QLabel('Hello World!')
    label.setAlignment(Qt.AlignCenter)
    window.setCentralWidget(label)
    window.show()
    exit_code = appctxt.app.exec_()      # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

fbs run




Здесь вы можете заметить, что создание приложения с помощью fbs представляет новую концепцию – ApplicationContext.

При создании приложений PyQt5 обычно используется ряд компонентов или ресурсов, которые используются во всем приложении.

И ApplicationContext предоставляет центральное место для инициализации и хранения этих компонентов, а также предоставляет доступ к некоторым основным функциям fbs.

Объект ApplicationContext также создает и содержит ссылку на глобальный объект QApplication, доступный в ApplicationContext.app, так как каждое приложение Qt должно иметь один и только один объект QApplication для хранения цикла событий и основных настроек.

Теперь, чтобы вставить нашу метку, помимо QMainWindow импортируем метку.

Создадим метку и методом setAlignment установим ее посередине.

Методом setCentralWidget добавим метку в окно QMainWindow.

В результате после вызова команды fbs run мы увидим окно с меткой.

https://doc.qt.io/qt-5/qmainwindow.html

Wiki Documentation Forum Bug Reports Code Review Resource Center Qt Extensions

Qt Documentation

Search Documentation

Qt 5.15 > Qt Widgets > C++ Classes > QMainWindow

Contents

Public Types
Properties
Public Functions
Public Slots
Signals
Reimplemented Protected Functions
Detailed Description
Qt Main Window Framework

# QMainWindow Class

The QMainWindow class provides a main application window. More...

| | |
|---|---|
| Header: | #include <QMainWindow> |
| qmake: | QT += widgets |
| Inherits: | QWidget |

› List of all members, including inherited members

Более подробно про окно QMainWindow можно посмотреть в QT
документации.

```
appctxt = ApplicationContext()

im=QImage(appctxt.get_resource('images/im.jpg'))
```

src/main/resources/base/images/

Теперь, приложениям обычно требуются дополнительные файлы данных помимо исходного кода – например, изображения.

И вот здесь может быть полезен ApplicationContext.

Вы можете поместить ресурсы приложения в папку resources, и чтобы упростить загрузку ресурсов из папки resources, fbs предоставляет метод ApplicationContext.get_resource.

Этот метод принимает имя файла, который можно найти в папке resources, и возвращает абсолютный путь к этому файлу.

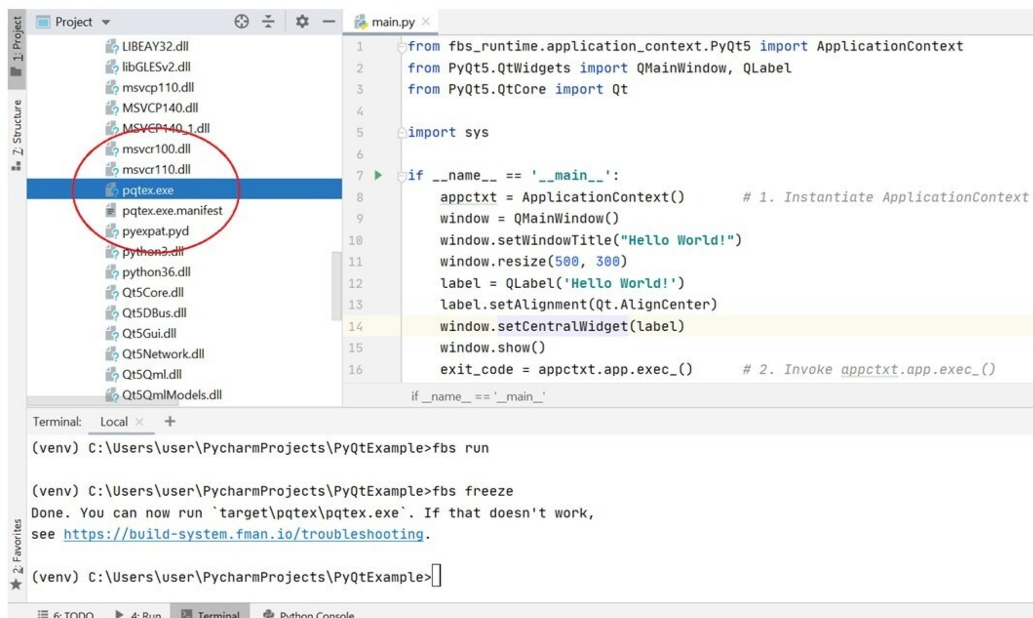
И вы можете использовать этот возвращенный абсолютный путь, чтобы открыть файл.

Папка resources должна содержать папку base плюс любую комбинацию других папок.

Базовая папка содержит файлы, общие для всех операционных систем, в то время как папки для конкретных платформ могут использоваться для файлов, специфичных для данной ОС.

Теперь, далее мы можем использовать fbs, чтобы превратить файл питона в отдельный исполняемый файл.

fbs freeze



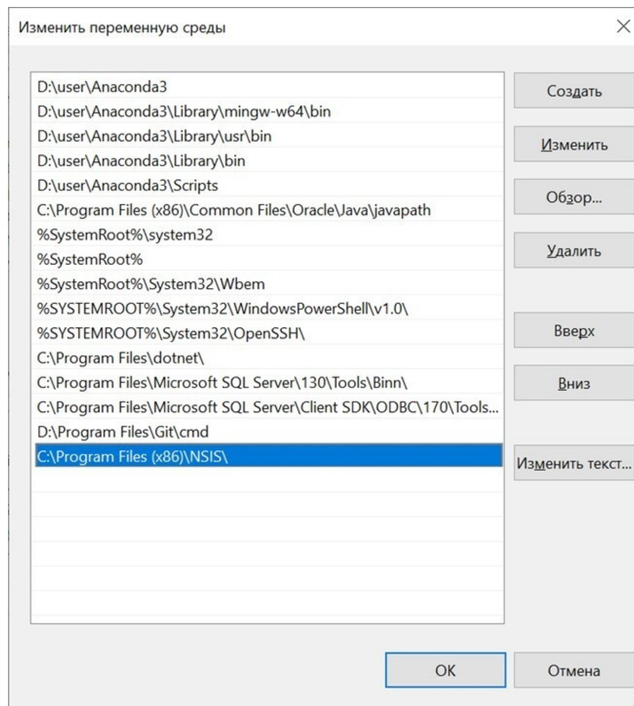

Для этого в терминале наберем команду fbs freeze.

Эта команда помещает исполняемый двоичный файл в целевую папку текущего каталога.

Далее мы можем создать установщик приложения.

https://nsis.sourceforge.io/Main_Page

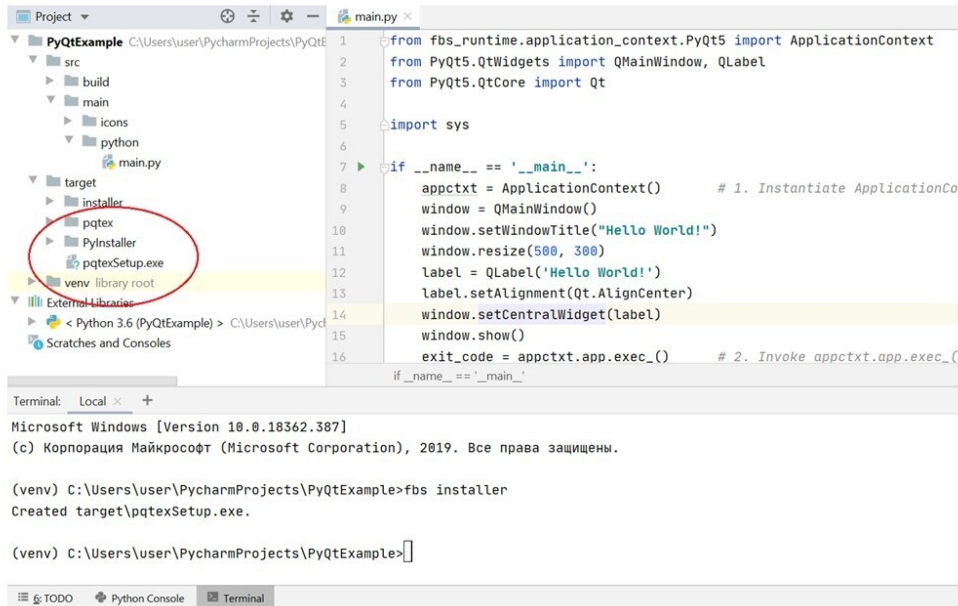
Но для начала мы должны установить NSIS – систему с открытым исходным кодом для создания установщиков Windows.




Также нужно добавить каталог NSIS в переменную среды
Windows
PATH.

После этого нужно перезапустить среду разработки PyCharm, чтобы она увидела эти изменения.

# fbs installer




Далее создадим установщик с помощью команды fbs installer.

Эта команда помещает исполняемый двоичный файл в целевую папку текущего каталога.

Теперь вы можете отправлять его для установки приложения.

# Виджеты и компоновки PyQt

Создание настольных Python приложений с GUI

Виджеты и компоновки PyQt

Система компоновки Qt предоставляет простой и мощный способ организации дочерних виджетов внутри родительского виджета.

QBoxLayout - выстраивает дочерние виджеты по горизонтали или вертикали
QButtonGroup - контейнер для организации групп виджетов кнопок
QFormLayout - управляет формами виджетов ввода и связанными с ними метками.
QGraphicsAnchorLayout - связывает виджеты между собой
QGridLayout - размещает виджеты в сетке
QGroupBox - групповой фрейм окна с заголовком
QHBoxLayout - выстраивает виджеты по горизонтали
QLayout - базовый класс компоновки
QLayoutItem - абстрактный элемент, которым управляет QLayout
QSizePolicy – описывает политику изменения размера по горизонтали и вертикали
QSpacerItem - пустое место в макете
QStackedLayout - стек виджетов, в котором одновременно виден только один виджет
QVBoxLayout - выстраивает виджеты вертикально
QWidgetItem - элемент, представляющий виджет

Qt предоставляет набор классов управления компоновкой.

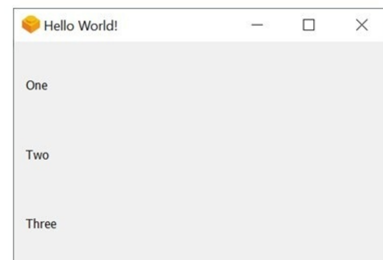
Эти компоновки автоматически позиционируют и изменяют размер виджетов.

И все виджеты Qt могут использовать компоновки для управления своими дочерними элементами с помощью функции setLayout.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel, QHBoxLayout
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    layout = QHBoxLayout()
    layout.addWidget(QLabel('One'))
    layout.addWidget(QLabel('Two'))
    layout.addWidget(QLabel('Three'))
    window.setLayout(layout)

    window.show()
    exit_code = appctxt.app.exec_()      # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

Здесь мы с помощью компоновки QHBoxLayout располагаем метки горизонтально в окне QWidget.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel, QHBoxLayout, QVBoxLayout
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    layout = QVBoxLayout()
    layout.addWidget(QLabel('One'))
    layout.addWidget(QLabel('Two'))
    layout.addWidget(QLabel('Three'))
    window.setLayout(layout)

    window.show()
    exit_code = appctxt.app.exec_()      # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```



Компоновка QVBoxLayout размещает метки вертикально.

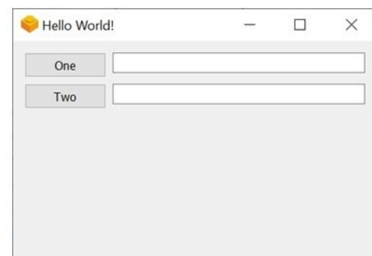
```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel, QGridLayout
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    layout = QGridLayout()
    layout.addWidget(QLabel('One'),0,0,1,2, Qt.AlignHCenter)
    layout.addWidget(QLabel('Two'),1,0, Qt.AlignHCenter)
    layout.addWidget(QLabel('Three'),1,1, Qt.AlignHCenter)
    window.setLayout(layout)

    window.show()
    exit_code = appctxt.app.exec_()      # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

Компоновка QGridLayout располагает элементы в сетке.

При этом для каждого элемента можно указать строку, столбец, в которых должен находиться элемент.

Также можно указать затем сколько строк и столбцов должен заполнять элемент.

```
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QFormLayout, QPushButton, QLineEdit
from PyQt5.QtCore import Qt
import sys

if __name__ == '__main__':
   appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
   window = QWidget()
   window.setWindowTitle("Hello World!")
   window.resize(500, 300)

   layout = QFormLayout()
   layout.addRow(QPushButton("One"), QLineEdit());
   layout.addRow(QPushButton("Two"), QLineEdit());
   window.setLayout(layout)

   window.show()
   exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
   sys.exit(exit_code)
```




Компоновка QFormLayout добавляет два виджета в строку, обычно QLabel и QLineEdit для создания форм.

В качестве резюме – виджеты могут иметь в качестве родительских только другие виджеты, но не компоновки.

Но вы можете вкладывать компоновки в родительскую компоновку с помощью метода addLayout, тогда внутренний макет становится дочерним по отношению к макету, в который он вставлен.

Теперь, модуль Qt Widgets предоставляет набор элементов пользовательского интерфейса для создания пользовательских интерфейсов приложения.

И все, что вы видите в приложении PyQt, представляет собой виджеты: кнопки, метки, окна, диалоговые окна, индикаторы выполнения и т. д.

```python
from fbs_runtime.application_context.PyQt5 import ApplicationContext
from PyQt5.QtWidgets import QWidget, QLabel
from PyQt5.QtCore import Qt

import sys

if __name__ == '__main__':
    appctxt = ApplicationContext()      # 1. Instantiate ApplicationContext
    window = QWidget()
    window.setWindowTitle("Hello World!")
    window.resize(500, 300)

    label = QLabel(window)
    label.setText("Hello PyQt5")
    label.adjustSize()
    label.move(200, 100)

    window.show()
    exit_code = appctxt.app.exec_()     # 2. Invoke appctxt.app.exec_()
    sys.exit(exit_code)
```

С меткой мы уже познакомились.

При создании метки, передавая в конструктор в качестве параметра объект окна, мы сообщаем, что метка является частью окна.

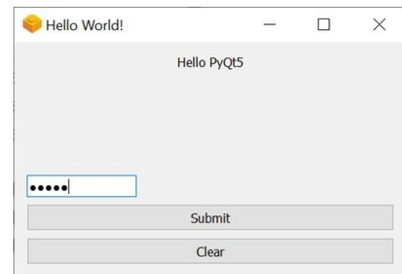
Метки имеют размер по умолчанию, и для длинных строк текста размер по умолчанию может быть слишком мал.

К счастью, у нас есть метод adjustSize, который автоматически настраивает размер метки.

В противном случае длинный текст будет отображаться на экране только частично.

И метод move определяет начальную позицию метки от левого верхнего угла экрана.

```
def show():
   print(line.text())


line = QLineEdit()
line.setEchoMode(QLineEdit.Password)
line.setFixedWidth(140)

buttonS = QPushButton()
buttonS.setText("Submit")
buttonS.clicked.connect(show)

buttonC = QPushButton()
buttonC.setText("Clear")
buttonC.setIcon(QIcon('im.jpg'))
buttonC.clicked.connect(line.clear)
```




Каждому графическому интерфейсу нужен какой-нибудь способ получения ввода от пользователя.

В PyQt такой способ ввода данных – это использование виджета QLineEdit.

Или если вы хотите получить ввод многострочного текста, вам нужно использовать виджет QTextEdit.

И конечно ввод как правило используется вместе с кнопкой QPushButton.

Чтобы получить введенный текст из виджета QLineEdit, вы должны использовать метод text.

Здесь мы создаем кнопку Submit, которая вызывает функцию show, использующую метод text виджета QLineEdit.

Мы также создаем кнопку Clear, которая вызывает метод clear виджета QLineEdit, который удаляет все его содержимое.

Метод SetEchoMode принимает несколько различных «режимов», одним из которых является режим пароля, который скрывает ввод.

Используя метод setFixedWidth, мы можем установить размер виджета QLineEdit в пикселях.

Значение по умолчанию для каждого виджета составляет около 100 пикселей.

Теперь о кнопке QPushButton.

Как следует из названия, это кнопка, которая запускает функцию при нажатии.

Кнопка, которая не связана с функцией, бесполезна.

Кнопки предназначены для подключения функции, которая будет выполняться после нажатия кнопки.

И такая функция подключается с помощью метода clicked.connect.

Также вы можете установить изображение на кнопку, с помощью виджета QIcon.

Просто передайте в него путь к файлу, и все готово.

```
def show():
   print(combo.currentText())


buttonS.clicked.connect(show)


combo = QComboBox()
combo.addItem("Python")
combo.addItem("Java")
combo.addItem("C++")
combo.setFixedWidth(140)


combo.addItems(["Python", "Java", "C++"])
```

Виджет QComboBox представляет собой раскрывающийся список элементов, из которых пользователь может выбрать свой вариант.

Преимущество этого виджета в том, что он занимает очень мало места на экране, при наличии большого списка элементов.

Здесь мы добавляем элементы в список по одному методом addItem.

Хотя мы можем добавить сразу все элементы кортежем с помощью метода addItems.

Зафиксировать ширину списка мы можем методом setFixedWidth.

Теперь, обработать выбор пользователем элемента в списке мы можем с помощью метода currentText(), который возвращает элемент списка в виде строки.

```
def show():
    print(check.text())
    print(check.checkState())
    print(check.isChecked())
```

```
check = QCheckBox()
check.setText("Option1")
check.stateChanged.connect(show)
```

Теперь перейдем к флажкам и радиокнопкам.

И здесь мы можем использовать сам флажок как кнопку, чтобы связать его с функцией обработки выбора флажка.

Мы делаем это с помощью метода stateChanged.connect.

Вы можете получить значение флажка с помощью метода text, который вернет текстовое значение флажка.

И вы можете использовать метод checkState, который возвращает целое число 0, если флажок не выбран и 2 – если он выбран.

Метод isChecked возвращает true, если флажок выбран.

```
def show():
    print(radio1.isChecked(), radio1.text())
    print(radio2.isChecked(), radio2.text())


radio1 = QRadioButton()
radio1.setText("Option1")
radio1.toggled.connect(show)
layout.addWidget(radio1)

radio2 = QRadioButton()
radio2.setText("Option2")
radio2.toggled.connect(show)
layout.addWidget(radio2)
```

В отличие от флажка, вы можете выбрать только одну радиокнопку из многих.

И здесь мы также можем использовать саму радиокнопку как кнопку, чтобы связать ее с функцией обработки выбора.

Мы делаем это с помощью метода toggled.connect.

Вы можете получить значение радиокнопки с помощью метода text, который вернет текстовое значение переключателя.

Есть еще один метод, который вы можете использовать – это метод isChecked, который возвращает true или false, показывая состояние выбора переключателя.

```
def show():
    print(text.toPlainText())

buttonS.clicked.connect(show)
```

```
text = QTextEdit()
text.setPlaceholderText("Enter some text here")
text.setUndoRedoEnabled(True)
layout.addWidget(text)
```

Как и виджет QLineEdit, виджет QTextEdit используется для ввода данных пользователем в виде текста.

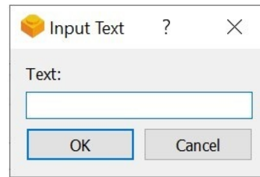
Однако, в отличие от QLineEdit, который вводит только одну строку, QTextEdit позволяет ввести несколько строк текста.

Чтобы получить от пользователя введенный текст, мы можем использовать метод toPlainText и метод кнопки clicked.connect.

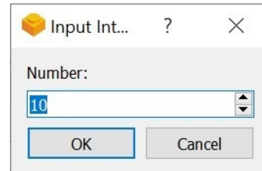
Метод setPlaceholderText используется для установки в виджет выделенного серым цветом текста, который исчезает при взаимодействии с виджетом.

Метод setUndoRedoEnabled отключает / включает возможность для пользователя использовать функции Undo и Redo с помощью клавиш Ctrl + Z и Ctrl + Y.

```
buttonD.clicked.connect(display)
```




```
def display():
   text , pressed = QInputDialog.getText(window, "Input Text", "Text: ",QLineEdit.Normal, "")
   if pressed:
      print(text)
```




```
intV, pressed = QInputDialog.getInt(window, "Input Integer","Number:", 10, 0, 100, 1)
```

Помимо строки ввода и текстовой области, диалоговые окна обычно используются для ввода данных пользователя.

И в PyQt5 есть виджет QInputDialog, который позволяет создавать множество различных диалоговых окон для ввода данных различными способами.

И самое распространенное диалоговое окно – это ввод текста.

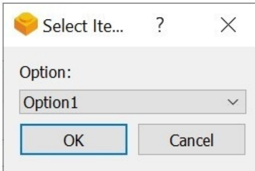
Оно представляет собой простое поле ввода и две кнопки, OK и Отмена.

Функция getText возвращает два значения, поэтому мы устанавливаем две переменные для них.

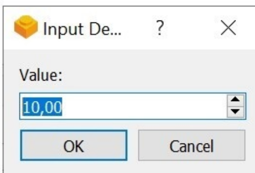
Первая переменная input получает ввод пользователя, вторая переменная получает значение True или False, указывающее, была ли нажата кнопка OK или нет.

Далее есть диалоговое окно ввода целого числа.

Здесь вы можете использовать стрелки для изменения целочисленного значения или непосредственно ввести значение самостоятельно.




```
options = ("Option1", "Option2", "Option3")
option, pressed = QInputDialog.getItem(window, "Select Item", "Option:", options, 0, False)
```



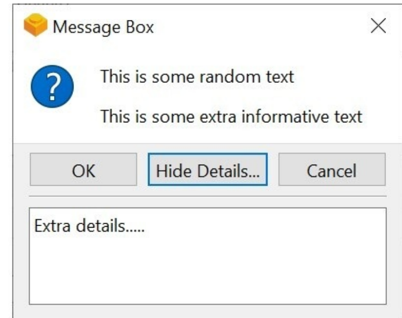

```
decimal, pressed = QInputDialog.getDouble(window, "Input Decimal", "Value:",10.00, 0, 100, 2)
```

Следующее окно – это окно выбора элемента.

И здесь мы сначала определяем список / кортеж строк, отображаемых в качестве элементов.

И последнее диалоговое окно – это окно ввода чисел с плавающей запятой.

Здесь вы также можете использовать стрелки для изменения значения или непосредственно ввести значение самостоятельно.




```
def show_popup():
    msg = QMessageBox()
    msg.setWindowTitle("Message Box")
    msg.setText("This is some random text")
    msg.setIcon(QMessageBox.Question)

    msg.setStandardButtons(QMessageBox.Cancel | QMessageBox.Ok)
    msg.setDefaultButton(QMessageBox.Ok)

    msg.setDetailedText("Extra details.....")
    msg.setInformativeText("This is some extra informative text")

    msg.buttonClicked.connect(popup)

    x = msg.exec_()

def popup(i):
    print(i.text())
```

QMessageBox – это виджет, который используется для создания диалоговых окон для отображения различных сообщений и кнопок.

Как правило, в приложении определяется метод, который открывает окно сообщения.

И этот метод связывается с какой-нибудь кнопкой или событием приложения.

Здесь метод setWindowTitle устанавливает заголовок окна сообщения.

Метод setText устанавливает текст под заголовком.

Метод exec открывает окно сообщения.

Окно сообщения имеет 4 различных типа значков, которые можно изменить с помощью метода setIcon.

В окне сообщения по умолчанию используется кнопка «ОК».

Однако на самом деле существует более десятка различных кнопок, которые QMessageBox предлагает для использования.

Используя функцию setStandardButtons, мы можем установить другой тип кнопки.

Когда окно открывается, вокруг кнопки «ОК» есть синий контур.

Это обозначение кнопки по умолчанию.

Используя функцию setDefaultButton, мы можем изменить кнопку по умолчанию.

По умолчанию у нас есть только одна область в QMessageBox, где мы показываем текст.

Однако есть еще две дополнительные области, которые мы можем разблокировать, чтобы добавить больше текста.

Первая область – это дополнительный текстовый раздел в самом окне сообщения, который мы можем определить с помощью функции setInformativeText.

Вторая область текста отображается в расширении окна.

При настройке этого раздела автоматически создается кнопка, которая используется для отображения этой области.

И для создания этой области потребуется функция setDetailedText.

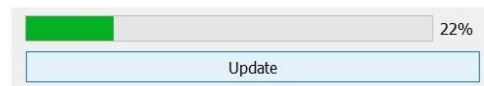
Теперь, если у нас есть 3 разные кнопки в окне сообщений, как мы узнаем, какая из них была нажата?

Используя метод buttonClicked.connect, мы можем вызвать запуск функции всякий раз, когда нажимается кнопка в окне сообщения.

Здесь когда нажимается кнопка окна сообщений, она автоматически запускает функцию popup, объявленную в buttonClicked.connect.

И вызов функции text кнопки вернет ее текстовое значение.

```
def update():
    value = prog_bar.value()
    prog_bar.setValue(value + 1)
```




```
prog_bar = QProgressBar()
prog_bar.setGeometry(50, 50, 250, 30)
prog_bar.setValue(0)
layout.addWidget(prog_bar)

buttonUp = QPushButton()
buttonUp.setText("Update")
buttonUp.clicked.connect(update)
layout.addWidget(buttonUp)
```

reset() - сбрасывает значение на индикаторе выполнения.

value() - возвращает текущее целочисленное значение.

setRange (n1, n2) - определяет начальную и конечную точки. Значения по умолчанию - 0 и 100 соответственно.

Виджет ProgressBar, индикатор выполнения – это отличный способ визуализировать процесс выполнения длительных операций, таких как передача файлов, загрузка, выгрузка, копирование и т. д.

Здесь с помощью метода setGeometry, мы определяем расположение и размеры индикатора выполнения.

Первые два параметра представляют положение X и Y индикатора выполнения в окне.

Третий и четвертый параметры – это ширина и высота индикатора выполнения.

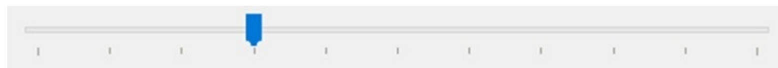
Метод setValue устанавливает значения индикатора.

Однако имейте в виду, что диапазон для индикатора выполнения составляет от 0 до 100.

Если вы хотите изменить этот диапазон, используйте метод setRange.

Метод reset сбрасывает значение на индикаторе выполнения.

Метод value возвращает текущее целочисленное значение индикатора.



```
slider = QSlider(Qt.Horizontal)
slider.setGeometry(50, 50, 200, 50)
slider.setMinimum(0)
slider.setMaximum(20)
slider.setTickPosition(QSlider.TicksBelow)
slider.setTickInterval(2)
slider.valueChanged.connect(show)
layout.addWidget(slider)

def show():
   print(str(slider.value()))
```

Виджет PyQt QSlider предоставляет графический интерфейс для выбора значения из диапазона различных значений.

У виджета Slider есть ползунок, который можно перемещать.

При перемещении ползунка выбранное значение соответственно изменяется.

По умолчанию ориентация слайдера вертикальная.

Для горизонтального ползунка вам нужно использовать значение Qt.Horizontal.

Затем нам нужно установить минимальный и максимальный диапазон с помощью методов setMinimum и setMaximum.

Используя метод setGeometry, мы определяем расположение и размеры слайдера.

Первые два параметра определяют положение ползунка по осям X и Y в окне.

Третий и четвертый параметры – это ширина и высота слайдера.

Затем нам нужно установить Ticks, маркеры по ползунку.

Вы можете выбрать место размещения тиков и установить интервал для расстояния между тиками.

Используя метод valueChanged, мы можем связать значение ползунка с функцией show.

Каждый раз, когда значение ползунка изменяется, вызывается функция show и показывается значение слайдера.

```
date = QDateEdit()
date.setMinimumDate(QDate(1900, 1, 1))
date.setMaximumDate(QDate(2100, 12, 31))
layout.addWidget(date)


def show():
   print(date.date().toPyDate())
```

Виджет PyQt5 QDateEdit – это интерактивный и визуальный способ ввести дату в приложении.

Здесь вы можете редактировать дату вручную или с помощью встроенных клавиш со стрелками.

И здесь вам не нужно назначать минимальное или максимальное значение, PyQt сам назначит значения по умолчанию.

Но вы можете сделать это с помощью методов setMinimumDate и setMaximumDate.

Чтобы получить введенную дату, вы можете использовать метод date и метод toPyDate, чтобы распечатать дату в более читаемом формате.

# Advanced PyQt

Создание настольных Python приложений с GUI

Advanced PyQt

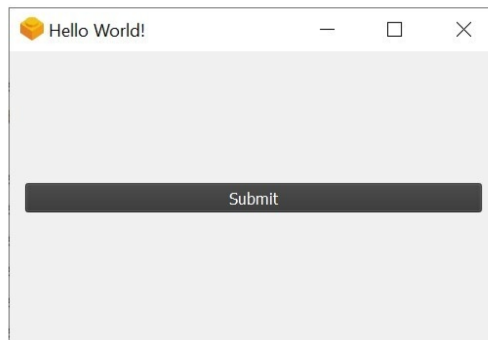
Одна из продвинутых возможностей Qt – это поддержка пользовательских стилей.

```
app = QApplication([])

#app.setStyle('Fusion’)

#app.setStyle('Windows’)

app.setStyle('WindowsVista')
```

Самый простой способ изменить внешний вид приложения – это установить глобальный стиль с помощью метода setStyle класса QApplication.

Доступные стили зависят от вашей платформы, но обычно это стили «Fusion», «Windows», «WindowsVista» и «Macintosh».

```
app = QApplication([])
app.setStyle('Fusion’)
dark_palette=app.palette()

button = QPushButton()
button.setText("Submit")
layout.addWidget(button)

dark_palette.setColor(QPalette.Window, QColor(53, 53, 53))
dark_palette.setColor(QPalette.WindowText, Qt.white)
dark_palette.setColor(QPalette.Base, QColor(25, 25, 25))
dark_palette.setColor(QPalette.AlternateBase, QColor(53, 53, 53))
dark_palette.setColor(QPalette.ToolTipBase, Qt.white)
dark_palette.setColor(QPalette.ToolTipText, Qt.white)
dark_palette.setColor(QPalette.Text, Qt.white)
dark_palette.setColor(QPalette.Button, QColor(53, 53, 53))
dark_palette.setColor(QPalette.ButtonText, Qt.white)
dark_palette.setColor(QPalette.BrightText, Qt.red)
dark_palette.setColor(QPalette.Link, QColor(42, 130, 218))
dark_palette.setColor(QPalette.Highlight, QColor(42, 130, 218))
dark_palette.setColor(QPalette.HighlightedText, Qt.black)

button.setPalette(dark_palette)
```



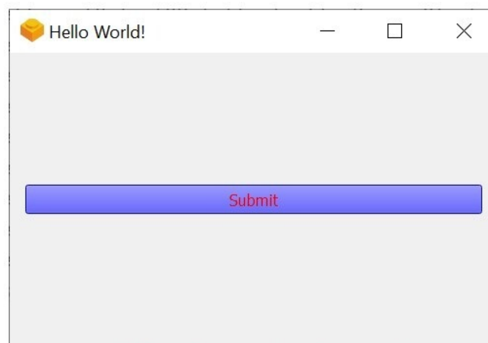

Если вам нравится глобальный стиль, но вы хотите изменить его цвета, вы можете использовать объект QPalette и метод setPalette.

Все виджеты в Qt содержат палитру и используют ее для рисования.

Это позволяет легко настраивать пользовательский интерфейс.

Здесь мы получаем объект палитры методом palette и модифицируем палитру.

Затем устанавливаем палитру для кнопки методом setPalette.




```
app.setStyleSheet("QPushButton { color:red; background-color: blue; }")
```

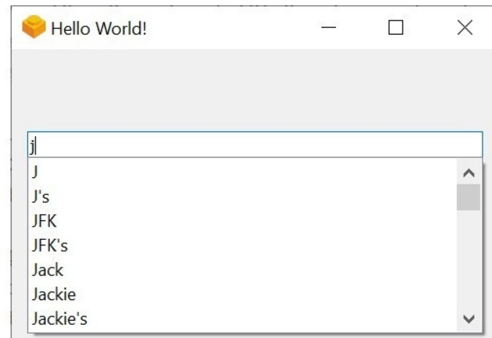
Также вы можете изменить внешний вид приложения с помощью таблиц стилей.

Это аналог CSS в Qt.

Здесь мы используем таблицы стилей, чтобы изменить фон и цвет надписи кнопки.

Терминология и синтаксические правила таблицы стилей Qt почти идентичны таблицам стилей CSS в HTML.

```
words = QFile('words')
words.open(QFile.ReadOnly | QFile.Text)
data = QTextStream(words).readAll()
list = data.split('\n');

completer = QCompleter(list)
completer.setCaseSensitivity(Qt.CaseInsensitive)
#completer.setModelSorting(QCompleter.CaseInsensitivelySortedModel)
#completer.setFilterMode(Qt.MatchContains)

line = QLineEdit()
line.setCompleter(completer)
layout.addWidget(line)

combo = QComboBox()
combo.setEditable(True)
combo.addItems(list)
combo.setCompleter(completer)
layout.addWidget(combo)
```




В строке поиска или ввода удобной опцией является автозаполнение текста.

И Qt имеет встроенную поддержку автозавершения текста с помощью класса QCompleter.

И QCompleter поддерживается виджетами QLineEdit и QComboBox.

Когда пользователь вводит текст, QCompleter предлагает возможные способы завершения ввода на основе списка слов, предоставленного моделью.

Модель может быть в простом случае просто списком слов.

Обычно вы создаете объект QCompleter, передавая модель в конструктор.

У виджетов, поддерживающих QCompleter, есть метод setCompleter, который позволяет указать средство завершения, которое будет использоваться для виджета.

QCompleter поддерживает ряд свойств для управления его поведением, включая, отсортирована ли модель, совпадения завершения должны быть основаны на том, чем введенная строка начинается, содержит или заканчивается, максимальное количество элементов, которые должны быть показаны для завершения, должно ли совпадение быть чувствительным к регистру.

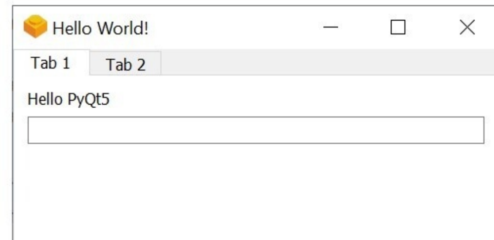
Здесь модель представляет собой файл слов, который загружается как ресурс.

Сначала мы открываем файл слов, читаем его и создаем список строк.

Затем создается завершитель, который создается с передачей списка строк в качестве модели.

И виджеты QLineEdit и QComboBox связываются с завершителем.

```
window = QTabWidget()
tab1 = QWidget()
tab2 = QWidget()
window.addTab(tab1, "Tab 1")
window.addTab(tab2, "Tab 2")
layout1 = QFormLayout()
layout2 = QFormLayout()
label1 = QLabel()
label1.setText("Hello PyQt5")
line1 = QLineEdit()
label2 = QLabel()
label2.setText("Hello PyQt5")
line2 = QLineEdit()
layout1.addRow(label1)
layout1.addRow(line1)
layout2.addRow(label2)
layout2.addRow(line2)
tab1.setLayout(layout1)
tab2.setLayout(layout2)
```




Если в пользовательском интерфейсе слишком много элементов для одновременного отображения, или интерфейс логически нужно разделить, элементы можно расположить на разных страницах, помещенных под каждой вкладкой виджета QTabWidget.

QTabWidget предоставляет панель вкладок и область страницы.

Отображается страница под первой вкладкой, а остальные страницы скрываются.

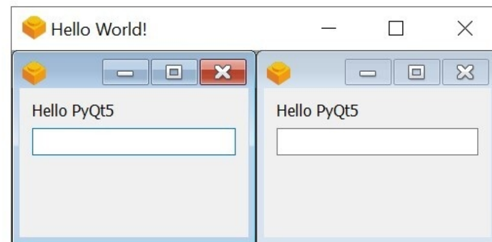
И пользователь может просмотреть любую страницу, щелкнув нужную вкладку.

В этом примере содержимое формы сгруппировано по двум страницам.

И каждая группа виджетов отображается на отдельной вкладке.

Само окно верхнего уровня – это QTabWidget. И в него добавлены две вкладки.

```
window = QMdiArea()
window.tileSubWindows()
#window.cascadeSubWindows()
sub1 = QMdiSubWindow()
sub1W = QWidget()
sub1.setWidget(sub1W)
sub2 = QMdiSubWindow()
sub2W = QWidget()
sub2.setWidget(sub2W)
layout1 = QFormLayout()
layout2 = QFormLayout()
label1 = QLabel()
label1.setText("Hello PyQt5")
line1 = QLineEdit()
label2 = QLabel()
label2.setText("Hello PyQt5")
line2 = QLineEdit()
layout1.addRow(label1)
layout1.addRow(line1)
layout2.addRow(label2)
layout2.addRow(line2)
sub1W.setLayout(layout1)
window.addSubWindow(sub1)
sub2W.setLayout(layout2)
window.addSubWindow(sub2)
```




Приложение с графическим интерфейсом пользователя может иметь несколько окон.

И один из способов одновременного отображения нескольких окон – создать их как независимые окна.

Это называется SDI (единый интерфейс документа).

Однако это требует больше ресурсов памяти, так как каждое окно может иметь свою собственную систему меню, панель инструментов и т. д.

Другой подход – это приложения MDI (Multiple Document Interface), которые потребляют меньше ресурсов памяти.

Здесь вспомогательные окна располагаются внутри основного контейнера относительно друг друга.

И виджет-контейнер называется QMdiArea.

Дочерние окна в этой области являются экземплярами класса QMdiSubWindow.

И можно установить любой QWidget как внутренний виджет объекта subWindow.

Подокна в области MDI могут быть расположены каскадом или мозаикой.

Здесь мы создаем QMdiArea и устанавливаем расположение подокон мозаикой.

Затем создаем подокна и устанавливаем в каждое подокно виджет с компоновкой.

И добавляем подокна в QMdiArea.

# PySide

**Создание настольных Python приложений с GUI**

PySide

PySide – это библиотека Python и обертка Qt для создания кроссплатформенных графических пользовательских интерфейсов.

Sign in

Page Discussion Read View source View history Search Qt Wiki

Main page
Recent changes
Random page
Help about MediaWiki

Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information

# Qt for Python

Jump to: navigation, search

En Ar Bg De El Es Fa Fi Fr Hi Hu It Ja Kn Ko Ms Nl Pl Pt Ru Sq Th Tr Uk Zh

## Qt for Python

Contents [hide]


1 Qt for Python
1.1 What does it look like?
2 Getting Started
3 Community
4 Development Status
5 Contributing to the Qt for Python Wiki


Py

Qt for Python official logo.

The Qt for Python project aims to provide a complete port of the PySide module to Qt. The development started on GitHub in May 2015. The project managed to port PySide to Qt 5.3, 5.4 & 5.5. During April 2016 The Qt Company decided to properly support the port (see details ).

The module was released mid June 2018 as a Technical Preview (supporting Qt 5.11), and it was officially released without the Technical Preview tag, in December 2018 for Qt 5.12. In December 2020, the module was released for Qt6, which is the latest available version, which has the following differences:

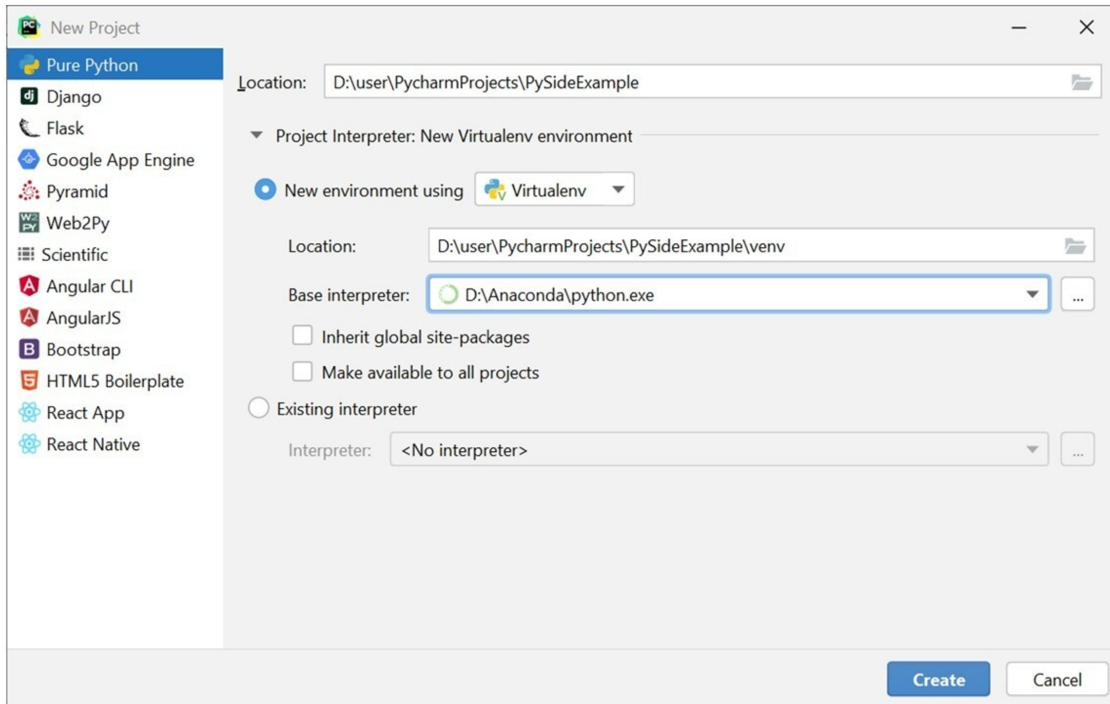
- It doesn't support Python 2.7,

В отличие от PyQt, библиотека

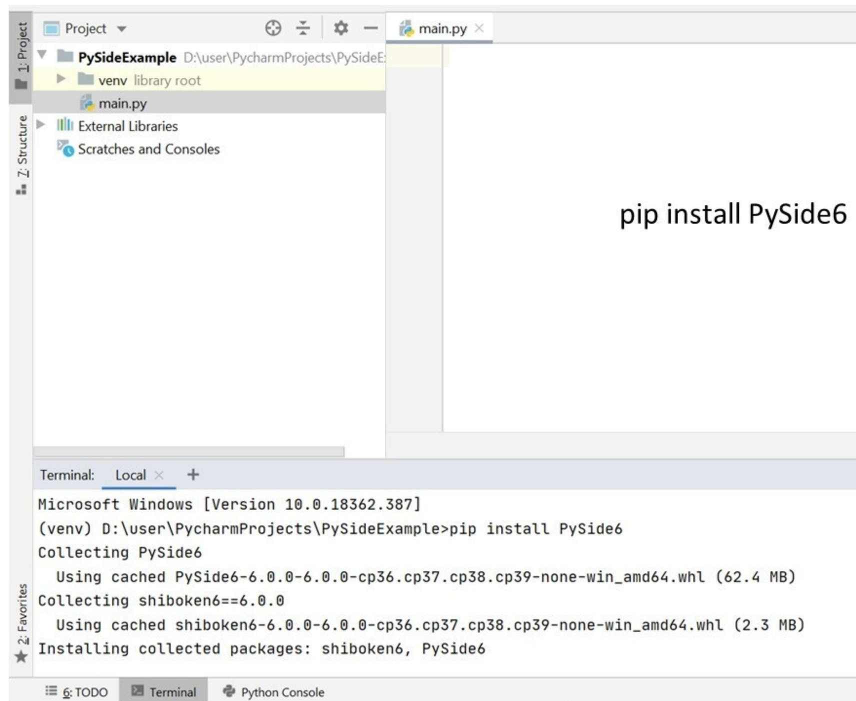
PySide доступна для свободного использования как в открытых, так и в коммерческих проектах, поскольку лицензирована по лицензии LGPL.

PySide реализован в виде набора модулей Python.

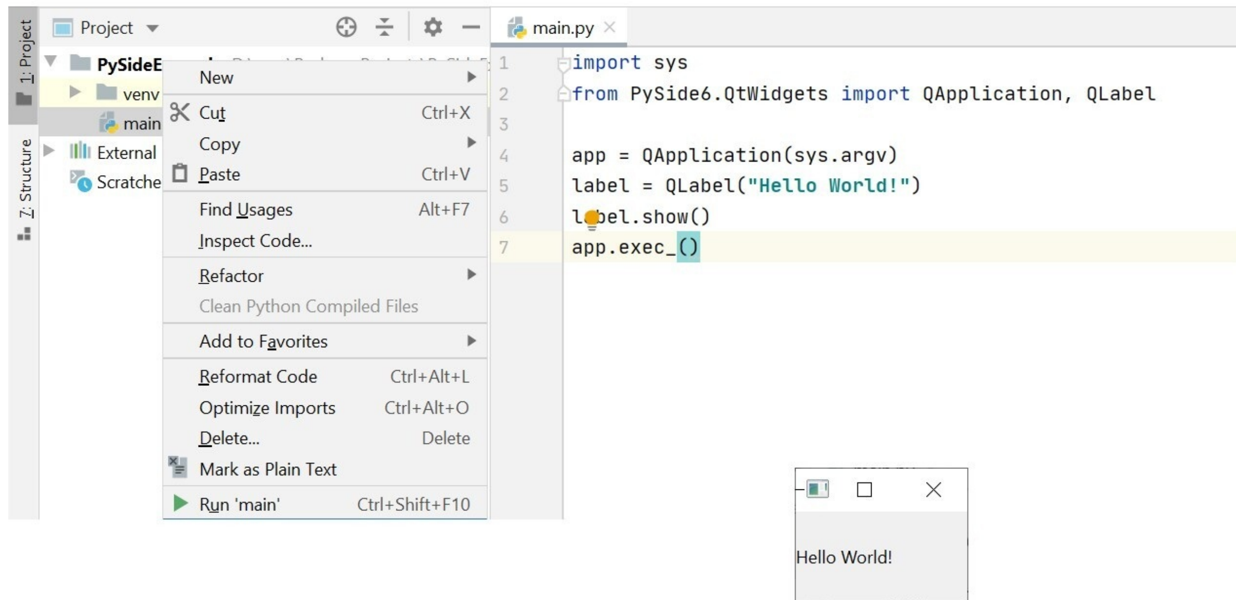
Эти модули предоставляют инструменты для работы с графическим интерфейсом пользователя, мультимедиа, XML-документами, сетью и базами данных.




Для создания PySide приложения создадим в PyCharm
проект.



Далее в окне терминала наберем команду установки библиотеки PySide.
pip install PySide6




После установки библиотеки, создадим в проекте файл питона и наберем в нем код.

Затем нажмем правой кнопкой мыши на файле и выберем Run.

В результате должно открыться окно PySide приложения.

```
import sys
from PySide6.QtWidgets import QApplication, QLabel

app = QApplication(sys.argv)
label = QLabel("Hello World!")
label.show()
app.exec_()
```

Для приложения, использующего PySide, вы всегда должны начинать с импорта соответствующего класса из модуля QtWidgets.

После импорта вы создаете экземпляр QApplication.

Так как Qt может получать аргументы из командной строки, вы можете передать любой аргумент объекту QApplication.

После создания объекта приложения мы создаем метку QLabel.

После создания метки мы вызываем для нее метод show.

И наконец, мы вызываем метод exec, чтобы войти в основной цикл Qt и начать выполнение кода Qt.

```
import sys
from PyQt5.QtWidgets import QApplication, QDialog

app = QApplication(sys.argv)

window = QDialog()
window.setGeometry(500, 300, 300, 200)
window.setWindowTitle('GUI Window')
window.show()

sys.exit(app.exec_())
```

```
import sys
from PySide6.QtWidgets import QApplication, QDialog

app = QApplication(sys.argv)

window = QDialog()
window.setGeometry(500, 300, 300, 200)
window.setWindowTitle('GUI Window')
window.show()

sys.exit(app.exec_())
```

Теперь, чем отличается PySide от PyQt?

Обе библиотеки являются обертками Python одной и той же среды графического интерфейса Qt.

PyQt – это более популярная библиотека, выпущенная раньше PySide.

И долгое время PyQt была единственной доступной библиотекой Python.

Однако в 2009 г. возник спор относительно того, под какой лицензией следует выпускать PyQt, между создателями PyQt и создателями Qt.

Поскольку группы не смогли прийти к соглашению, родилась новая Python библиотека PySide.

PySide была выпущена под лицензией LGPL, тогда как PyQt была выпущена под лицензией GPL.

Лицензия LGPL позволяет распространять код без необходимости делиться своим исходным кодом.

Это позволяет разрабатывать коммерческие приложения с помощью PySide.

Однако лицензия GPL не позволяет вам скрывать исходный код.

Короче говоря, если вы хотите разрабатывать и распространять коммерческие программы на PyQt, вам следует приобрести коммерческую лицензию на Qt.

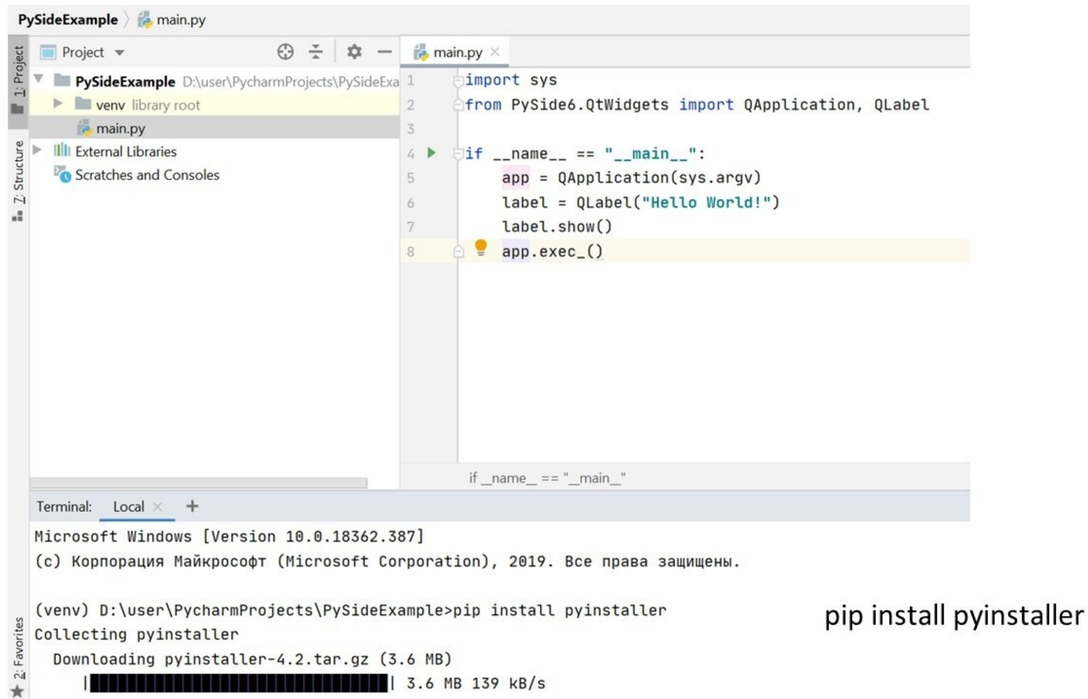
Вы все равно можете продавать программное обеспечение за деньги (без коммерческой лицензии), но вам придется поделиться исходным кодом.

Это неприемлемо для большинства коммерческих программ.

Сравнение PyQt и PySide показывает, что обе библиотеки примерно равны.

За исключением нескольких различий в синтаксисе в том, как происходит импорт и запуск, синтаксис обеих библиотек абсолютно одинаков.

И они используют одни и те же виджеты – виджеты Qt.




Теперь, как создать исполняемую программу PySide.

Для этого мы используем PyInstaller, инструмент который позволяет заморозить приложение Python в автономном исполняемом файле.

Этот установщик поддерживает Linux, macOS, Windows и другие операционные системы, а также совместим со сторонними модулями Python, такими как PySide6.

Поэтому сначала установим PyInstaller с помощью pip.

И в коде используем функцию main для запуска приложения.

```
pyinstaller main.py

pyinstaller --onefile main.py
```

Далее в окне терминала наберем pyinstaller

main

.py.

Этот процесс создает два каталога: dist / и build /.

Исполняемый файл приложения и необходимые библиотеки помещаются в каталог dist.

Чтобы запустить приложение, перейдите в dist и запустите программу.

В Windows PyInstaller имеет возможность создать сборку из одного файла, то есть один EXE-файл, который содержит весь ваш код, библиотеки и файлы данных в одном исполняемом файле.

Чтобы указать однофайловую сборку, укажите в командной строке флаг –onefile.

# Tkinter

Создание настольных Python приложений с GUI

Tkinter

Tkinter – это обертка Python библиотеки Tk GUI, которая поставляется вместе с Python.

# tkinter — Python interface to Tcl/Tk

**Source code:** Lib/tkinter/__init__.py

The tkinter package ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running `python -m tkinter` from the command line should open a window demonstrating a simple Tk interface, letting you know that tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

Tkinter – это стандартная библиотека графического интерфейса для Python.

И Tkinter – это интерфейс Python для Tk.

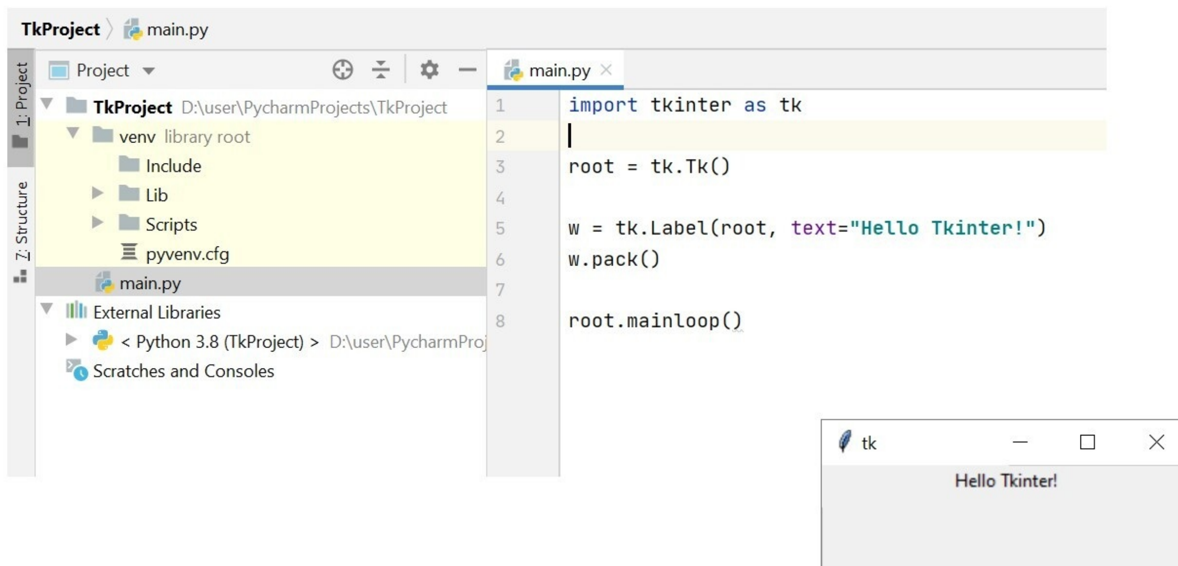
Tkinter – это аббревиатура от «Tk interface».

Tk был разработан Джоном Остерхаутом как расширение графического интерфейса для языка сценариев Tcl.

И Tk был очень популярен в 1990-х годах, так как его легче было изучать и использовать, чем другие наборы инструментов.

Поэтому неудивительно, что многие программисты захотели использовать Tk независимо от Tcl.

Вот почему были разработаны обертки для множества других языков программирования, включая Perl, Ada, Python, Ruby и Common Lisp.




Создание приложения с графическим интерфейсом пользователя с использованием Tkinter – это несложная задача.

Все, что вам нужно сделать, это выполнить следующие шаги -

Импортировать модуль Tkinter.

Создать главное окно приложения GUI.

Добавить один или несколько виджетов в приложение с графическим интерфейсом.

Войти в основной цикл событий, чтобы обрабатывать каждое событие, инициированное пользователем.

Чтобы инициализировать tkinter, мы должны создать корневой виджет Tk, который представляет собой окно со строкой заголовка.

Корневой виджет должен быть создан перед любыми другими виджетами, и может быть только один корневой виджет.

Метод pack сообщает Tk, что размер окна подгоняется под заданный текст.

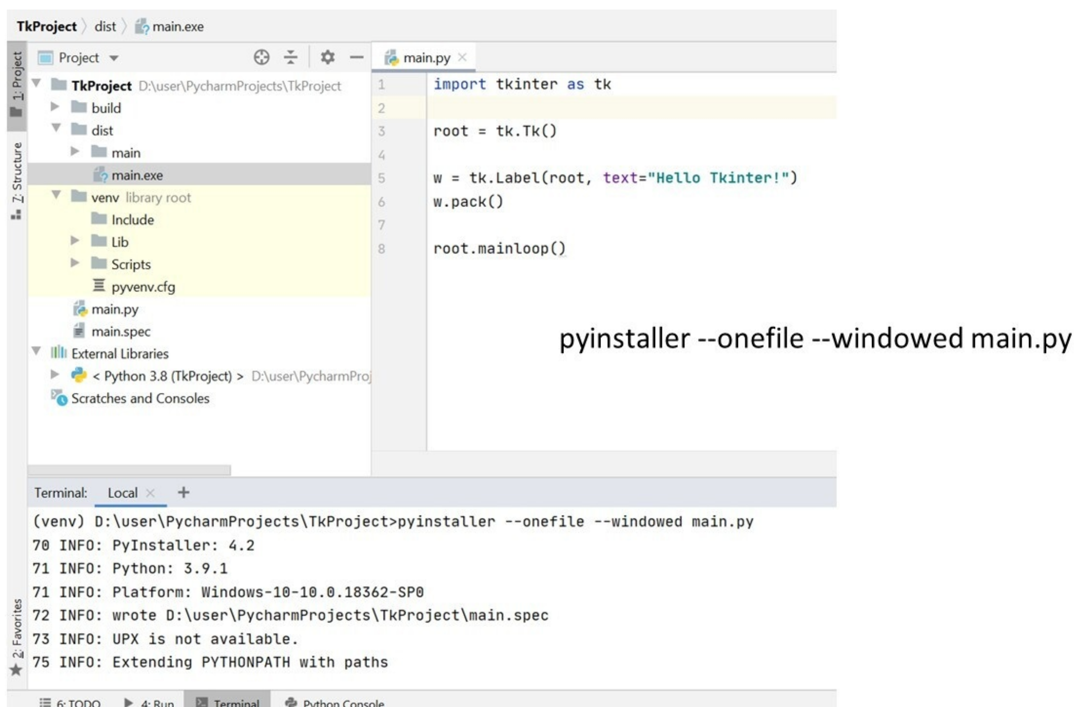
И окно не появится, пока мы не войдем в цикл событий Tkinter mainloop.

И наш скрипт останется в цикле событий, пока мы не закроем окно.

pip install pyinstaller

```
Terminal:   Local ×  +

(venv) D:\user\PycharmProjects\TkProject>pip install pyinstaller
Collecting pyinstaller
  Using cached pyinstaller-4.2.tar.gz (3.6 MB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done

6: TODO   4: Run   Terminal   Python Console
```

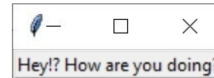
Теперь, как создать исполняемый файл для приложения Tkinter. Для начала, установим инструмент pyinstaller.



Затем в окне терминала наберем команду pyinstaller –onefile –windowed main.py

В результате в папке dist будет создан один исполняемый файл.

```
from tkinter import *

root = Tk()
var = StringVar()
label = Label( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```




Для создания графического интерфейса пользователя Tk предоставляет виджеты, окна верхнего уровня и три менеджера геометрии.

И мы начнем знакомство с Tk с одного из самых простых виджетов – с метки.

Метка – это виджет Tkinter, который используется для отображения текста или изображения.

И метка – это виджет, который пользователь просто просматривает, но не взаимодействует с ним.

Посмотрите на строку кода, которая содержит виджет Label.

Здесь первый параметр – это имя родительского окна, в нашем случае «root».

Так как наш виджет Label является потомком корневого виджета.

Параметр text или textvariable определяет текст, который будет отображаться.

Параметр relief задает внешний вид декоративной рамки вокруг метки. По умолчанию – FLAT.

Некоторые виджеты (например, виджеты ввода текста, переключатели и т. д.) могут быть напрямую связаны с переменными приложения с помощью специальных параметров: variable, textvariable, onvalue, offvalue, и value.

И это связывание работает в обоих направлениях: если переменная изменяется по какой-либо причине, виджет, к которому она подключена, будет обновлен, чтобы отразить новое значение.

Эти управляющие переменные Tkinter используются как обычные переменные Python для хранения определенных значений.

Но при этом невозможно передать обычную переменную виджету с помощью параметра variable или textvariable.

Единственные типы переменных, для которых это работает, – это переменные, которые являются подклассами класса Variable, определенного в модуле Tkinter.

Это переменные StringVar, IntVar, DoubleVar, BooleanVar.

Чтобы прочитать текущее значение такой переменной, можно вызвать метод get.

Значение такой переменной можно изменить с помощью метода set.

```
import tkinter as tk

master = tk.Tk()

whatever_you_do = "Whatever you do will be insignificant, but it is very
important that you do it.\n(Mahatma Gandhi)“

msg = tk.Message(master, text = whatever_you_do)
msg.config(bg='lightgreen', font=('times', 24, 'italic’))

msg.pack()
tk.mainloop()
```

Виджет Message можно использовать для отображения коротких текстовых сообщений.

И виджет сообщения аналогичен по своим функциям виджету Label, но более гибок в отображении текста, например можно изменить шрифт, в то время как виджет метки может отображать текст только одним шрифтом. Хотя невозможно использовать для текста более одного шрифта.

Если вам нужно отображать текст несколькими шрифтами, можно использовать виджет Text.

Здесь мы используем метод config чтобы установить фон сообщения и шрифт сообщения.

```
frame = Frame(root)
frame.pack()

buttonQ = tk.Button(frame, text="QUIT", fg="red", command=quit)
buttonQ.pack(side=LEFT)

buttonT = Button(frame, text="Hello", command=write_slogan)
buttonT.pack(side=LEFT)
```

Виджет Button используется для добавления кнопок в приложение Python.

Эти кнопки могут отображать текст или изображения, которые передают назначение кнопок.

И вы можете прикрепить функцию или метод к кнопке, который будет вызываться автоматически при нажатии кнопки.

Здесь мы создаем фрейм как родительский виджет для двух кнопок.

Первая кнопка с красной надписью QUIT, при нажатии на которую окно закрывается – это встроенная команда quit, прикрепленная к кнопке.

Ко второй кнопке прикреплен метод write_slogan, который печатает в вывод текст.

```
v = tk.IntVar()
v.set(1)

languages = [("Python", 101),("Perl", 102),("Java", 103),("C++",
104),("C", 105)]

def ShowChoice():
   print(v.get())

tk.Label(root, text="""Choose your favourite programming
language:""", justify = tk.LEFT, padx = 20).pack()

for language, val in languages:
   tk.Radiobutton(root, text=language, padx = 20, variable=v,
command=ShowChoice, value=val).pack(anchor=tk.W)
```

Радиокнопка представляет собой элемент графического пользовательского интерфейса Tkinter, который позволяет пользователю выбрать одну опцию из предопределенного набора опций.

Радиокнопки могут содержать текст или изображения.

И кнопка может отображать текст только одним шрифтом.

С кнопкой можно связать функцию Python.

Эта функция будет вызываться, если вы нажмете этот переключатель.

Таким образом, этот виджет реализует кнопку с множественным выбором, которая является способом предложить пользователю множество возможных вариантов выбора и позволяет пользователю выбрать только один из них.

Чтобы реализовать эту функциональность, каждая группа радиокнопок должна быть связана с одной и той же переменной, и каждая из кнопок должна представлять только одно значение.

Радиокнопки названы в честь физических кнопок, используемых на старых радиостанциях для выбора диапазонов волн или предустановленных радиостанций.

Если была нажата такая кнопка, другие кнопки выскакивали, оставляя нажатую кнопку единственной нажатой кнопкой.

И каждая группа виджетов радиокнопок должна быть связана с одной и той же переменной.

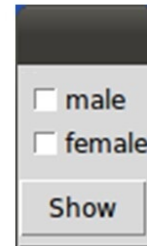
Нажатие кнопки изменяет значение этой переменной на заранее определенное значение.

Здесь у нас есть список «языков», который содержит тексты кнопок и соответствующие значения.

И мы можем использовать цикл for для создания всех переключателей.

Список кортежей содержит как тексты, так и значения, которые будут присвоены переменной v, если будет выбран соответствующий язык.

```
def print_states():
  print("male: %d,\nfemale: %d" % (var1.get(), var2.get()))

var1 = IntVar()
var2 = IntVar()

C1 = Checkbutton(root, text = "male", variable = var1, onvalue = 1, offvalue = 0, height=5, width = 20)
C2 = Checkbutton(root, text = "female", variable = var2, onvalue = 1, offvalue = 0, height=5, width = 20)
C1.pack()
C2.pack()

btn=Button(root, text='Show', command=print_states)
btn.pack()
```




Checkbutton или флажок представляет собой виджет, который позволяет пользователю делать множественный выбор из ряда различных опций.

Это отличается от радиокнопки, где пользователь может сделать только один выбор.

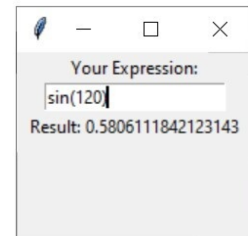
Обычно флажки отображаются на экране в виде квадратных полей, которые могут содержать галочки при выборе флажка.

Таким образом, флажок имеет два состояния: включен или выключен.

И флажок может содержать текст, но только одним шрифтом или изображение.

Параметр variable является управляющей переменной, которая отслеживает текущее состояние флажка.

Обычно эта переменная является переменной IntVar, и 0 означает очищено, а 1 означает установлено.




```
def evaluate(event):
   res.configure(text="Result: " + str(eval(entry.get())))

w = Tk()
Label(w, text="Your Expression:").pack()

entry = Entry(w)
entry.bind("<Return>", evaluate)
entry.pack()

res = Label(w)
res.pack()
```

Виджет ввода – это виджет Tkinter, используемый для получения ввода данных от пользователя приложения.

Этот виджет позволяет пользователю вводить одну строку текста.

Если пользователь вводит строку, длина которой превышает доступное пространство для отображения, содержимое будет прокручено.

Если вы хотите ввести несколько строк текста, вы должны использовать виджет Text.

И виджет ввода также ограничен одним шрифтом.

Как и с другими виджетами, можно дополнительно влиять на отображение виджета с помощью параметров.

Здесь, в этом примере, мы предоставляем графический интерфейс, который способен оценивать любое математическое выражение и печатать результат.

Приложение Tkinter большую часть времени работает внутри цикла событий, вход в который осуществляется с помощью метода mainloop.

Этот метод ждет событий.

И события могут быть нажатием клавиш или операциями мыши пользователя.

И Tkinter предоставляет механизм, позволяющий программисту иметь дело с событиями.

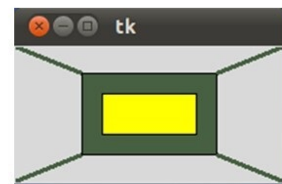
Для каждого виджета можно привязать функции Python к событию с помощью метода widget.bind (событие, обработчик)

Если определенное событие происходит в виджете, функция «обработчик» вызывается с объектом события.

В этом примере используется событие <Return>, когда пользователь нажимает клавишу Enter.

И с этим событием связывается метод evaluate, который вычисляет математическое выражение и печатает результат.

```
c = Canvas(w, width=200, height=100)
c.pack()

c.create_rectangle(50, 20, 150, 80, fill="#476042")
c.create_rectangle(65, 35, 135, 65, fill="yellow")
c.create_line(0, 0, 50, 20, fill="#476042", width=3)
c.create_line(0, 100, 50, 80, fill="#476042", width=3)
c.create_line(150,20, 200, 0, fill="#476042", width=3)
c.create_line(150, 80, 200, 100, fill="#476042", width=3)
```




```
def paint( event ):
  python_green = "#476042"
  x1, y1 = ( event.x - 1 ), ( event.y - 1 )
  x2, y2 = ( event.x + 1 ), ( event.y + 1 )
  c.create_oval( x1, y1, x2, y2, fill = python_green )

c.bind( "<B1-Motion>", paint )
```

Виджет Canvas предоставляет возможность рисования таких графических объектов, как линии, круги, изображения и даже другие виджеты.

С помощью Canvas можно рисовать графики, создавать графические редакторы и реализовывать различные виды пользовательских виджетов.

Метод create_line используется для рисования прямой линии.

Координаты здесь представлены четырьмя целыми числами: x1, y1, x2, y2.

Это означает, что прямая идет от точки (x1, y1) к точке (x2, y2).

После этих координат следует разделенный запятыми список дополнительных параметров, который может быть пустым.

Здесь мы можем установить, например, цвет линии.

Для создания прямоугольников есть метод create_rectangle.

Координаты снова определяются двумя точками, но на этот раз первая – это верхняя левая точка и далее нижняя правая точка прямоугольника.

Метод create_text можно применить к объекту холста, чтобы написать на нем текст.

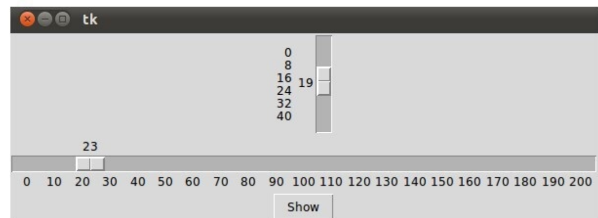
Также есть методы create_arc, create_image, create_oval, create_polygon.

Также можно организовать интерактивное рисование на холсте.

К сожалению, нарисовать одну точку на холсте невозможно.

Но мы можем решить эту проблему, используя небольшой овал.

Здесь мы связываем методом bind с холстом событие мыши и метод pain, который будет рисовать на холсте.




```
def show_values():
   print (sl1.get(), sl2.get())

sl1 = Scale(w, from_=0, to=42, tickinterval=8)
sl1.set(19)
sl1.pack()
sl2 = Scale(w, from_=0, to=200, length=600,tickinterval=10,
orient=HORIZONTAL)
sl2.set(23)
sl2.pack()
Button(w, text='Show', command=show_values).pack()
```

Слайдер или ползунок – это объект Tkinter, с помощью которого пользователь может установить значение, перемещая индикатор.

И слайдеры могут быть расположены вертикально или горизонтально.

Ползунок создается с помощью метода Scale.

В качестве параметров можно установить минимальное и максимальное значения, а также разрешение.

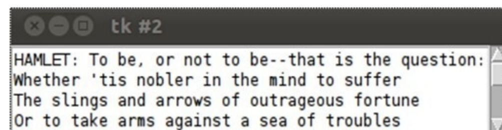
И мы также можем определить, хотим ли мы расположить слайдер вертикально или горизонтально.

Виджет Scale – альтернатива виджету Entry, если пользователь должен ввести число из конечного диапазона, то есть предустановленное числовое значение.

И мы можем увеличить длину слайдера с помощью параметра length, который определяет размер x, если масштаб горизонтальный, и размер y, если масштаб вертикальный.

Методом get мы можем получить значение ползунка.

```
S = Scrollbar(w)
T = Text(w, height=4, width=50)

S.pack(side=RIGHT, fill=Y)
T.pack(side=LEFT, fill=Y)

S.config(command=T.yview)
T.config(yscrollcommand=S.set)
T.tag_configure('big', font=('Verdana', 20, 'bold’))

quote = """HAMLET: To be, or not to be--that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune
That flesh is heir to. 'Tis a consummation
Devoutly to be wished."""“

T.insert(END, quote, 'big')
```




Текстовый виджет используется для ввода многострочного текста.

Текстовые виджеты также могут использоваться как простые текстовые редакторы или даже веб-браузеры.

Кроме того, текстовые виджеты могут использоваться для отображения ссылок, изображений и HTML, даже с использованием стилей CSS.

Здесь мы создаем текстовый виджет с помощью метода Text.

И мы устанавливаем высоту, то есть количество строк, и ширину, то есть количество символов.

И мы можем применить метод insert, чтобы добавить текст в виджет.

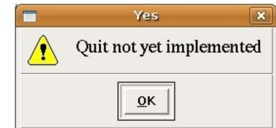
Также мы добавляем полосу прокрутки в наше окно.

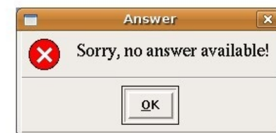
С этой целью Tkinter предоставляет метод Scrollbar.

И мы вызываем его с корневым объектом в качестве единственного параметра.

И мы связываем полосу прокрутки с текстовым виджетом с помощью параметра yscrollcommand.

```
from tkinter import messagebox as mb
def answer():
  mb.showerror("Answer", "Sorry, no answer available")

def callback():
  if mb.askyesno('Verify', 'Really quit?'):
    mb.showwarning('Yes', 'Not yet implemented')
  else:
    mb.showinfo('No', 'Quit has been cancelled')

Button(text='Quit', command=callback).pack(fill=X)
Button(text='Answer', command=answer).pack(fill=X)
```






Tkinter предоставляет набор диалогов, которые можно использовать для отображения окон сообщений, предупреждений или ошибок или виджетов для выбора файлов и цветов.

Существуют также простые диалоги, в которых пользователю предлагается ввести строку, целые числа или числа с плавающей запятой.

Диалоги предоставляются подмодулем messagebox в tkinter и его функциями.

Функция askokcancel спрашивает, следует ли продолжить операцию и возвращает истину или ложь.

Функция askquestion задает вопрос.

Функция askretrycancel спрашивает, нужно ли повторить операцию и возвращает истину, если ответ положительный.

Функция askyesno задает вопрос и возвращает истину, если ответ положительный.

Функция askyesnocancel также задает вопрос и возвращает true, если ответ положительный, или None, если он отменен.

Функция showerror показывает сообщение об ошибке.

Функция showinfo показывает информационное сообщение.

Функция showwarning показывает предупреждающее сообщение.

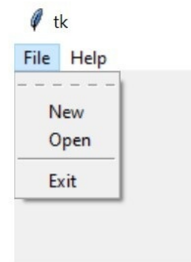
```
from tkinter import filedialog as fd
from tkinter.colorchooser import askcolor

def callback():
    name = fd.askopenfilename()
    print(name)
    result = askcolor(color="#6A9662", title="Bernd's Colour Chooser")
    print(result)

Button(text='File Open',command=callback).pack(fill=X)
```

Для чтения из файла или записи в файл Tkinter предоставляет модуль tk
f
ile
d
ialog и его функцию askopenfilename.

А для выбора цвета Tkinter предоставляет всплывающее меню.

Для этого мы должны импортировать модуль colorchooser и использовать метод askColor.

```
def NewFile():
    print("New File!")
def OpenFile():
    name = fd.askopenfilename()
    print(name)
def About():
    print("This is a simple example of a menu")

menu = Menu(w)
w.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label="File", menu=filemenu)
filemenu.add_command(label="New", command=NewFile)
filemenu.add_command(label="Open", command=OpenFile)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=quit)

helpmenu = Menu(menu)
menu.add_cascade(label="Help", menu=helpmenu)
helpmenu.add_command(label="About", command=About)
```




Tkinter представляет возможность создавать раскрывающиеся меню, то есть списки в верхней части окна, которые появляются или раскрываются, если вы нажимаете на такой элемент, как, например, «Файл», «Редактировать» или «Справка».

Меню добавляется в окно с помощью метода config и параметра menu.

Далее метод add_cascade создает новое иерархическое меню, связывая данное меню с родительским меню.

Метод add_command добавляет пункт меню в меню.

При этом параметр command связывает с пунктом меню функцию.

```
mb= Menubutton (w, text="condiments", relief=RAISED )
mb.pack(fill=Y)
mb.menu = Menu ( mb, tearoff = 0 )
mb["menu"] = mb.menu

mayoVar = IntVar()
ketchVar = IntVar()

mb.menu.add_checkbutton ( label="mayo", variable=mayoVar )
mb.menu.add_checkbutton ( label="ketchup", variable=ketchVar )
```

Кнопка меню – это часть раскрывающегося меню, которое постоянно отображается на экране.

И кнопка меню может отображать варианты выбора для этого меню, когда пользователь нажимает на нее.

Здесь параметр relief определяет эффекты затенения границ кнопки.

Метод add_checkbutton добавляет флажок как пункт меню, так как каждая кнопка меню связана с виджетом меню.

```
w = tk.Label(root, text="Red Sun", bg="red", fg="white")
w = tk.Label(root, text="Green Grass", bg="green", fg="black")
w = tk.Label(root, text="Blue Sky", bg="blue", fg="white")

w.pack(fill=tk.X, padx=10)
w.pack(fill=tk.X, pady=10)
w.pack(ipadx=10)
w.pack(ipadx=10)
w.pack(padx=5, pady=10, side=tk.LEFT)
```

Теперь давайте познакомимся с менеджерами компоновки или менеджерами геометрии, как их также иногда называют.

Tkinter имеет три менеджера компоновки pack, grid и place.

И не разрешено смешивать три менеджера компоновки в одном главном окне!

Менеджеры геометрии выполняют различные функции, такие как расположение виджетов на экране, регистрация виджетов в системе, управление отображением виджетов на экране.

Размещение виджетов на экране включает в себя определение размера и положения компонентов.

Виджеты могут предоставлять информацию о размере и выравнивании менеджерам геометрии, но менеджеры геометрии всегда имеют последнее слово по позиционированию и размеру.

Pack – самый простой в использовании из трех менеджеров геометрии.

Вместо того, чтобы точно объявлять, где должен отображаться виджет на экране дисплея, мы можем объявить позиции виджетов относительно друг друга с помощью команды pack.

Параметр expand команды pack – если установлено значение true, виджет расширяется, чтобы заполнить любое пространство родительского виджета.

Параметр fill определяет, заполняет ли виджет дополнительное пространство, выделенное ему, или сохраняет свои собственные минимальные размеры: NONE (по умолчанию), X (заливка только по

горизонтали), Y (заливка только по вертикали) или BOTH (заливка как по горизонтали, так и по вертикали).

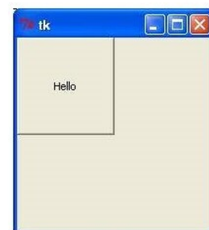
Параметр side определяет, какая сторона родительского виджета общая: TOP (по умолчанию), BOTTOM, LEFT или RIGHT.

Параметр padx – внешний отступ по горизонтали.

Параметр pady – внешний отступ по вертикали.

Параметр ipadx – внутренний отступ по горизонтали.

Параметр ipady – внутренний отступ по вертикали.




```
B = Tkinter.Button(top, text ="Hello", command = helloCallBack)

B.pack()

B.place(bordermode=OUTSIDE, height=100, width=100)
```

Компоновка Place позволяет явно установить положение и размер виджета в абсолютном выражении или относительно другого виджета.

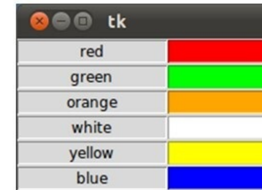
То есть этот менеджер геометрии упорядочивает виджеты, помещая их в определенную позицию в родительском виджете.

Параметр anchor метода place позволяет указать привязку виджета, это может быть N, E, S, W, NE, NW, SE или SW, то есть направления по компасу, указывающие углы и стороны виджета, по умолчанию это NW (левый верхний угол).

Параметр bordermode указывает, что другие параметры игнорируют границу родителя или нет.

Параметры height, width указывают высоту и ширину в пикселях.

Параметры x, y указывают смещение по горизонтали и вертикали в пикселях.




```
colours = ['red','green','orange','white','yellow','blue']

r = 0
for c in colours:
    tk.Label(text=c, relief=tk.RIDGE, width=15).grid(row=r,column=0)
    tk.Entry(bg=c, relief=tk.SUNKEN, width=10).grid(row=r,column=1)
    r = r + 1
```

Компоновка Grid помещает виджеты в двухмерную таблицу, которая состоит из ряда строк и столбцов.

И положение виджета определяется номером строки и столбца.

То есть этот менеджер геометрии организует виджеты в виде таблицы в родительском виджете.

Виджеты с одинаковым номером столбца и разными номерами строк будут располагаться друг над другом.

Соответственно, виджеты с одним и тем же номером строки, но с разными номерами столбцов будут находиться в одной «строке» и будут располагаться рядом друг с другом.

При этом размер сетки не нужно определять, потому что менеджер автоматически определяет наилучшие размеры для используемых виджетов.

```
frame = Frame(root)
frame.pack()

bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )

blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)
```

Виджет Frame работает как контейнер, который отвечает за расположение других виджетов.

И фрейм также можно использовать в качестве базового класса для реализации сложных виджетов.

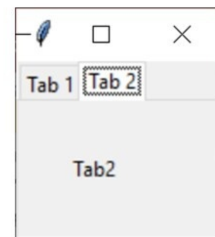
```
Lb1 = Listbox(top)
Lb1.insert(1, "Python")
Lb1.insert(2, "Perl")
Lb1.insert(3, "C")
Lb1.insert(4, "PHP")
Lb1.insert(5, "JSP")
Lb1.insert(6, "Ruby")

Lb1.pack()
```

Виджет Listbox используется для отображения списка элементов, из которого пользователь может выбрать несколько элементов.

Метод get возвращает кортеж, содержащий текст строк с индексами от первого до последнего включительно.

Если второй аргумент опущен, возвращает текст строки, ближайшей к первой.

```
import tkinter as tk
from tkinter import *
from tkinter import ttk

root = tk.Tk()
root.title("Tab Widget")
tabControl = ttk.Notebook(root)
tab1 = ttk.Frame(tabControl)
tab2 = ttk.Frame(tabControl)
tabControl.add(tab1, text='Tab 1')
tabControl.add(tab2, text='Tab 2')
tabControl.pack(expand=1, fill="both")
ttk.Label(tab1, text='Tab1').grid(column=0, row=0, padx=30, pady=30)
ttk.Label(tab2, text='Tab2').grid(column=0, row=0, padx=30, pady=30)
root.mainloop()
```




Модуль Tkinter предлагает широкий спектр виджетов, которые можно использовать для разработки приложений с графическим интерфейсом.

Модуль tkinter.ttk служит улучшением уже существующего модуля tk.

И модуль Ttk предоставляет 18 виджетов, 12 из которых есть в модуле Tkinter.

Добавленные виджеты – это Combobox, Notebook, Sizegrip, Progressbar, Separator и Treeview.

Здесь мы показываем создание виджета с вкладками.

Виджет ttk.Notebook управляет коллекцией окон и отображает их по одному.

Каждое дочернее окно связано с вкладкой.

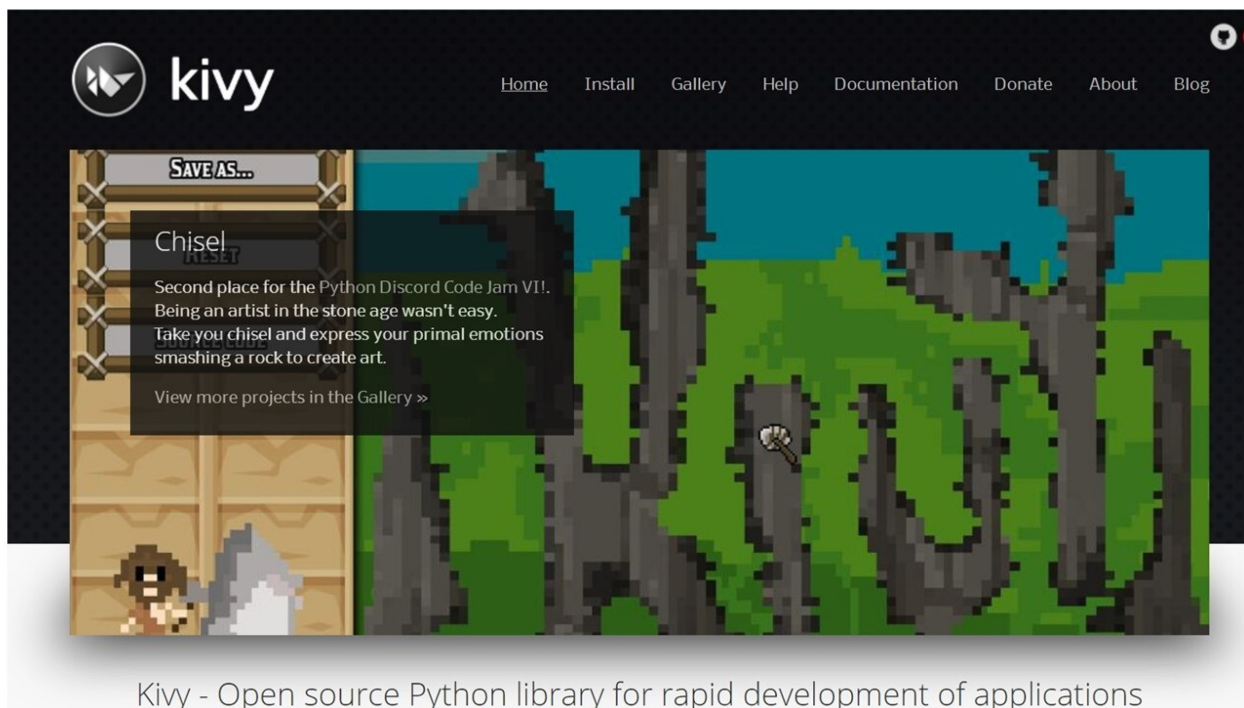
И пользователь может выбирать по одной вкладке за раз, чтобы просмотреть содержимое окна.

# Kivy

Создание настольных Python приложений с GUI

Kivy

Kivy – это многоплатформенная среда разработки приложений для Python, которая позволяет разрабатывать приложения для Windows, Linux, Android, macOS, iOS и Raspberry Pi.




Kivy позволяет разрабатывать мобильные приложения с поддержкой мультитач и дает возможность создать приложение один раз и использовать его на всех устройствах.

Она также позволяет получать доступ к мобильным API-интерфейсам для управления такими вещами, как камера на телефоне, отслеживание GPS, вибратор и т. д.

Kivy включает в себя различные модули для воспроизведения видео файлов и потоков.

Kivy использует широкий спектр виджетов, поддерживающих мультитач и жесты.

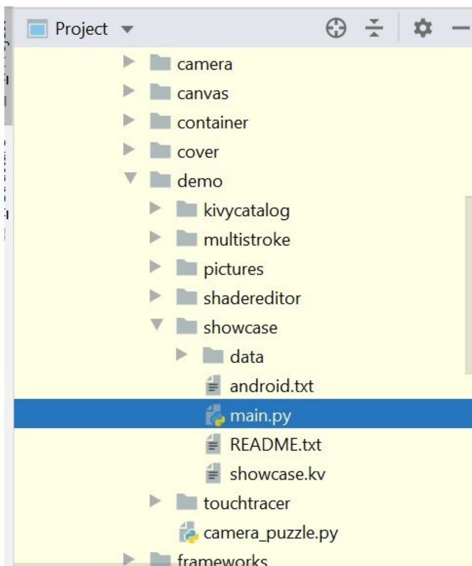
Киви использует PyGame для простого создания игр.

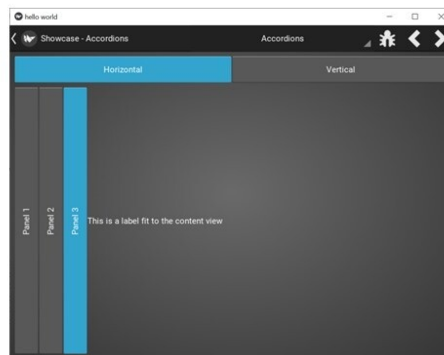
И PyGame поставляется с различными модулями для рисования форм, работы с цветами и воспроизведения музыки.

pip install kivy[base] kivy_examples

```
Terminal:   Local ×   +
Microsoft Windows [Version 10.0.18362.387]
(c) Корпорация Майкрософт (Microsoft Corporation), 2019. Все права защищены.

(venv) C:\Users\user\PycharmProjects\KivyProject>pip install kivy[base] kivy_examples
Collecting kivy[base]
  Downloading Kivy-2.0.0-cp39-cp39-win_amd64.whl (4.1 MB)
     |████████████████████████████████| 4.1 MB 3.3 MB/s
Collecting kivy_examples
  Downloading Kivy_examples-2.0.0-py2.py3-none-any.whl (9.2 MB)
     |████████████████████████████████| 9.2 MB 1.3 MB/s
Collecting kivy-deps.glew~=0.3.0
  Downloading kivy_deps.glew-0.3.0-cp39-cp39-win_amd64.whl (123 kB)
     |████████████████████████████████| 123 kB 1.6 MB/s
6: TODO   Python Console   Terminal
```

Для разработки Киви приложения создадим проект в PyCharm
и в окне терминала наберем команду установки Киви.

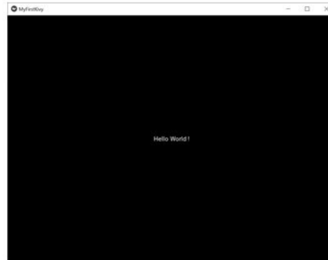


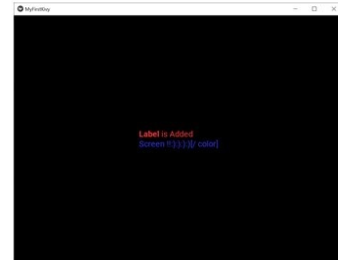




\share\kivy-examples\demo\showcase\main.py

Далее мы можем запустить готовые примеры приложений Киви.

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

# Defining a class
class MyFirstKivyApp(App):

    # Function that returns
    # the root widget
    def build(self):
        # Label with text Hello World is
        # returned as root widget
        return Label(text="Hello World !")

MyFirstKivyApp().run()
```




```
Label(text="[color=ff3333][b]Label[/b] is
Added [/color]\n[color=3333ff]Screen
!!:):):):)[/ color]",font_size='20sp',
markup=True)
```

И мы можем сами теперь создавать Киви приложения.

Прежде всего мы должны импортировать kivy.

Теперь, чтобы создать интерфейс Kivy, нам нужно импортировать модуль приложения App.

Далее импортируем метку.

И определим класс с функцией, которая возвращает корневой виджет – метку.

Затем инициализируем класс и вызовем его метод run, запустив приложение Киви.

К метке мы также можем применить стиль, то есть увеличить текст, размер, цвет и многое другое.

Вы можете изменить стиль текста с помощью разметки текста Text Markup.

**https://kivy.org/doc/stable/guide/packaging.html**

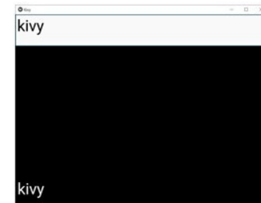
Programming Guide » Packaging your application

- Create a package for Windows
  - Requirements
- PyInstaller default hook
  - Packaging a simple app
  - Single File Application
  - Bundling Data Files
  - Packaging a video app with gstreamer
- Overwriting the default hook
  - Including/excluding video and audio and reducing app size
  - Alternate installations
- Create a package for Android
  - Buildozer
  - Packaging with python-for-android
  - Packaging your application for the Kivy Launcher
  - Release on the market
  - Targeting Android
- Kivy on Android
  - Package for Android
  - Debugging your application on the Android platform
  - Using Android APIs
  - Status of the Project and Tested Devices
- Creating packages for OS X
  - Using the Kivy SDK

Упаковка Киви приложения в исполняемый файл для различных платформ достаточно сложная и не всегда обходится без сложностей и ошибок.

Подробные инструкции для упаковки приложения можно найти в документации.

```
def build(self):
    b = BoxLayout(orientation='vertical')
    # Adding the text input
    t = TextInput(font_size=50,
            size_hint_y=None,
            height=100)
    f = FloatLayout()
    # By this you are abel to move the
    # Text on the screen to anywhere you want
    s = Scatter()
    l = Label(text="Hello !", font_size=50)
    f.add_widget(s)
    s.add_widget(l)
    b.add_widget(t)
    b.add_widget(f)
    # Binding it with the label
    t.bind(text=l.setter('text'))
    return b
```




Виджет TextInput предоставляет поле для редактирования простого текста.

Этот виджет поддерживает Unicode, multiline, курсорную навигацию, выделение и буфер обмена.

И TextInput использует две разные системы координат – (x, y) – координаты в пикселях, в основном используются для рендеринга на экране и (col, row) – индекс курсора в символах / строках, используемый для выделения и перемещения курсора.

Чтобы создать однострочный текстовый ввод, можно установить для свойства TextInput.multiline значение False.

Здесь параметр size_hint – это набор значений, используемых макетами для управления размерами своих дочерних элементов. Он указывает размер относительно размера макета, а не абсолютный размер в пикселях.

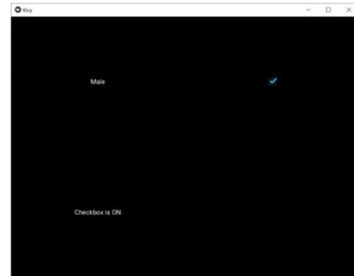
Если вы не хотите использовать size_hint для ширины или для высоты, установите значение параметра None.

В этом примере Scatter используется для создания интерактивных виджетов, которые можно поворачивать и масштабировать двумя или более пальцами на системе мультитач.

Компоновка BoxLayout размещает виджеты вертикально один над другим или горизонтально один за другим.

И методом bind мы связываем значения двух виджетов – метки и поля ввода.

```
def build(self):
    layout = GridLayout()
    layout.cols=2
    layout.add_widget(Label(text='Male'))
    self.active = CheckBox(active=True)
    layout.add_widget(self.active)
    self.lbl_active = Label(text='Checkbox is on')
    layout.add_widget(self.lbl_active)
    self.active.bind(active=self.on_checkbox_Active)
    return layout

  def on_checkbox_Active(self, checkboxInstance, isActive):
    if isActive:
        self.lbl_active.text = "Checkbox is ON"
        print("Checkbox Checked")
    else:
        self.lbl_active.text = "Checkbox is OFF"
        print("Checkbox unchecked")
```



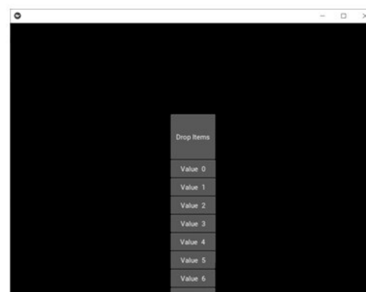

Виджет CheckBox – это особая кнопка с двумя состояниями, которую можно установить или снять.

Если флажок находится в группе, он становится переключателем, в этом случае одновременно можно выбрать только одну кнопку.

Здесь мы для класса приложения устанавливаем два свойства active, с которыми связываем флажок и метку.

Затем с помощью метода bind мы связываем с флажком функцию, которая обрабатывает выбор флажка.

В результате при выборе флажка изменяется свойство active класса, которое связано с меткой.




```
import kivy
from kivy.uix.dropdown import DropDown
from kivy.uix.button import Button
from kivy.base import runTouchApp

# create a dropdown with 10 buttons
dropdown = DropDown()
for index in range(10):
    btn = Button(text='Value % d' % index, size_hint_y=None, height=40)
    btn.bind(on_release=lambda btn: dropdown.select(btn.text))
    dropdown.add_widget(btn)
# create a main button
mainbutton = Button(text='Drop Items', size_hint=(None, None), pos=(350, 300))
mainbutton.bind(on_release=dropdown.open)
dropdown.bind(on_select=lambda instance, x: setattr(mainbutton, 'text', x))
# If you pass only a widget in runtouchApp(), a Window will
# be created and your widget will be added to the window
# as the root widget.
runTouchApp(mainbutton)
```

Выпадающий список можно использовать с другими виджетами. Потому что можно отображать список виджетов под отображаемым виджетом.

В отличие от других наборов, список виджетов может содержать виджет любого типа: простые кнопки, изображения и т. д.

При добавлении виджетов в список нам нужно указать высоту вручную (отключив size_hint_y), чтобы раскрывающийся список мог вычислить необходимую площадь.

И все кнопки в раскрывающемся списке будут запускать метод DropDown.select().

После его вызова текст основной кнопки будет отображать выбор из раскрывающегося списка.

Здесь в этом примере мы создаем раскрывающийся список DropDown и добавляем в него кнопки методом add_widge.

При этом мы связываем с кнопками метод dropdown.select, передавая в него текст кнопки.

Далее мы создаем виджет – кнопку и связываем с ней метод dropdown.open.

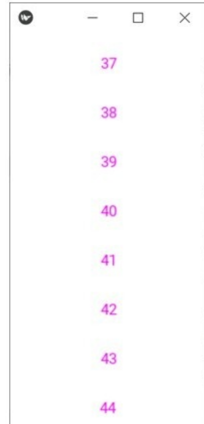
Таким образом, при нажатии кнопки список раскрывается.

И в этом примере мы запускаем приложение с помощью метода runTouchApp.

Это статическая функция, запускающая цикл приложения.

Если вы передадите в runtouchApp виджет, будет создано окно, и виджет будет добавлен в это окно как корневой виджет.

```
layout = GridLayout(cols=1, spacing=10, size_hint_y=None)
# Make sure the height is such that there is something to scroll.
layout.bind(minimum_height=layout.setter('height'))

Window.clearcolor = (1, 1, 1, 1)
Window.size = (200, 400)

for i in range(100):
   lbl = Label(text=str(i), size_hint_y=None, height=40, color=(1,0,1,1))
   layout.add_widget(lbl)

root = ScrollView(size_hint=(1, None), size=(Window.width, Window.height))
root.add_widget(layout)

runTouchApp(root)
```




Виджет ScrollView предоставляет прокручиваемое окно просмотра.

И ScrollView принимает только один дочерний элемент и применяет к нему прокрутку в соответствии со свойствами scroll_x и scroll_y.

По умолчанию ScrollView позволяет прокручивать по осям X и Y.

Вы можете явно отключить прокрутку по оси, установив для свойств do_scroll_x или do_scroll_y значение False.

И вы должны тщательно указать размер вашего контента, чтобы получить желаемый эффект прокрутки.

По умолчанию size_hint равен (1, 1), поэтому размер содержимого точно соответствует вашему ScrollView и вам нечего будет прокручивать.

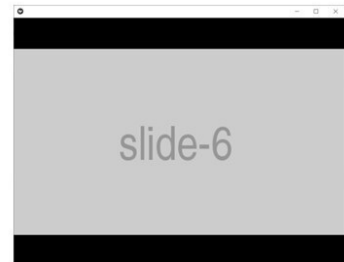
Вы должны деактивировать хотя бы одну из инструкций size_hint (x или y) дочернего элемента, чтобы включить прокрутку.

Здесь, чтобы прокрутить GridLayout по оси Y по вертикали, установим ширину дочернего элемента равной ширине ScrollView (size_hint_x = 1) и установим для свойства size_hint_y значение None.

Здесь мы также используем класс Window – базовый класс для создания окна Kivy по умолчанию.

Мы используем свойство clearcolor, чтобы установить фон окна и свойство size, чтобы установить размер окна.

Для метки мы используем параметр color, чтобы установить цвет текста.




```
root = Carousel(direction ='right', loop=True)
for i in range(10):
    src = "http://placehold.it/480x270.png&text=slide-%d&.png" %i
    # using Asynchronous image
    image = AsyncImage(source = src, allow_stretch = True)
    root.add_widget(image)

Clock.schedule_interval(root.load_next, 1)
runTouchApp(root)
```

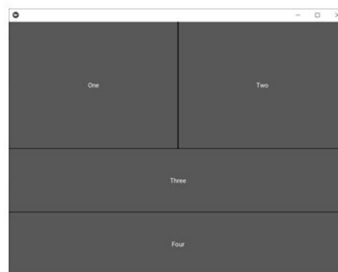
Виджет «Карусель» представляет собой классический удобный для мобильных устройств вид карусели, в котором можно перемещаться между слайдами.

Вы можете добавить любой контент в карусель и заставить его перемещаться по горизонтали или вертикали.

Этот виджет может содержать изображения, видео или любой другой контент.

Здесь мы асинхронно загружаем изображения в карусель и используем класс Clock, чтобы автоматически перемещаться между слайдами.

```
superBox = BoxLayout(orientation='vertical')
HB = BoxLayout(orientation='horizontal')

btn1 = Button(text="One")
btn2 = Button(text="Two")

HB.add_widget(btn1)
HB.add_widget(btn2)

VB = BoxLayout(orientation='vertical')

btn3 = Button(text="Three")
btn4 = Button(text="Four")

VB.add_widget(btn3)
VB.add_widget(btn4)

superBox.add_widget(HB)
superBox.add_widget(VB)

runTouchApp(superBox)
```




```
btn1 = Button(text='Hello', size=(200, 100), size_hint=(None,
None))
btn2 = Button(text='Kivy', size_hint=(.5, 1))
btn3 = Button(text='World', size_hint=(.5, 1))
```

Компоновка BoxLayout упорядочивает виджеты либо вертикально, либо горизонтально.

Ориентацию компоновки определяет параметр orientation.

И если вы не укажете размер, тогда дочерние виджеты делят размер своего родительского виджета поровну.

Если указать параметр size_hint, он будет использовать доступное пространство после вычитания всех виджетов фиксированного размера.

```
def on_value(instance,brightness):
    brightnessValue.text = "% d" % brightness

brightnessControl = Slider(min = 0, max = 100,orientation ='vertical',
value_track = True,value_track_color =[1, 0, 0, 1])
layout = GridLayout(cols=4)
# 1st row - one label, one slider
layout.add_widget(Label(text='brightness'))
layout.add_widget(brightnessControl)

# 2nd row - one label for caption,
# one label for slider value
layout.add_widget(Label(text='Slider Value'))
brightnessValue = Label(text='0')
layout.add_widget(brightnessValue)

# On the slider object Attach a callback
# for the attribute named value
brightnessControl.bind(value=on_value)

runTouchApp(layout)
```

Виджет Slider используется для увеличения яркости, громкости и т. д.

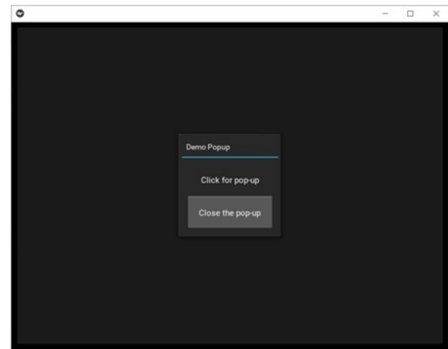
Этот виджет поддерживает горизонтальную и вертикальную ориентацию, минимальные / максимальные значения и значение по умолчанию.

Здесь параметр value_track определяет, должен ли ползунок рисовать линию между значением min и значением ползунка.

Параметр value_track_color определяет цвет этой линии в формате rgba.

И методом bind мы связываем со значением ползунка функцию on_value, в которой мы изменяем текст метки.

```
def onButtonPress(button):
    layout = GridLayout(cols=1, padding=10)
    popupLabel = Label(text="Click for pop-up")
    closeButton = Button(text="Close the pop-up")
    layout.add_widget(popupLabel)
    layout.add_widget(closeButton)
    # Instantiate the modal popup and display
    popup = Popup(title='Demo
Popup',content=layout,auto_dismiss=False, size_hint=(None,
None), size=(200, 200))
    popup.open()
    # Attach close button press with popup.dismiss action
    closeButton.bind(on_press=popup.dismiss)

Config.set('graphics', 'resizable', True)
layout = GridLayout(cols = 1, padding = 10)
button = Button(text="Click for pop-up")
layout.add_widget(button)
# Attach a callback for the button press event
button.bind(on_press=onButtonPress)

runTouchApp(layout)
```




Виджет Popup используется для создания всплывающих окон.

По умолчанию всплывающее окно покрывает все «родительское» окно.

Когда вы создаете всплывающее окно, вы должны как минимум установить Popup.title и Popup.content.

Всплывающие диалоговые окна используются, когда мы должны передать пользователю определенные сообщения.

Всплывающее окно Popup – это особый виджет.

Не пытайтесь добавить его как дочерний элемент к любому другому виджету.

Если вы это сделаете, всплывающее окно будет обрабатываться как обычный виджет и не будет скрыто в фоновом режиме.

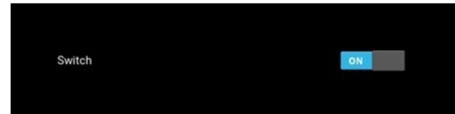
Если вы не хотите, чтобы всплывающее окно отображалось в полноэкранном режиме, вы должны либо указать size_hint со значениями меньше 1 (например, size_hint = (. 8, .8)), либо отключить size_hint и использовать атрибуты фиксированного размера.

По умолчанию любой щелчок за пределами всплывающего окна закрывает его.

Если вы этого не хотите, вы можете установить параметр auto_dismiss как False.

Чтобы вручную закрыть всплывающее окно, можно использовать команду dismiss.

```
def switch_callback(switchObject, switchValue):
    # Switch value are True and False
    if (switchValue):
        print('Switch is ON:):):)')
    else:
        print('Switch is OFF:(:(:(')

switch=Switch(active = False)
layout = GridLayout(cols=2, padding=10)
layout.add_widget(Label(text ="Switch"))
layout.add_widget(switch)
switch.bind(active = switch_callback)

runTouchApp(layout)
```




Виджет Switch это переключатель, который может быть в одном из состояний – активен или неактивен, как механический выключатель света.

И пользователь может провести пальцем влево или вправо, чтобы активировать или деактивировать переключатель.

Значение, выдаваемое переключателем – это True или False.

И к переключателю может быть присоединена функция обратного вызова, чтобы получить значение переключателя.

Когда переключатель выполняет переход из состояния в состояние, запускается обратный вызов, и любое другое действие может быть выполнено в зависимости от состояния.

```
def on_spinner_select(spinner, text):
    label.text = "Selected: %s" % spinner.text
    print('The spinner', spinner, 'have text', text)

layout = FloatLayout()
spinner = Spinner(text ="Python",
        values =("Python", "Java", "C++", "C", "C#", "PHP"),
        background_color =(0.784, 0.443, 0.216, 1))
spinner.size_hint = (0.3, 0.2)
spinner.pos_hint = {'x': .35, 'y': .75}
layout.add_widget(spinner)
spinner.bind(text = on_spinner_select)
label = Label(text="Selected: %s" % spinner.text)
layout.add_widget(label)
label.pos_hint = {'x': .0, 'y': .3}

runTouchApp(layout)
```

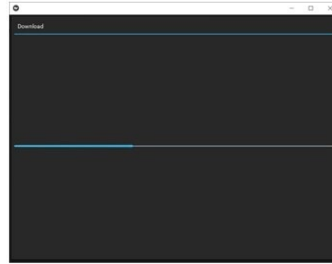
Spinner – это виджет, который позволяет быстро выбрать одно значение из предопределенного набора значений.

В состоянии по умолчанию спиннер показывает текущее выбранное значение.

При касании спиннера отображается раскрывающийся список со всеми другими доступными значениями, из которых пользователь может выбрать значение.

И к спиннеру методом bind может быть прикреплен обратный вызов для получения уведомлений о выборе значения пользователем.

```
def pop(instance):
    progress_bar.value = 1
    popup.open()

def next(dt):
    if progress_bar.value >= 100:
        return False
    progress_bar.value += 1

def puopen(instance):
    Clock.schedule_interval(next, 1 / 25)

layout=BoxLayout()
progress_bar = ProgressBar()
popup = Popup(title ='Download',content = progress_bar)
popup.bind(on_open = puopen)
layout.add_widget(Button(text ='Download', on_release = pop))
runTouchApp(layout)
```



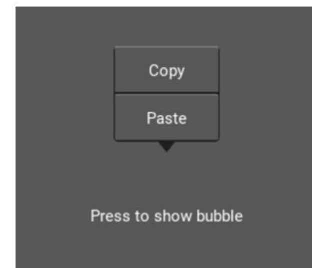

Виджет ProgressBar используется для визуализации прогресса долго выполняющейся задачи.

В настоящее время поддерживается только горизонтальный режим.

Индикатор выполнения не имеет интерактивных элементов и является виджетом только для отображения прогресса выполнения.

Здесь мы показываем в всплывающем окне прогресс бар и при открытии окна запускаем таймер, который выполняет функцию увеличения значения прогресс бара.

```
class BubbleDemo(FloatLayout):

    def __init__(self, **kwargs):
        super(BubbleDemo, self).__init__(**kwargs)
        self.but = Button(text='Press to show bubble')
        self.but.bind(on_release=self.show_bubble)
        self.add_widget(self.but)

    def show_bubble(self, *l):
        self.bubb = bubb = Bubble(orientation = 'vertical',size_hint=(None, None),
                                  size=(300, 200),
                                  pos_hint={'center_x': .5, 'y': .6})
        bubb.add_widget(Button(text='Copy'))
        bubb.add_widget(Button(text='Paste'))
        self.add_widget(self.bubb)

class BubbleApp(App):
    def build(self):
        return BubbleDemo()
if __name__ == '__main__':
    BubbleApp().run()
```




Виджет Bubble – это меню или небольшое всплывающее окно, в котором пункты меню расположены вертикально или горизонтально.

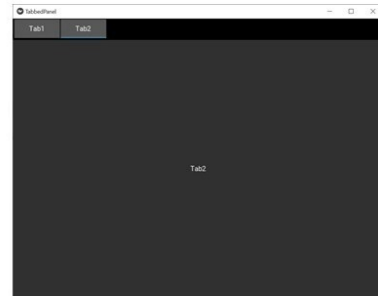
И этот виджет содержит стрелку, указывающую в выбранном направлении.

Чтобы выбрать направление стрелки, используется параметр arrow_pos виджета.

Ориентация виджета по умолчанию горизонтальная, но вы можете изменить ее с помощью параметра orientation.

Чтобы добавить элементы в виджет, используется метод add_widget.

```
class TabbedPanelApp(App):
    def build(self):
        tp = TabbedPanel(do_default_tab=False)
        th1 = TabbedPanelHeader(text='Tab1')
        layout1=FloatLayout()
        layout1.add_widget(Label(text='Tab1'))
        th1.content=layout1
        tp.add_widget(th1)
        th2 = TabbedPanelHeader(text='Tab2')
        layout2 = FloatLayout()
        layout2.add_widget(Label(text='Tab2'))
        th2.content=layout2
        tp.add_widget(th2)
        return tp

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




Виджет TabbedPanel предоставляет вкладки, с областью заголовка для кнопок вкладок и областью содержимого для отображения содержимого текущей вкладки.

И TabbedPanel предоставляет одну вкладку по умолчанию, которую можно отключить параметром do_default_tab.

Отдельная вкладка называется TabbedPanelHeader.

Это специальная кнопка, содержащая контент.

Сначала вы добавляете TabbedPanelHeader, а затем устанавливаете ее содержимое.

Этим контентом может быть любой виджет.

Это может быть компоновка с иерархией виджетов или отдельный виджет, например метка или кнопка.

```
class ScatterApp(App):
    def build(self):
        root = FloatLayout()
        scatter = Scatter()
        scatter.add_widget(Label(text='Scatter'))
        root.add_widget(scatter)
        return root

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

Виджет

Scatter используется для создания интерактивных виджетов, которые можно перемещать, вращать и масштабировать двумя или более пальцами в системе мультитач.

Дочерние элементы виджета располагаются также как и в RelativeLayout.

Таким образом, при перетаскивании Scatter положение дочерних элементов не меняется, изменяется только положение Scatter.

И размер Scatter не влияет на размер его дочерних элементов.

Если вы хотите изменить размер Scatter, используйте scale, а не size .

Параметр scale трансформирует как Scatter, так и его дочерние элементы.

И Scatter – это не компоновка. Вы сами должны управлять размером дочерних виджетов.

По умолчанию Scatter не имеет графического представления – это только контейнер.

Идея состоит в том, чтобы объединить Scatter с другим виджетом, например с изображением или иерархией виджетов.

```
def animate(instance):
    animation = Animation(pos=(100, 100), t='out_bounce')
    animation += Animation(pos=(200, 100), t='out_bounce')
    animation &= Animation(size=(500, 500))
    animation += Animation(size=(100, 50))
    animation.start(instance)

layout = FloatLayout()
button = Button(size_hint =(None, None), text ='Animate', on_press = animate)
layout.add_widget(button)
runTouchApp(layout)
```

Классы Animation и AnimationTransition используются для анимации свойств виджета.

И вы должны указать имя свойства и целевое значение.

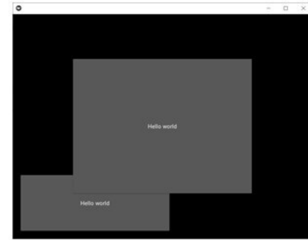
Чтобы использовать анимацию, нужно создать объект Animation и далее нужно использовать объект Animation в виджете.

Здесь мы создаем объект анимации.

Этот объект можно сохранить и повторно использовать в разных виджетах.

Здесь + – это последовательный шаг, а & – параллельный шаг.

Далее мы применяем анимацию к кнопке, переданную в аргументе "instance".




```
layout = FloatLayout()

button1 = Button(text='Hello world',size_hint=(.5, .25), pos=(20, 20))

button2 = Button(text='Hello world', size_hint=(.6, .6), pos_hint={'x':.2, 'y':.2})

layout.add_widget(button1)
layout.add_widget(button2)
runTouchApp(layout)
```

Компоновка Floatlayout позволяет размещать элементы относительно друг друга в зависимости от текущего размера и высоты окна.

То есть Floatlayout позволяет размещать элементы, используя так называемое относительное положение.

Это означает, что вместо того, чтобы определять конкретную позицию или координаты, мы будем размещать виджеты, используя процент от размера окна.

И когда мы изменяем размеры окна, все, что помещается в окне, соответственно изменяет свой размер и положение.

Это делает приложение более надежным и масштабируемым в соответствии с размером окна.

При использовании FloatLayout мы используем свойства pos_hint и size_hint дочерних элементов.

По умолчанию все виджеты имеют свой size_hint = (1, 1).

Чтобы создать кнопку с шириной 50% и 25% от высоты компоновки и расположенную в точке (20, 20), мы определяем size_hint=(.5, .25) и pos=(20, 20).

Чтобы создать кнопку, нижний левый угол которой будет находиться минус 20% с каждой стороны, мы определяем pos_hint={'x':.2, 'y':.2}.

```
layout = GridLayout(cols=2, row_force_default=True, row_default_height=40)
layout.add_widget(Button(text='Hello 1', size_hint_x=None, width=100))
layout.add_widget(Button(text='World 1'))
layout.add_widget(Button(text='Hello 2', size_hint_x=None, width=100))
layout.add_widget(Button(text='World 2’))

runTouchApp(layout)
```

Gridlayout – это компоновка, которая упорядочивает дочерние элементы в матричном формате.

Она берет доступное пространство (квадрат) и делит это пространство на строки и столбцы, а затем добавляет виджеты в соответствии с полученными ячейками.

Здесь мы не можем явно разместить виджет в определенном столбце / строке.

Каждому дочернему элементу автоматически назначается определенная позиция.

И для этой компоновки нужно указать как минимум количество столбцов или строк.

Если мы не укажем столбцы или строки, программа выдаст исключение.

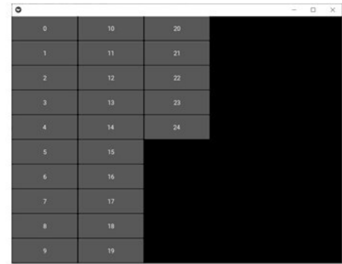
Первоначальный размер компоновки задается свойствами col_default_width и row_default_height.

Мы можем установить размер по умолчанию, используя свойство col_force_default или row_force_default.

Это заставит макет игнорировать размеры дочерних элементов и использовать размер по умолчанию компоновки.

Чтобы настроить размер одного столбца или строки, используйте cols_minimum или rows_minimum.

```
root = StackLayout(orientation ='tb-lr')
for i in range(25):
    btn = Button(text=str(i), size_hint =(.2, .1))
    root.add_widget(btn)
runTouchApp(root)
```



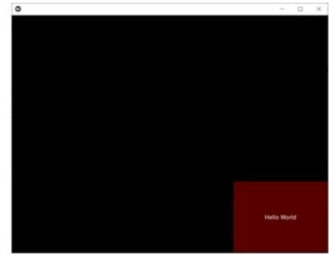
['lr-tb', 'tb-lr', 'rl-tb', 'tb-rl', 'lr-bt', 'bt-lr', 'rl-bt', 'bt-rl']

Компоновки StackLayout и Boxlayout очень похожи.

И здесь и там можно упорядочивать виджеты вертикально или горизонтально.

Но с помощью StackLayout вы можете комбинировать ориентации.

Имеется 4 ориентации по строкам и 4 по столбцам, это справа налево или слева направо, сверху вниз или снизу вверх.



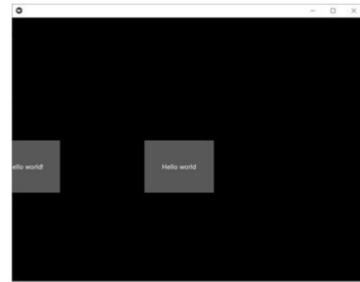

```
layout = AnchorLayout(anchor_x='right', anchor_y='bottom')
btn = Button(text='Hello World',size_hint=(.3, .3),background_color=(1.0, 0.0, 0.0, 1.0))
layout.add_widget(btn)

runTouchApp(layout)
```

Компоновка AnchorLayout размещает свои дочерние элементы по границе контейнера (сверху, снизу, слева, справа) или по центру.

AnchorLayout можно инициализировать с помощью параметров:

anchor_x и anchor_y – могут быть переданы значения «слева», «справа» и «по центру».

Таким образом, здесь для выбора места размещения виджетов в родительском контейнере, есть 9 различных областей компоновки, верхний левый, верхний центр, верхний правый, центральный левый, центральный, центральный правый, нижний левый, нижний центральный и нижний правый.




```
layout = RelativeLayout()

btn1 = Button(text='Hello world', size_hint =(.2, .2), pos =(300.0, 200.0))
btn2= Button(text='Hello world!', size_hint =(.2, .2), pos =(-50.0, 200.0))
layout.add_widget(btn1)
layout.add_widget(btn2)

runTouchApp(layout)
```

Компоновка Relativelayout похожа на компоновку FloatLayout, разница в том, что дочерние виджеты компоновки Relativelayout располагаются относительно компоновки, а не относительно окна.

Эта компоновка работает так же, как FloatLayout, но свойства позиционирования (x, y, center_x, right, y, center_y и top) относятся к размеру компоновки, а не к размеру окна.

И дочерние виджеты перемещаются при изменении положения компоновки.

То есть координаты дочернего виджета остаются такими же, так как они всегда определяются относительно родительской компоновки.

И параметры pos_hint – x, center_x, right, y, center_y и top используются для выравнивания виджета по краям или центрирования независимо от размера окна.

Таким образом, эта компоновка позволяет устанавливать относительные координаты для дочерних виджетов.

Если вам нужно абсолютное позиционирование, используйте FloatLayout.

И в RelativeLayout необходимо указать размер и положение каждого дочернего виджета.

```
layout = PageLayout()

btn1 = Button(text='Page 1')
btn1.background_color = get_color_from_hex('#FF0000')
btn2 = Button(text='Page 2')
btn2.background_color = get_color_from_hex('#00FF00')

layout.add_widget(btn1)
layout.add_widget(btn2)

runTouchApp(layout)
```

Компоновка PageLayout работает иначе, чем другие макеты.

Эта компоновка позволяет перелистывать страницы, используя свои границы.

Идея состоит в том, что все страницы сложены друг над другом, и мы видим ту страницу, которая находится сверху.

Таким образом класс PageLayout используется для создания многостраничного интерфейса таким образом, чтобы можно было легко переключаться с одной страницы на другую с помощью перелистывания.

# wxPython

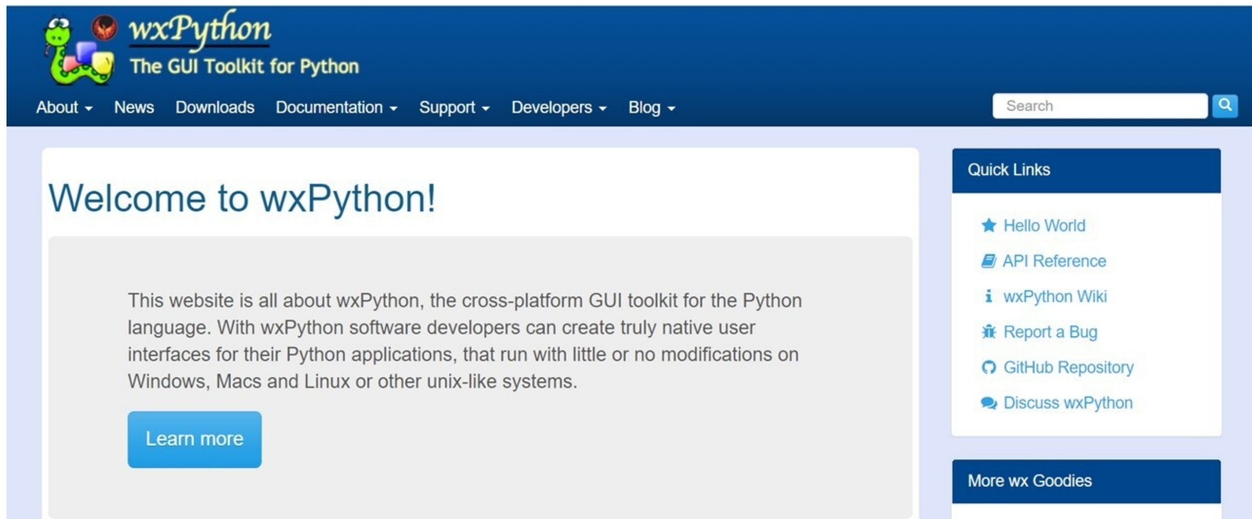
Создание настольных Python приложений с GUI

wxPython

Библиотека WxPython – это питон обёртка библиотеки C++ графического интерфейса wxWidgets.

Оригинальная библиотека wxWidgets, написанная на C ++, – это огромная библиотека классов.

И классы графического интерфейса пользователя из этой библиотеки переносятся на Python с помощью модуля wxPython, который пытается максимально точно отразить исходную библиотеку wxWidgets.

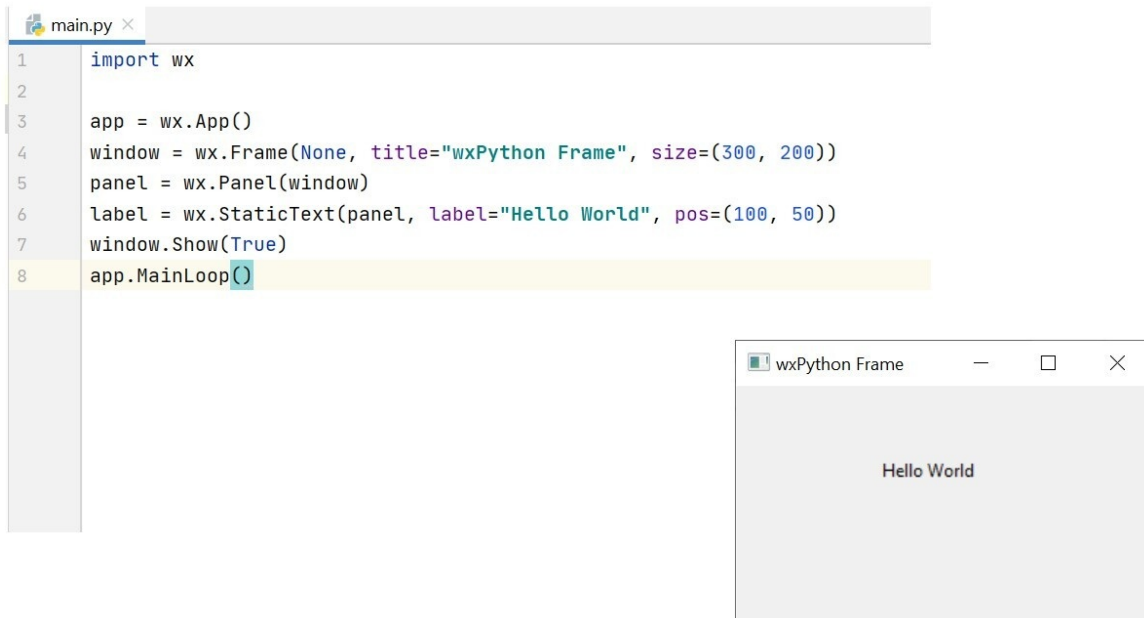



Для начала работы с библиотекой WxPython, создадим проект в PyCharm

и в окне терминала наберем команду pip install -U wxPython для установки библиотеки.

```
WXProject
Project
WXProject D:\user\PycharmProjects\WXProject
External Libraries
Scratches and Consoles
Terminal: Local +
Microsoft Windows [Version 10.0.18362.387]
(c) Корпорация Майкрософт (Microsoft Corporation), 2019. Все права защищены.

(venv) D:\user\PycharmProjects\WXProject>pip install -U wxPython
Collecting wxPython
  Downloading wxPython-4.1.1-cp39-cp39-win_amd64.whl (18.1 MB)
     |████████████████████████████████| 18.1 MB 172 kB/s
Collecting pillow
  Downloading Pillow-8.1.2-cp39-cp39-win_amd64.whl (2.2 MB)
     |████████████████████████████████| 2.2 MB 437 kB/s
Collecting six
  Using cached six-1.15.0-py2.py3-none-any.whl (10 kB)
Collecting numpy
  Downloading numpy-1.20.1-cp39-cp39-win_amd64.whl (13.7 MB)
     |████████████████████████████████| 13.7 MB 247 kB/s
Installing collected packages: six, pillow, numpy, wxPython
Successfully installed numpy-1.20.1 pillow-8.1.2 six-1.15.0 wxPython-4.1.1
6: TODO   Python Console   Terminal
```

Далее создадим файл питона и наберем простой код.




Здесь мы импортируем модуль wx.

Далее мы определяем объект класса Application.

И создаем окно верхнего уровня как объект класса Frame, заголовок и размер которого задаются в конструкторе.

Frame – это наиболее часто используемое окно верхнего уровня, размер и положение которого может быть изменено пользователем.

Оно имеет строку заголовка и кнопки управления.

И при необходимости можно добавить другие компоненты, такие как строка меню, панель инструментов и строка состояния.

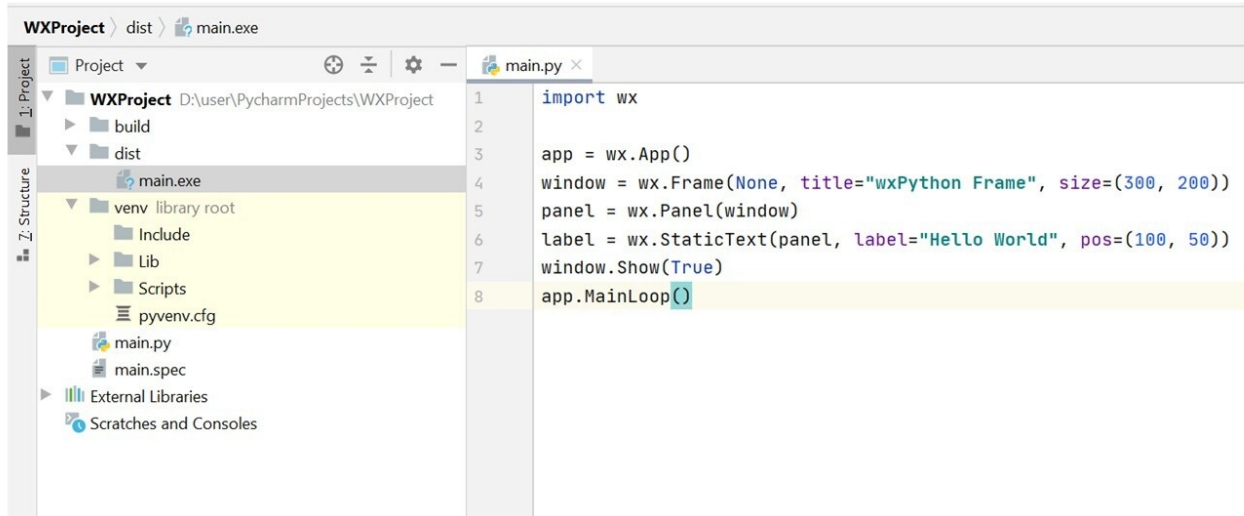
Далее мы помещаем объект Panel в Frame.

И добавляем объект StaticText для отображения «Hello World» внутри окна.

И наконец мы активируем окно фрейма методом show и входим в основной цикл событий объекта Application.

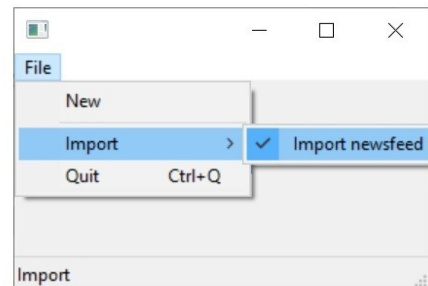
pip install pyinstaller

pyinstaller --onefile main.py




Для создания исполняемого файла приложения, можно воспользоваться инструментом PyInstaller.

Установив который с помощью команды pip, можно набрать команду pyinstaller в терминале и получить исполняемый файл.




```
class Example(wx.Frame):
    def __init__(self, *args, **kwargs):
        super(Example, self).__init__(*args, **kwargs)
        self.InitUI()
    def InitUI(self):
        self.SetSize(300,200)
        panel = wx.Panel(self)
        label = wx.StaticText(panel, label="Hello World", pos=(100, 50))

        menubar = wx.MenuBar()
        fileMenu = wx.Menu()
        fileMenu.Append(wx.ID_NEW, '&New’)
        fileMenu.AppendSeparator()
        imp = wx.Menu()
        sub=imp.Append(wx.ID_ANY, 'Import newsfeed list...', kind=wx.ITEM_CHECK)
        fileMenu.AppendMenu(wx.ID_ANY, 'I&mport', imp)
        imp.Check(sub.GetId(), True)
        fileItem = fileMenu.Append(wx.ID_EXIT, '&Quit\tCtrl+Q', 'Quit application')
        menubar.Append(fileMenu, '&File’)

        self.SetMenuBar(menubar)
        self.Bind(wx.EVT_MENU, self.OnQuit, fileItem)
    def OnQuit(self, e):
        self.Close()
```

Фрейм – это окно, размер и положение которого может быть изменено пользователем.

Это окно имеет границы и строку заголовка и может дополнительно содержать строку меню, панель инструментов и строку состояния.

И фрейм может содержать любое другое окно, кроме фрейма или диалога.

Строку состояния, панель инструментов и меню можно добавить с помощью функций setStatusBar, SetToolBar и SetMenuBar.

В меню есть пункты меню.

Пункты меню – это команды, которые выполняют определенное действие внутри приложения.

И в меню также могут быть подменю, в которых есть свои собственные пункты меню.

Для создания меню используются классы MenuBar, Menu и MenuItem.

Здесь мы добавляем пункт меню в объект меню методом Append.

Первый параметр здесь – это идентификатор пункта меню.

Второй параметр – это название пункта меню.

Третий параметр определяет строку, которая отображается в строке состояния при выборе пункта меню.

Метод Append возвращает созданный пункт меню.

Эта ссылка используется для привязки события.

Здесь мы привязываем пункт меню к методу OnQuit. Этот метод закрывает приложение.

В каждом меню также может быть подменю.

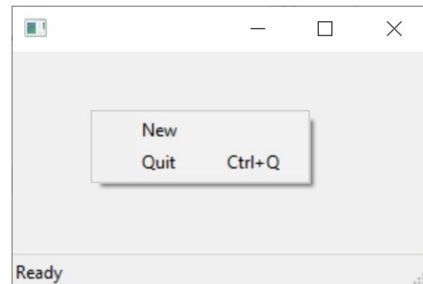
Таким образом, мы можем объединять похожие команды в группы.

В меню мы можем разделять команды разделителем с помощью метода AppendSeparator. Это простая линия.

Также есть три вида пунктов меню. Это обычное меню, это флажок и это радиоэлемент.

Если мы хотим добавить флажок, мы устанавливаем параметр kind как ITEM_CHECK.

```
def InitUI(self):
    self.SetSize(300,200)
    self.panel = wx.Panel(self)
    label = wx.StaticText(self.panel, label="Hello World", pos=(100, 50))

    self.fileMenu = wx.Menu()
    self.fileMenu.Append(wx.ID_NEW, '&New')
    fileItem = self.fileMenu.Append(wx.ID_EXIT, '&Quit\tCtrl+Q', 'Quit application')
    self.Bind(wx.EVT_MENU, self.OnQuit, fileItem)

    self.panel.Bind(wx.EVT_RIGHT_DOWN, self.OnRightDown)

    self.statusbar = self.CreateStatusBar()
    self.statusbar.SetStatusText('Ready')

  def OnRightDown(self, e):
    self.PopupMenu(self.fileMenu)
    print('click')

  def OnQuit(self, e):
    self.Close()
```




Контекстное меню – это список команд, который появляется в некотором контексте.

Например, когда мы щелкаем правой кнопкой мыши на элементе окна, мы видим контекстное меню.

И контекстные меню иногда называют всплывающими меню.

В этом примере мы создаем контекстное меню для главного окна.

В нем два элемента. Пункт меню создается и добавляется к контекстному меню.

И к этому пункту меню может быть привязан обработчик событий.

Если мы щелкнем правой кнопкой мыши на фрейме, мы вызываем метод OnRightDown.

Для этой привязки мы используем событие wx.EVT_RIGHT_DOWN.

В методе OnRightDown мы вызываем метод PopupMenu.

Этот метод показывает контекстное меню.

И контекстные меню появляются в точке наведения курсора мыши.

```
def InitUI(self):
    self.SetSize(300,200)
    self.panel = wx.Panel(self)
    self.label = wx.StaticText(self.panel, label="Hello World", pos=(100, 50))
    self.Centre()

    toolbar = self.CreateToolBar()
    qtool = toolbar.AddTool(101, 'Quit', wx.Bitmap("q.png") )
    toolbar.AddTool(102, 'New', wx.Bitmap("n.png"))
    toolbar.AddRadioTool(222, 'Right',wx.Bitmap("r.png"))
    toolbar.AddRadioTool(333, 'Center', wx.Bitmap("c.png"))
    toolbar.AddRadioTool(444, 'Justify',wx.Bitmap("j.png"))
    toolbar.Realize()

    self.Bind(wx.EVT_TOOL, self.OnTool, qtool)

  def OnTool(self, e):
    self.label.SetLabel(str(e.GetId()))
```

В меню, которое мы уже рассмотрели, группируются все команды, которые мы можем использовать в приложении.

Панели инструментов обеспечивают быстрый доступ к наиболее часто используемым командам.

Чтобы создать панель инструментов, мы вызываем метод CreateToolBar виджета фрейма.

По умолчанию панель инструментов горизонтальна, не имеет границ и отображает значки.

Чтобы создать элемент панели инструментов, мы вызываем метод AddTool.

Первый параметр, это идентификатор элемента.

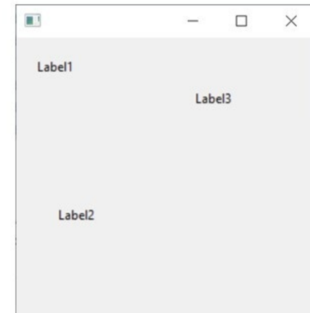
Второй параметр – это метка элемента, третий – изображение элемента.

И обратите внимание, что метка не отображается, потому что стиль по умолчанию показывает только значки.

Также мы можем добавить группу RadioTool, из которой можно выбрать только один элемент.

После того, как мы разместили наши элементы на панели инструментов, мы вызываем метод Realize.

Кнопки инструментов генерируют событие EVT_TOOL, с помощью которого мы можем привязать обработчик элемента панели инструментов.




```
def InitUI(self):
    self.SetSize(300,300)
    self.panel = wx.Panel(self)
    label1 = wx.StaticText(self.panel, label="Label1")
    label2 = wx.StaticText(self.panel, label="Label2")
    label3 = wx.StaticText(self.panel, label="Label3")
    self.Centre()

    label1.SetPosition((20, 20))
    label2.SetPosition((40, 160))
    label3.SetPosition((170, 50))
```

Теперь, давайте поговорим о компоновках в wxPython, с помощью которых виджеты размещаются внутри виджетов-контейнеров.

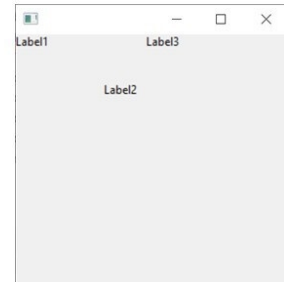
В wxPython можно размещать виджеты, используя абсолютное позиционирование или используя сайзеры.

При абсолютном позиционировании мы указываем положение и размер каждого виджета в пикселях.

И абсолютное позиционирование имеет ряд недостатков, а именно – размер и положение виджета не меняются, если мы изменяем размер окна, также приложения выглядят по-разному на разных платформах и изменение шрифтов в приложении может испортить компоновку.

Поэтому чаще всего в реальных программах используются сайзеры.

Здесь в этом примере мы размещаем три метки методом SetPosition используя абсолютное позиционирование.




```
def InitUI(self):
    self.SetSize(300,300)
    panel = wx.Panel(self)
    label1 = wx.StaticText(panel, label="Label1")
    label2 = wx.StaticText(panel, label="Label2")
    label3 = wx.StaticText(panel, label="Label3")
    self.Centre()
    hbox = wx.BoxSizer(wx.HORIZONTAL)
    hbox.Add(label1, proportion=0, flag=wx.RIGHT, border=8)
    hbox.Add(label2, proportion=1, flag=wx.LEFT|wx.TOP, border=50)
    hbox.Add(label3, proportion=2, flag=wx.LEFT|wx.RIGHT|wx.EXPAND, border=10)
    panel.SetSizer(hbox)
```

Сайзеры решают все проблемы абсолютного позиционирования, и wxPython имеет сайзеры

BoxSizer

,

StaticBoxSizer

,

GridSizer

,

FlexGridSizer

,

GridBagSizer

, и

BoxSizer

.

Компоновка BoxSizer позволяет размещать несколько виджетов в строке или столбце.

И мы можем добавлять один сайзер в другой. Таким образом мы можем создавать очень сложные макеты.

Ориентация компоновки может быть вертикальная или горизонтальная.

И добавление виджетов в BoxSizer осуществляется с помощью метода Add.

Параметр пропорции определяет соотношение изменения виджетов в заданной ориентации.

Предположим, у нас есть три элемента с пропорциями 0, 1 и 2.

И они добавлены в горизонтальный BoxSizer.

Элемент с пропорцией 0 вообще не изменится.

Элемент с пропорцией 2 изменится в два раза больше, чем элемент с пропорцией 1 по горизонтали.

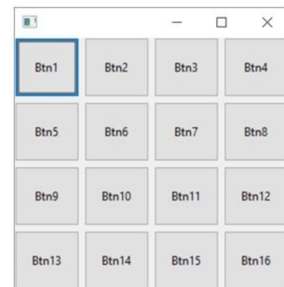
С помощью параметра flag вы можете дополнительно настроить поведение виджетов в BoxSizer.

Здесь мы можем контролировать границу border между виджетами, добавляя промежуток между виджетами в пикселях.

Чтобы применить границу, нам нужно определить стороны, где будет использоваться граница.

И мы можем комбинировать стороны, например LEFT | BOTTOM.

И наконец сайзер задается для контейнера с помощью метода setSizer.




```
def InitUI(self):
    self.SetSize(300,300)
    panel = wx.Panel(self)

    self.Centre()
    gs = wx.GridSizer(4, 4, 5, 5)

    for i in range(1, 17):
        btn = "Btn" + str(i)
        gs.Add(wx.Button(panel, label=btn), flag=wx.ALL|wx.EXPAND, border=10)

    panel.SetSizer(gs)
```

**Add (self, window, proportion=0, flag=0, border=0, userData=None)**

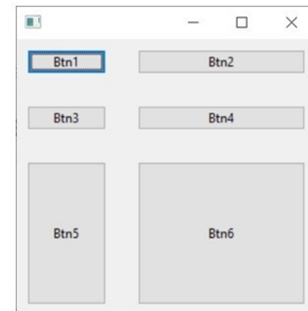
Компоновка GridSizer размещает виджеты в двухмерной таблице.

И каждая ячейка в этой таблице имеет одинаковый размер.

Здесь в конструкторе мы указываем количество строк и столбцов в таблице, а также вертикальное и горизонтальное расстояние между ячейками.

Здесь мы использовали метод Add. И виджеты размещаются внутри таблицы в том порядке, в котором они добавляются. Сначала заполняется первая строка, затем вторая строка и т. д.

В методе Add мы используем параметры flag и border, которые определяют ширину границы ячейки.




```
def InitUI(self):
    self.SetSize(300,300)
    panel = wx.Panel(self)

    self.Centre()
    fgs = wx.FlexGridSizer(3, 2, 10,10)

    for i in range(1, 7):
      btn = "Btn" + str(i)
      fgs.Add(wx.Button(panel, label=btn), flag=wx.ALL|wx.EXPAND, border=10)

    fgs.AddGrowableCol(1, 1)
    fgs.AddGrowableRow(2, 1)
    panel.SetSizer(fgs)
```

Компоновка
FlexGridSizer
похожа на
GridSizer
.

Она также размещает свои виджеты в двухмерной таблице.

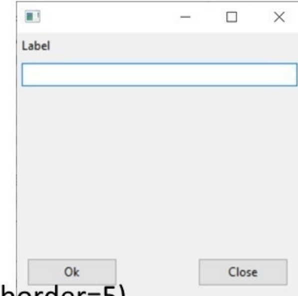
Но в то время как ячейки GridSizer имеют одинаковый размер, в FlexGridSizer строки и столбцы не обязательно имеют одинаковую высоту или ширину.

Поэтому эта компоновка обеспечивает немного больше гибкости при размещении элементов в ячейках.

Здесь размер каждой ячейки не одинаков, как в GridSizer.

И ширина и высота ячеек в одном столбце или строке может быть увеличена с помощью методов AddGrowableRow и AddGrowableCol.

Первый параметр здесь индекс строки или индекс столбца, а второй параметр – это доля прироста.




```
def InitUI(self):
    self.SetSize(300,300)
    panel = wx.Panel(self)

    sizer = wx.GridBagSizer(4, 4)
    text = wx.StaticText(panel, label="Label")
    sizer.Add(text, pos=(0, 0), flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)
    tc = wx.TextCtrl(panel)
    sizer.Add(tc, pos=(1, 0), span=(1, 5), flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
    buttonOk = wx.Button(panel, label="Ok", size=(90, 28))
    buttonClose = wx.Button(panel, label="Close", size=(90, 28))
    sizer.Add(buttonOk, pos=(3, 0),flag=wx.LEFT | wx.BOTTOM,border=10)
    sizer.Add(buttonClose, pos=(3, 3), flag=wx.RIGHT | wx.BOTTOM, border=10)
    sizer.AddGrowableCol(1)
    sizer.AddGrowableRow(2)
    panel.SetSizer(sizer)
```

Компоновка GridBagSizer – это самая гибкая компоновка в wxPython.

Эта компоновка позволяет явно позиционировать элементы.

Элементы также могут занимать более одной строки или столбца.

И мы добавляем элементы с помощью метода Add.

Здесь параметр pos указывает положение в сетке.

В верхней левой ячейке pos (0, 0).

Параметр span – это диапазон элемента, например span (3, 2) – это виджет занимает 3 строки и 2 столбца.

Параметры флаг и граница обсуждались в BoxSizer.

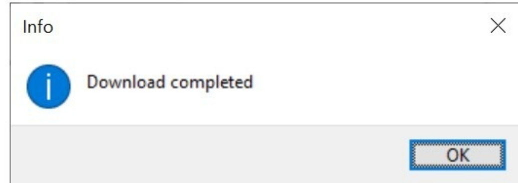
И ширина и высота ячеек в одном столбце или строке может быть увеличена с помощью методов AddGrowableRow и AddGrowableCol.

Таким образом, GridBagSizer – это универсальная компоновка.

Здесь дочерний виджет можно добавить в определенную ячейку сетки.

Кроме того, дочерний виджет может занимать более одной ячейки по горизонтали или вертикали.

```
def InitUI(self):
    self.SetSize(300,300)
    panel = wx.Panel(self)

    btn = wx.Button(panel, label='Dialog')
    btn.Bind(wx.EVT_BUTTON, self.ShowMessage)

def ShowMessage(self, event):
    dial = wx.MessageDialog(None, 'Download completed', 'Info', wx.OK | wx.ICON_INFORMATION)
    dial.ShowModal()
```




| | |
|---|---|
| wx.OK | show OK button |
| wx.CANCEL | show Cancel button |
| wx.YES_NO | show Yes, No buttons |
| wx.YES_DEFAULT | make Yes button the default |
| wx.NO_DEFAULT | make No button the default |
| wx.ICON_EXCLAMATION | show an alert icon |
| wx.ICON_ERROR | show an error icon |
| wx.ICON_HAND | same as wx.ICON_ERROR |
| wx.ICON_INFORMATION | show an info icon |
| wx.ICON_QUESTION | show a question icon |

Диалоговые окна являются неотъемлемой частью большинства приложений с графическим интерфейсом.

Диалог используется для ввода данных, изменения данных, изменения настроек приложения и т. д.

И мы можем использовать предопределенные диалоги, такие как окна сообщений, диалоги шрифтов или цветов, или создавать свои собственные диалоги.

Окно сообщения предоставляет пользователю краткую информацию.

И класс MessageBox показывает небольшое диалоговое окно.

Здесь мы предоставляем три параметра: текстовое сообщение, заголовок сообщения и флаги. Флаги используются для отображения различных кнопок и значков.

В нашем случае мы показываем кнопку OK и значок информации.

Создать диалоговое окно сообщения просто.

Мы устанавливаем диалог как окно верхнего уровня, указав None в качестве родителя.

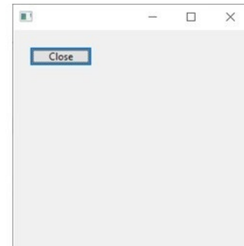
Чтобы отобразить диалог на экране, мы вызываем метод ShowModal.

Чтобы создать диалоговое окно about, мы должны создать два объекта – wx.adv.AboutDialogInfo и wx.adv.AboutBox.

Для создания собственного диалога, мы должны создать класс, который наследует от виджета wx.Dialog.

Мы создаем экземпляр этого класса, а затем мы вызываем метод ShowModal.

Позже мы должны уничтожить наш диалог с помощью метода Destroy.




```
def InitUI(self):
    self.SetSize(300,300)
    panel = wx.Panel(self)

    btn = wx.Button(panel, label='Close', pos=(20, 20))
    btn.Bind(wx.EVT_BUTTON, self.OnClose)

  def OnClose(self, e):
    self.Close(True)
```

Теперь давайте пройдемся по виджетам библиотеки wxPython.

Button – это простой виджет, который содержит текстовую строку и используется для запуска действия.

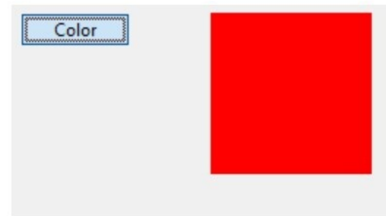
В этом примере мы создаем кнопку «Закрыть», которая при нажатии завершает работу приложения.

В конструкторе виджета мы указываем метку для кнопки и позицию на панели.

Методом Bind мы связываем событие с методом, и событие запускается, когда мы нажимаем на кнопку.

Здесь мы указываем обработчик для этого события, в котором мы завершаем приложение с помощью метода Close.

```
        tb = wx.ToggleButton(panel, label='Color', pos=(20, 50))
        tb.Bind(wx.EVT_TOGGLEBUTTON, self.Toggle)

        self.cpnl = wx.Panel(panel, pos=(150, 50), size=(110, 110))
        self.cpnl.SetBackgroundColour(wx.Colour(255,255,255))

    def Toggle(self, e):
        obj = e.GetEventObject()
        isPressed = obj.GetValue()
        if isPressed:
            self.cpnl.SetBackgroundColour(wx.RED)
        else:
            self.cpnl.SetBackgroundColour(wx.BLUE)
        self.cpnl.Refresh()
```




Виджет ToggleButton – это кнопка, которая имеет два состояния: нажата и не нажата.

И вы переключаетесь между этими двумя состояниями, нажимая на кнопку.

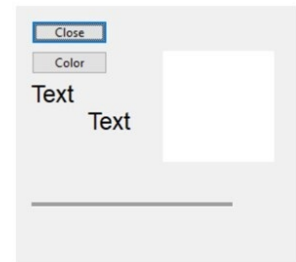
В этом примере мы меняем цвет панели, нажимая на кнопку.

Здесь создается виджет ToggleButton и панель, цвет которой мы будем изменять с помощью кнопки-переключателя.

Обработчик события Toggle вызывается, когда мы нажимаем кнопку.

В этом методе мы устанавливаем цвет фона панели в зависимости от того нажата кнопка или нет.

```
txt='''Text
      Text
      '''
str = wx.StaticText(panel, label=txt, pos=(20, 80))
font = wx.Font(18, wx.DEFAULT, wx.NORMAL, wx.DEFAULT)
str.SetFont(font)
```




```
wx.StaticLine(panel, pos=(20, 200), size=(200, 5))
```

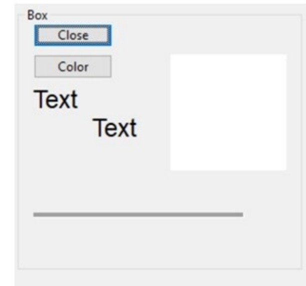
Виджет StaticText отображает одну или несколько строк текста, доступного только для чтения.

Здесь мы создаем шрифт для текста.

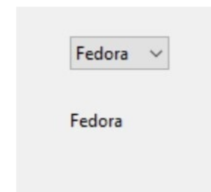
И устанавливаем шрифт с помощью метода SetFont.

Виджет StaticLine отображает в окне простую строку, которая может использоваться как разделительная строка.

```
wx.StaticBox(panel, label='Box', pos=(5, 5), size=(270, 250))
```




```
os = ['Ubuntu', 'Arch', 'Fedora', 'Debian', 'Mint']
        cb = wx.ComboBox(panel, pos=(50, 250), choices=os, style=wx.CB_READONLY)
        cb.Select(0)
        self.st = wx.StaticText(panel, label='', pos=(50, 300))
        cb.Bind(wx.EVT_COMBOBOX, self.OnSelect)

def OnSelect(self, e):
        i = e.GetString()
        self.st.SetLabel(i)
```




Виджет StaticBox используется для логической группировки различных виджетов.

И эти виджеты должны быть родственными, а не дочерними по отношению к статическому блоку.

Здесь включаемые виджеты регулируются размерами рамки статического блока.

Какая будет рамка, столько виджетов и поместится в блок.

Виджет ComboBox – это выпадающий список с кнопкой.

Когда вы нажимаете кнопку, появляется список.

И пользователь может выбрать только один вариант из предоставленного списка строк.

При создании виджета ComboBox, параметр choices принимает список строк, отображаемых в поле со списком.

Стиль READONLY делает строки списка доступными только для чтения.

И когда мы выбираем параметр из поля со списком, запускается событие COMBOBOX.

К этому событию мы подключаем обработчик события OnSelect.

```
    self.cb = wx.CheckBox(panel, pos=(50, 350), label='Show')
    self.cb.SetValue(True)
    self.cb.Bind(wx.EVT_CHECKBOX, self.ShowOrHide)

def ShowOrHide(self, e):
    i = e.GetEventObject()
    isChecked = i.GetValue()
    if isChecked:
        self.cb.SetLabel('Show')
    else:
        self.cb.SetLabel('Hide')
```

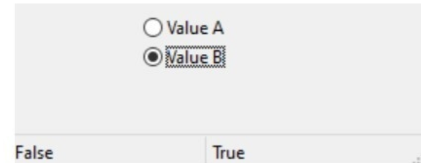
CheckBox – это виджет, который имеет два состояния: включен и выключен.

Здесь мы проверяем состояние виджета CheckBox с помощью метода GetValue.

И событие CHECKBOX запускается, когда мы щелкаем по виджету CheckBox.

При запуске этого события вызывается обработчик ShowOrHide.

```
    self.rb1 = wx.RadioButton(panel, label='Value A', pos=(200, 250), style=wx.RB_GROUP)
    self.rb2 = wx.RadioButton(panel, label='Value B', pos=(200, 270))
    self.rb1.Bind(wx.EVT_RADIOBUTTON, self.SetVal)
    self.rb2.Bind(wx.EVT_RADIOBUTTON, self.SetVal)
    self.sb = self.CreateStatusBar(2)
    self.sb.SetStatusText("True", 0)
    self.sb.SetStatusText("False", 1)

def SetVal(self, e):
    state1 = str(self.rb1.GetValue())
    state2 = str(self.rb2.GetValue())
    self.sb.SetStatusText(state1, 0)
    self.sb.SetStatusText(state2, 1)
```




RadioButton – это виджет, который позволяет пользователю выбрать один вариант из группы параметров.

Группа переключателей определяется тем, что первый переключатель в группе содержит стиль GROUP.

Все остальные переключатели, определенные после первого переключателя с этим флагом стиля, будут добавлены в функциональную группу первого переключателя.

Объявление другой радиокнопки с флагом GROUP запустит новую группу радиокнопок.

Здесь у нас есть группа из двух переключателей.

И состояние каждого из переключателей отображается в строке состояния.

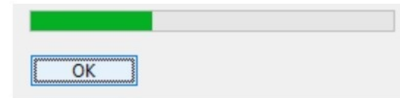
И мы привязываем событие RADIOBUTTON к обработчику события SetVal.

Также мы создаем статусбар с двумя полями.

В методе SetVal мы узнаем состояния переключателей.

И мы обновляем поля строки состояния до текущих значений переключателей.

```
    self.gauge = wx.Gauge(panel, range=50, size=(250, -1),pos=(20, 370))

    self.btnS = wx.Button(panel, wx.ID_OK, pos=(20, 400))
    self.Bind(wx.EVT_BUTTON, self.OnStart, self.btnS)
    self.timer = wx.Timer(self, 1)
    self.count = 0
    self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)

def OnStart(self, e):
    if self.count >= 50:
        return
    self.timer.Start(100)
    self.sb.SetStatusText("Task in Progress", 0)

def OnTimer(self, e):
    self.count = self.count + 1
    self.gauge.SetValue(self.count)
    if self.count == 50:
        self.timer.Stop()
        self.sb.SetStatusText("Task Completed", 0)
```




Датчик Gauge – это виджет, который используется при выполнении длительных задач, где требуется индикатор, показывающий текущее состояние задачи.

В этом примере у нас есть датчик и кнопка, которая запускает датчик.

И мы используем Timer для выполнения кода через определенные промежутки времени.

В эти моменты мы обновляем шкалу датчика.

Переменная count используется для определения того, какая часть задачи уже выполнена.

В конструкторе виджета Gauge, параметр диапазона устанавливает максимальное целочисленное значение виджета.

Когда мы нажимаем кнопку OK, вызывается метод OnStart.

Здесь сначала мы проверяем, находится ли переменная count в диапазоне задачи.

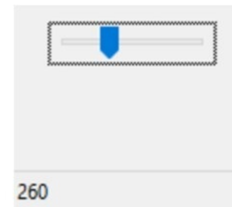
Если нет, то возвращаемся из метода.

Если же задача еще не выполнена, мы запускаем таймер.

И метод OnTimer вызывается периодически после запуска таймера.

В этом методе мы обновляем переменную count и виджет датчика.

Если переменная count равна максимальному значению, мы останавливаем таймер.




```
    sld = wx.Slider(panel, value=200, minValue=150, maxValue=500,
pos=(20, 450),style=wx.SL_HORIZONTAL)

    sld.Bind(wx.EVT_SCROLL, self.OnSliderScroll)

  def OnSliderScroll(self, e):
    obj = e.GetEventObject()
    val = obj.GetValue()
    self.sb.SetStatusText(str(val), 0)
```

Slider – это виджет с ползунком, который можно тянуть вперед и назад.

Таким образом, мы можем выбрать конкретное значение из диапазона.

Здесь, значение, выбранное в ползунке, отображается в строке статуса.

При создании Slider, мы указываем начальную позицию ползунка параметром value, а минимальное и максимальное положение ползунка – параметрами minValue и maxValue.

Стиль HORIZONTAL делает ползунок горизонтальным.

Когда появляется событие SCROLL, здесь вызывается метод OnSliderScroll, в котором текущее выбранное значение ползунка отображается в строке статуса.

```
self.sc = wx.SpinCtrl(panel, value='0',pos=(20, 500))
self.sc.SetRange(-459, 1000)
self.sc.Bind(wx.EVT_SPINCTRL, self.OnCompute)

def OnCompute(self, e):
    fahr = self.sc.GetValue()
    val = round((fahr - 32) * 5 / 9.0, 2)
    self.sb.SetStatusText(str(val), 0)
```

Виджет SpinCtrl позволяет увеличивать и уменьшать значение в определенном диапазоне.

Здесь мы переводим температуру по Фаренгейту в градусы Цельсия.

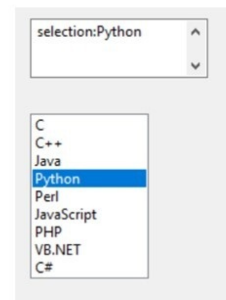
Мы создаем виджет SpinCtrl с начальным значением 0.

И метод SetRange устанавливает диапазон значений для виджета.

Когда мы выбираем значение в виджете, вызывается метод OnCompute.

В этом методе мы получаем текущее значение и вычисляем температуру по Цельсию и показываем вычисленную температуру в строке статуса.

```
languages = ['C', 'C++', 'Java', 'Python', 'Perl', 'JavaScript', 'PHP', 'VB.NET', 'C#']
self.text = wx.TextCtrl(panel, style=wx.TE_MULTILINE, pos=(300, 20), size=(150, 50))

lst = wx.ListBox(panel, size=(100, -1), choices=languages, style=wx.LB_SINGLE, pos=(300, 100))
self.Bind(wx.EVT_LISTBOX, self.onListBox, lst)

def onListBox(self, event):
    self.text.AppendText("Current selection:"+event.GetEventObject().GetStringSelection()+"\n")
```




Виджет ListBox представляет собой список строк с вертикальной прокруткой.

И по умолчанию можно выбрать один элемент в списке. Однако список можно настроить и для множественного выбора.

Здесь параметр Choices – это список строк, используемых для заполнения списка.

В этом примере ListBox заполняется строками с помощью объекта languages.

И ListBox связывается с обработчиком onListBox с помощью события LISTBOX.

Этот обработчик добавляет выбранную строку в многострочный виджет TextCtrl.

```
maxint=2**31-1
players = [('Tendulkar', '15000', '100'), ('Dravid', '14000', '1'),
          ('Kumble', '1000', '700'), ('KapilDev', '5000', '400'),
          ('Ganguly', '8000', '50')]
self.list = wx.ListCtrl(panel, -1, style=wx.LC_REPORT,pos=(300, 250))
self.list.InsertColumn(0, 'name', width=100)
self.list.InsertColumn(1, 'runs', wx.LIST_FORMAT_RIGHT, 100)
self.list.InsertColumn(2, 'wkts', wx.LIST_FORMAT_RIGHT, 100)

for i in players:
   index = self.list.InsertStringItem(maxint, i[0])
   self.list.SetStringItem(index, 1, i[1])
   self.list.SetStringItem(index, 2, i[2])

self.list.Bind(wx.EVT_LIST_ITEM_SELECTED, self.OnClick, self.list)

def OnClick(self, event):
   ind = event.GetIndex()
   item = self.list.GetItem(ind, 1)
   self.sb.SetStatusText(item.GetText(), 0)
```

| name | runs | wkts |
|---|---|---|
| Tendulkar | 15000 | 100 |
| Dravid | 14000 | 1 |
| Kumble | 1000 | 700 |
| KapilDev | 5000 | 400 |
| Ganguly | 8000 | 50 |

Теперь, как создать не список, а таблицу?

ListCtrl – это улучшенный виджет списка.

Если ListBox показывает только один столбец, ListCtrl может содержать несколько столбцов.

Внешний вид виджета ListCtrl контролируется параметрами стиля.

Столбцы заголовка создаются методом InsertColumn, который принимает параметры номера столбца, заголовок, стиль и ширину.

Здесь список кортежей, каждый из которых содержит три строки, хранит данные, которые используются для заполнения столбцов объекта ListCtrl.

Новая строка заполняется методом InsertStringItem, который возвращает индекс текущей строки.

Использование maxint дает номер строки после последней строки.

Используя индекс, другие столбцы заполняются методом SetStringItem.

Событие LIST_ITEM_SELECTED передает индекс выбранного элемента из таблицы.

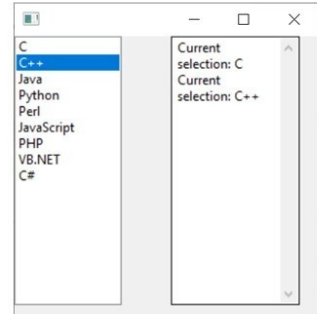
И мы используем этот индекс в обработчике, чтобы получить элемент и столбец в таблице, чтобы получить текст ячейки.

```
#pip install comtypes

html = wx.lib.iewin.IEHtmlWindow(panel, pos=(300, 420), size=(300,100))

html.Navigate('https://www.wxpython.org')
```

Библиотека wxHTML и iewin содержат классы для анализа и отображения содержимого HTML.

Хотя этот браузер не предназначен для использования в качестве полнофункционального браузера, объект IEHtmlWindow может служить универсальным средством просмотра HTML.

Хотя для использования этого браузера понадобится установка дополнительного модуля comtypes, питон COM-пакета.




```
    splitter = wx.SplitterWindow(self, -1)
    panel = wx.Panel(splitter,size=(100, 300))
    self.text = wx.TextCtrl(panel, size=(120, 250), style=wx.TE_MULTILINE)
    panelS = wx.Panel(splitter, size=(100, 300))
    languages = ['C', 'C++', 'Java', 'Python', 'Perl',
               'JavaScript', 'PHP', 'VB.NET', 'C#']
    lst = wx.ListBox(panelS, size=(100, 250), choices=languages, style=wx.LB_SINGLE)
    splitter.SplitVertically(panelS, panel)
    self.Bind(wx.EVT_LISTBOX, self.onListBox, lst)

  def onListBox(self, event):
    self.text.AppendText("Current selection: " + event.GetEventObject().GetStringSelection() + "\n")
```

SplitterWindow – это специальная компоновка, которая содержит два подокна, размер которых можно динамически изменять, перетаскивая границы между ними.

Класс SplitterWindow имеет очень простой конструктор с параметрами, имеющими значения по умолчанию.

В этом примере SplitterWindow добавляется к фрейму верхнего уровня.

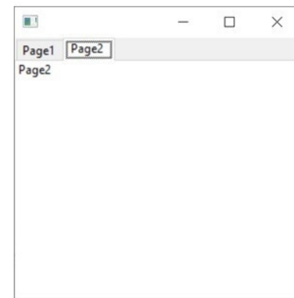
Одна панель предназначена для хранения многострочного поля TextCtrl.

Список ListBox помещается на другую панель.

И SplitterWindow разделяет по вертикали окно, и две панели добавляются в подокна.

Ширину подокон можно изменять с помощью перетаскивания границы.

```
nb = wx.Notebook(self)

panel1 = wx.Panel(nb)
panel2 = wx.Panel(nb)

nb.AddPage(panel1, "Page1")
nb.AddPage(panel2, "Page2")

txt1 = wx.StaticText(panel1, -1, style=wx.ALIGN_CENTER)
txt1.SetLabel("Page1")
txt2 = wx.StaticText(panel2, -1, style=wx.ALIGN_CENTER)
txt2.SetLabel("Page2")
```




Библиотека wxPython содержит набор виджетов Book, которые позволяют пользователю переключаться между различными панелями в окне.

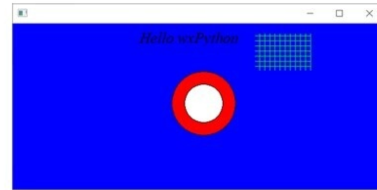
Это такие виджеты как Notebook, Choicebook, Listbook и Treebook.

Виджет Notebook представляет собой окно с вкладками или страницами.

И пользователь может переключаться между страницами, щелкая заголовок соответствующей вкладки.

Объекты этих вкладок добавляются как страницы в Notebook во фрейме верхнего уровня.

```
dc = wx.PaintDC(self)
brush = wx.Brush("blue")
dc.SetBackground(brush)
dc.Clear()

color = wx.Colour(255, 0, 0)
b = wx.Brush(color)
dc.SetBrush(b)
dc.DrawCircle(300, 125, 50)
dc.SetBrush(wx.Brush(wx.Colour(255, 255, 255)))
dc.DrawCircle(300, 125, 30)

font = wx.Font(18, wx.ROMAN, wx.ITALIC, wx.NORMAL)
dc.SetFont(font)
dc.DrawText("Hello wxPython", 200, 10)

pen = wx.Pen(wx.Colour(0, 0, 255))
dc.SetPen(pen)
dc.DrawLine(200, 50, 350, 50)

dc.SetBrush(wx.Brush(wx.Colour(0, 255, 0), wx.CROSS_HATCH))
dc.DrawRectangle(380, 15, 90, 60)
```




API рисования wxPython предлагает различные функции для рисования форм, текста и изображений.

Объекты, необходимые для рисования, такие как цвет, перо, кисть и шрифт, могут быть созданы с использованием классов интерфейса.

Класс PaintDC используется для рисования в клиентской области окна, с помощью события PaintEvent.

Существуют также такие классы как ScreenDC, который используется для рисования на экране, и ClientDC, который используется для рисования в клиентской области окна без события PaintEvent.

Класс Colour представляет собой комбинацию значений RGB и есть также предопределенные цветовые объекты, такие как BLACK, BLUE, GREEN и так далее.

Цвет с произвольной комбинацией значений RGB формируется как объект Colour.

Объект Pen определяет цвет, ширину и стиль форм, таких как линия, прямоугольник, круг и т. д.

Кисть Brush – это еще один графический объект, необходимый для заливки фона таких фигур, как прямоугольник, эллипс, круг и т. д.

Для настраиваемого объекта Brush требуются параметры стиля Colour и Brush.

Здесь пример показывает использование объектов Pen, Brush, Color и Font.

# Dear PyGUI

**Создание настольных Python приложений с GUI**

**Dear PyGUI**

Dear PyGui – это простой и мощный фреймворк графического интерфейса пользователя Python.

Dear PyGui – это библиотека, созданная с помощью библиотеки C++ Dear ImGui, для имитации традиционного графического интерфейса.

Dear PyGui отличается от других графических интерфейсов Python тем, что под капотом Dear PyGui использует графический процессор компьютера для отрисовки динамических интерфейсов.

Dear PyGui не использует нативные виджеты, а вместо этого использует графическую карту компьютера для непосредственной отрисовки.

**pip install dearpygui**

PyGUI

Project

PyGUI D:\user\PycharmProjects\PyGUI
External Libraries
Scratches and Consoles

Search Everywhere Double Shift
Go to File Ctrl+Shift+N
Recent Files Ctrl+E
Navigation Bar Alt+Home

Terminal: Local +

```
Microsoft Windows [Version 10.0.18362.387]
(с) Корпорация Майкрософт (Microsoft Corporation), 2019. Все права защищены.

(venv) D:\user\PycharmProjects\PyGUI>pip install dearpygui
Collecting dearpygui
  Downloading dearpygui-0.6.398-cp39-cp39-win_amd64.whl (1.1 MB)
     |████████████████████████████████| 1.1 MB 819 kB/s
Installing collected packages: dearpygui
Successfully installed dearpygui-0.6.398
```

6: TODO Terminal Python Console

Для установки библиотеки Dear PyGui создадим проект в PyCharm
и в окне терминала наберем команду pip install dearpygui.

Dear PyGui **https://hoffstadt.github.io/DearPyGui/index.html**

Index

Last Updated: 2021-04-19

Sub-modules

- dearpygui.core
- dearpygui.simple

Package **dearpygui**

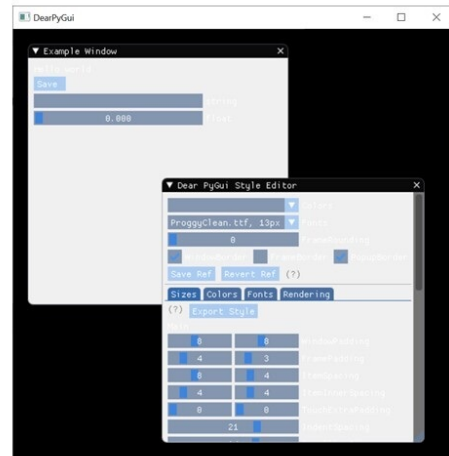
Sub-modules

dearpygui.core

dearpygui.simple

Существует онлайн документация библиотеки, где можно посмотреть все функции и параметры.

```
from dearpygui import core, simple
from dearpygui.core import *
from dearpygui.simple import *

def save_callback(sender, data):
   print("Save Clicked")

set_main_window_size(400, 400)

with simple.window("Example Window", width=350, height=350,
x_pos=20, y_pos=20):
   core.add_text("Hello world")
   core.add_button("Save", callback=save_callback)
   core.add_input_text("string")
   core.add_slider_float("float")

show_style_editor()
core.set_theme_item(mvGuiCol_WindowBg, 240, 240, 240, 255)
core.start_dearpygui()
```




Dear PyGui состоит из окна программы, окон и виджетов.

Окно программы – это главное окно программы, которое создается в конце основного скрипта Python с помощью вызова функции start_dearpygui.

Dear PyGui состоит из двух модулей core и simple.

Модуль core содержит базовую функциональность Dear PyGUI.

Модуль simple содержит простые оболочки и другие утилиты, созданные из core, чтобы обеспечить более удобный интерфейс Dear PyGui.

Методом set_main_window_size мы устанавливаем размер основного окна.

Это самое большое черное окно.

Метод window модуля simple является оберткой метода add_window модуля core.

Этот метод добавляет окно в основное окно.

И в это добавленное окно мы уже добавляем элементы методом add.

Здесь мы добавили текст, кнопку, поле ввода и слайдер.

Теперь, как изменять внешний вид окна.

Мы вызываем метод show_style_editor, который открывает окно редактора.

Здесь это окно ниже.

В этом окне мы меняем размеры и цвета и нажимаем кнопку Save, где она есть, и кнопку Export.

При этом в буфер копируется код изменения внешнего вида.

Мы сохраняем этот код в блокноте и можем выбрать там нужный нам метод и перенести его уже в наш код.

Здесь мы выбрали метод set_theme_item с параметром WindowBg, для изменения цвета фона окна, который стал светлым, а не темным как по умолчанию.

Далее мы уже убираем метод show_style_editor из нашего кода, чтобы не показывать редактор стилей.

Параметр callback позволяет связать обработчик с кнопкой, чтобы выполнять код при нажатии пользователем кнопки.

Dear PyGui

Functions

add_about_window
add_additional_font
add_annotation
add_area_series
add_bar_series
add_button
add_candle_series
add_character_remap
add_checkbox
add_child
add_collapsing_header
add_color_button
add_color_edit3
add_color_edit4
add_color_picker3
add_color_picker4

Module **dearpygui.core**

Functions

```
def add_about_window(...)
```

Creates an about window.

Return Type: None

Parameters

- name : str
- Keyword Only Arguments

В документации легко посмотреть все методы для добавления виджетов в окно.

```
add_text('Below You Will See two buttons')

add_button("Button 1")       add_button("Apply##1")        add_button("Apply2", label="Apply")

add_same_line(spacing=30) #Adding a space on the same line

add_checkbox("Checkbox")

add_menu_bar("Main Menu Bar")

add_menu("Format")

add_menu_item("form1")

add_simple_plot("Histogram", [1,4,2,8,12], height=180, histogram=True)

add_drawing("First_Drawing", width=300, height=300)

draw_circle("First_Drawing", [150, 150], radius=50, color=[255, 255, 255, 255], segments=0)
```

У каждого виджета должно быть уникальное имя.

По умолчанию имя присваивается как метка виджета, если это применимо.

Если вы хотите изменить метку виджета, вы можете поместить две решетки «##» в конце имени и все, что будет после решеток «##» будет скрыто от отображаемого имени.

Также вы можете использовать ключевое слово label, которое будет отображать метку вместо имени виджета.

Некоторые имена виджетов создаются автоматически, например для виджета same_line.

Однако у них есть необязательное ключевое слово name, которое можно указать, если вам понадобится ссылаться на виджет позже.

По умолчанию виджеты создаются в порядке их кодирования.

Однако элементы могут быть добавлены не по порядку, если указать родительский контейнер.

Использование ключевого слова parent вставит виджет в конец дочернего списка родителя.

Если вы хотите вставить его в другое место, используйте ключевое before в сочетании с ключевым словом parent, чтобы поместить элемент перед другим виджетом в дочернем списке.

Каждый виджет ввода имеет значение, которое можно установить с помощью ключевого слова default_value при создании, или во время

выполнения командой set_value.

Чтобы получить доступ к значению виджета, мы можем использовать команду get_value.

Каждый виджет ввода имеет обработчик, который запускается при взаимодействии с виджетом.

Обработчики могут быть назначены виджету при создании или после создания с помощью метода set_item_callback.

# Список использованных источников

https://riverbankcomputing.com/software/pyqt
https://pypi.org/project/PySide/
https://docs.python.org/3/library/tkinter.html
https://kivy.org/#home
https://www.wxpython.org
https://github.com/hoffstadt/DearPyGui
https://habr.com/ru/
https://coderlessons.com