Tkinter
Программирование GUI на Python

Курс

Опубликован: https://younglinux.info/tkinter/course

Автор: Светлана Шапошникова (plustilino)

Версия: май 2021 года

PDF-версия содержит текст уроков курса + решения с пояснениями к практическим работам

Курс "Tkinter. Программирование GUI на Python" знакомит с особенностями создания графического интерфейса пользователя средствами пакета `tkinter` языка программирования Python. Требует знания языка Python на уровне структурного программирования, желательно также владение азами ООП.

В курсе уделяется внимание работе с базовыми элементами интерфейса (виджетами). Изучаются основные свойства и методы кнопок, меток, текстовых полей, списков, флажков и радиокнопок, холста и меню.

Рассматриваются различные способы размещения виджетов в родительском окне: простой упаковкой, табличным разбиением и с помощью координат.

Изучаются настройки главного окна приложения, а также диалоговые окна.

Курс включает 16 уроков:	*Стр.*	*Стр. решения практической работы*
1 Что такое Tkinter	3	73
2 Виджеты Button, Label, Entry	9	77
3 Метод pack	13	80
4 Text – многострочное текстовое поле	18	82
5 Radiobutton и Checkbutton. Переменные Tkinter	23	84
6 Виджет Listbox	29	86
7 Метод bind	32	87
8 События	35	88
9 Canvas	40	90
10 Canvas. Идентификаторы, теги и анимация	44	91
11 Окна	48	93
12 Метод grid	52	95
13 Диалоговые окна	56	97
14 Виджет Menu	60	99
15 Метод place	64	101
16 Модуль tkinter.ttk	67	102

Урок 1. Что такое Tkinter

Tkinter – это пакет для Python, предназначенный для работы с библиотекой Tk. Библиотека Tk содержит компоненты графического интерфейса пользователя (graphical user interface – GUI). Эта библиотека написана на языке программирования Tcl.

Под графическим интерфейсом пользователя (GUI) подразумеваются все те окна, кнопки, текстовые поля для ввода, скроллеры, списки, радиокнопки, флажки и др., которые вы видите на экране, открывая то или иное приложение. Через них вы взаимодействуете с программой и управляете ею. Все эти элементы интерфейса будем называть виджетами (widgets).

В настоящее время почти все приложения, которые создаются для конечного пользователя, имеют GUI. Редкие программы, подразумевающие взаимодействие с человеком, остаются консольными. В предыдущих двух курсах мы писали только консольные программы.

Существует множество библиотек GUI, среди которых Tk не самый популярный инструмент, хотя с его помощью написано немало проектов. Он был выбран для Python по-умолчанию. Установочный файл интерпретатора Питона обычно уже включает пакет **`tkinter`** в составе стандартной библиотеки.

Tkinter можно представить как переводчик с языка Python на язык Tcl. Вы пишете программу на Python, а код модуля **`tkinter`** переводит ваши инструкции на язык Tcl, который понимает библиотека Tk.

Программы с графическим интерфейсом пользователя *событийно-ориентированные*. Вы уже должны иметь представление о структурном и желательно объектно-ориентированном программировании. Событийно-ориентированное ориентировано на события. То есть та или иная часть программного кода начинает выполняться лишь тогда, когда случается то или иное событие.

Событийно-ориентированное программирование базируется на объектно-ориентированном и структурном. Даже если мы не создаем собственных классов и объектов, то все-равно ими пользуемся. Все виджеты – объекты, порожденные встроенными классами.

События бывают разными. Сработал временной фактор, кто-то кликнул мышкой или нажал `Enter`, начал вводить текст, переключил радиокнопки, прокрутил страницу вниз и т. д. Когда случается что-то подобное, то, если был создан соответствующий обработчик, происходит срабатывание определенной части программы, что приводит к какому-либо результату.

Tkinter импортируется стандартно для модуля Python любым из способов:

- `import tkinter`
- `from tkinter import *`
- `import tkinter as tk`

Можно импортировать отдельные классы, что делается редко. В данном курсе мы будем использовать выражение `from tkinter import *`.

Если необходимо, узнать установленную версию Tk можно через константу **`TkVersion`**:

```
>>> from tkinter import *
>>> TkVersion
8.6
```

Чтобы написать GUI-программу, надо выполнить приблизительно следующее:

1. Создать главное окно.
2. Создать виджеты и выполнить конфигурацию их свойств (опций).
3. Определить события, то есть то, на что будет реагировать программа.
4. Описать обработчики событий, то есть то, как будет реагировать программа.
5. Расположить виджеты в главном окне.
6. Запустить цикл обработки событий.

Последовательность не обязательно такая, но первый и последний пункты всегда остаются на своих местах. Посмотрим все это в действии.

В современных операционных системах любое пользовательское приложение заключено в окно, которое можно назвать главным, так как в нем располагаются все остальные виджеты. Объект окна верхнего уровня создается от класса `Tk` модуля `tkinter`. Переменную, связываемую с объектом, часто называют *root* (корень):

```
root = Tk()
```

Пусть в окне приложения располагаются текстовое поле, метка и кнопка. Данные объекты создаются от классов `Entry`, `Label` и `Button` модуля `tkinter`. Мы сразу сконфигурируем некоторые их свойства с помощью передачи аргументов конструкторам этих классов:

```
ent = Entry(root, width=20)
but = Button(root, text="Преобразовать")
lab = Label(root, width=20, bg='black', fg='white')
```

Устанавливать свойства объектов не обязательно при их создании. Существуют еще пара способов, с помощью которых это можно сделать после.

Первым аргументом в конструктор виджета передается виджет-хозяин, то есть тот, на котором будет располагаться создаваемый. В случае, когда элементы GUI помещаются непосредственно на главное окно, родителя можно не указывать. То есть в нашем примере мы можем убрать *root*:

```
ent = Entry(width=20)
but = Button(text="Преобразовать")
lab = Label(width=20, bg='black', fg='white')
```

Однако виджеты не обязательно располагаются на *root*'е. Они могут находиться на других виджетах, и тогда указывать "мастера" необходимо.

Пусть в программе текст, введенный человеком в поле, при нажатии на кнопку разбивается на список слов, слова сортируются по алфавиту и выводятся в метке. Выполняющий все это код надо поместить в функцию:

```
def str_to_sort_list(event):
    s = ent.get()
    s = s.split()
```

```
    s.sort()
    lab['text'] = ' '.join(s)
```

У функций, которые вызываются при наступлении события с помощью метода **bind**, должен быть один параметр. Обычно его называют *event*, то есть "событие".

В нашей функции с помощью метода **get** из поля забирается текст, представляющий собой строку. Она преобразуется в список слов с помощью метода **split**. Потом список сортируется. В конце изменяется свойство **text** метки. Ему присваивается строка, полученная из списка с помощью строкового метода **join**.

Теперь необходимо связать вызов функции с событием:

```
but.bind('<Button-1>', str_to_sort_list)
```

В данном случае это делается с помощью метода **bind**. Ему передается событие и функция-обработчик. Событие будет передано в функцию и присвоено параметру *event*. Здесь событием является щелчок левой кнопкой мыши, что обозначается строкой **'<Button-1>'**.

В любом приложении виджеты не разбросаны по окну как попало, а организованы, интерфейс продуман до мелочей и обычно подчинен определенным стандартам. Пока расположим элементы друг под другом с помощью наиболее простого менеджера геометрии **tkinter** – метода **pack**:

```
ent.pack()
but.pack()
lab.pack()
```

Метод **mainloop** экземпляра **Tk** запускает главный цикл обработки событий, что в том числе приводит к отображению главного окна со всеми "упакованными" на нем виджетами:

```
root.mainloop()
```

Полный код программы:

```
from tkinter import *


def str_to_sort_list(event):
    s = ent.get()
    s = s.split()
    s.sort()
    lab['text'] = ' '.join(s)


root = Tk()

ent = Entry(width=20)
but = Button(text="Преобразовать")
lab = Label(width=20, bg='black', fg='white')

but.bind('<Button-1>', str_to_sort_list)

ent.pack()
but.pack()
lab.pack()
root.mainloop()
```

В результате выполнения данного скрипта появляется окно, в текстовое поле которого можно ввести список слов, нажать кнопку и получить его отсортированный вариант:

Попробуем теперь реализовать в нашей программе объектно-ориентированный подход. Это необязательно, но нередко бывает удобным. Пусть группа из метки, кнопки и поля представляет собой один объект, порождаемый от класса *Block*. Тогда в основной ветке программы будет главное окно, объект типа *Block* и запуск окна. Поскольку блок должен быть привязан к главному окну, то неплохо бы передать в конструктор класса окно-родитель:

```
from tkinter import *

root = Tk()

first_block = Block(root)

root.mainloop()
```

Теперь напишем сам класс Block:

```
class Block:
    def __init__(self, master):
        self.ent = Entry(master, width=20)
        self.but = Button(master,
                          text="Преобразовать")
        self.lab = Label(master, width=20,
                         bg='black', fg='white')
        self.but['command'] = self.str_to_sort
        self.ent.pack()
        self.but.pack()
        self.lab.pack()

    def str_to_sort(self):
        s = self.ent.get()
        s = s.split()
        s.sort()
        self.lab['text'] = ' '.join(s)
```

Здесь виджеты являются значениями полей объекта типа *Block*, функция-обработчик события нажатия на кнопку устанавливается не с помощью метода `bind`, а с помощью свойства кнопки `command`. В этом случае в вызываемой функции не требуется параметр *event*. В метод мы передаем только сам объект.

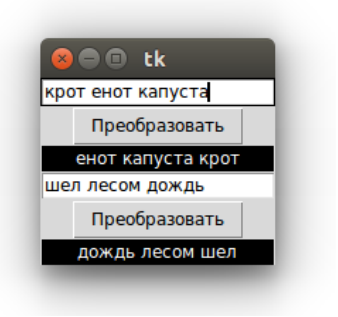
Однако, если код будет выглядеть так, то необходимости в классе нет. Смысл появится, если нам потребуется несколько или множество похожих объектов-блоков. Допустим, нам нужно несколько блоков, состоящих из метки, кнопки, поля. Причем у кнопки каждой группы будет своя функция-обработчик клика.

Тогда можно передавать значения для свойства command в конструктор. Значение будет представлять собой привязываемую к кнопке функцию-обработчик события. Полный код программы:

```
from tkinter import *


class Block:
    def __init__(self, master, func):
        self.ent = Entry(master, width=20)
        self.but = Button(master, text="Преобразовать")
        self.lab = Label(master, width=20, bg='black', fg='white')
        self.but['command'] = getattr(self, func)
        self.ent.pack()
        self.but.pack()
        self.lab.pack()

    def str_to_sort(self):
        s = self.ent.get()
        s = s.split()
        s.sort()
        self.lab['text'] = ' '.join(s)

    def str_reverse(self):
        s = self.ent.get()
        s = s.split()
        s.reverse()
        self.lab['text'] = ' '.join(s)


root = Tk()

first_block = Block(root, 'str_to_sort')
second_block = Block(root, 'str_reverse')

root.mainloop()
```

Выражение `getattr(self, func)`, где вместо *func* подставляется строка 'str_to_sort' или 'str_reverse', преобразуется в выражение `self.str_to_sort` или `self.str_reverse`.

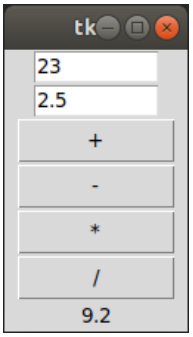
При выполнения этого кода в окне будут выведены два однотипных блока, кнопки которых выполняют разные действия.



Класс можно сделать более гибким, если жестко не задавать свойства виджетов, а передавать значения как аргументы в конструктор, после чего присваивать их соответствующим опциям при создании объектов.

Практическая работа

Напишите простейший калькулятор, состоящий из двух текстовых полей, куда пользователь вводит числа, и четырех кнопок "+", "-", "*", "/". Результат вычисления должен отображаться в метке. Если арифметическое действие выполнить невозможно (например, если были введены буквы, а не числа), то в метке должно появляться слово "ошибка".



Урок 2. Виджеты Button, Label, Entry

В этом уроке рассмотрим подробнее три наиболее простых и популярных виджета GUI – кнопку, метку и однострочное текстовое поле. В `tkinter` объекты этих элементов интерфейса порождаются соответственно от классов `Button`, `Label` и `Entry`.

Свойства и методы виджетов бывают относительно общими, характерными для многих типов, а также частными, зачастую встречающимися только у какого-то одного класса. В любом случае список настраиваемых свойств велик. В этом курсе мы будем рассматривать только ключевые свойства и методы классов пакета `tkinter`.

В Tkinter существует три способа конфигурирования свойств виджетов:

- в момент создания объекта,
- с помощью метода `config`, он же `configure`,
- путем обращения к свойству как к элементу словаря.

Button – кнопка

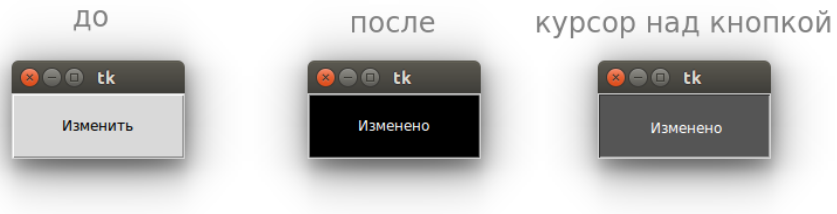
Самыми важными свойствами виджета класса `Button` являются `text`, с помощью которого устанавливается надпись на кнопке, и `command` для установки действия, то есть того, что будет происходить при нажатии на кнопку.

По умолчанию размер кнопки соответствует ширине и высоте текста, однако с помощью свойств `width` и `height` эти параметры можно изменить. Единицами измерения в данном случае являются знакоместа.

Такие свойства как `bg`, `fg`, `activebackground` и `activeforeground` определяют соответственно цвет фона и текста, цвет фона и текста во время нажатия и установки курсора мыши над кнопкой.

```
from tkinter import *


def change():
    b1['text'] = "Изменено"
    b1['bg'] = '#000000'
    b1['activebackground'] = '#555555'
    b1['fg'] = '#ffffff'
    b1['activeforeground'] = '#ffffff'


root = Tk()
b1 = Button(text="Изменить",
            width=15, height=3)
b1.config(command=change)
b1.pack()
root.mainloop()
```

Здесь свойство `command` устанавливается с помощью метода `config`. Однако можно было сделать и так: `b1['command'] = change`. Вот так будет выглядеть кнопка после запуска программы и после нажатия на нее:



Label – метка

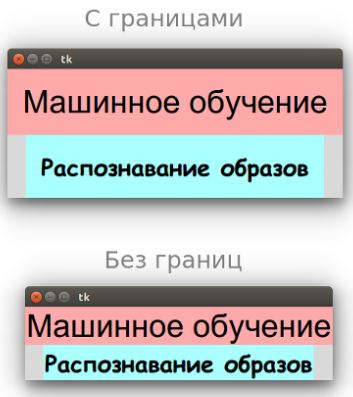
Виджет `Label` просто отображает текст в окне и служит в основном для информационных целей (вывод сообщений, подпись других элементов интерфейса). Свойства метки во многом схожи с таковыми у кнопки. Однако у меток нет опции `command`. Поэтому связать их с событием можно только с помощью метода `bind`.

На примере объекта типа `Label` рассмотрим свойство `font` – шрифт.

```
from tkinter import *

root = Tk()

l1 = Label(text="Машинное обучение",
           font="Arial 32")
l2 = Label(text="Распознавание образов",
           font=("Comic Sans MS",
                 24, "bold"))
l1.config(bd=20, bg='#ffaaaa')
l2.config(bd=20, bg='#aaffff')
l1.pack()
l2.pack()
root.mainloop()
```

Значение шрифта можно передать как строку или как кортеж. Второй вариант удобен, если имя шрифта состоит из двух и более слов. После названия шрифта можно указать размер и стиль.

Также как `font` свойство `bd` есть не только у метки. С его помощью регулируется размер границ (единица измерения – пиксель):



Бывает, что метки и кнопки не присваивают переменным, если потом к ним в коде не приходится обращаться. Их создают от класса и сразу размещают:

```
from tkinter import *


def take():
    lab['text'] = "Выдано"


root = Tk()

Label(text="Пункт выдачи").pack()
Button(text="Взять", command=take).pack()

lab = Label(width=10, height=1)
lab.pack()

root.mainloop()
```

В данном примере только у одной метки есть связь с переменной, так как одно из ее свойств может быть изменено в процессе выполнения программы.

Entry – однострочное текстовое поле

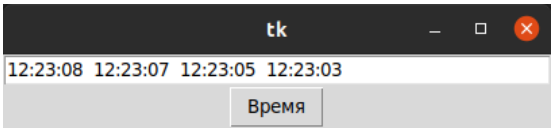
Текстовые поля предназначены для ввода информации пользователем. Однако нередко также для вывода, если предполагается, что текст из них будет скопирован. Текстовые поля как элементы графического интерфейса бывают однострочными и многострочными. В **`tkinter`** вторым соответствует класс **`Text`**, который будет рассмотрен позже.

Свойства экземпляров **`Entry`** во многом схожи с двумя предыдущими виджетами. А вот методы – нет. Из текстового поля можно взять текст. За это действие отвечает метод **`get`**. В текстовое поле можно вставить текст методом **`insert`**. Также можно удалить текст методом **`delete`**.

Метод **`insert`** принимает позицию, в которую надо вставлять текст, и сам текст. Такой код

```
from tkinter import *
from datetime import datetime as dt


def insert_time():
    t = dt.now().time()
    e1.insert(0, t.strftime('%H:%M:%S  '))


root = Tk()
e1 = Entry(width=50)
but = Button(text="Время",
             command=insert_time)
e1.pack()
but.pack()
root.mainloop()
```

приведет к тому, что после каждого нажатия на кнопку будет вставляться новое время пред уже существующей в поле строкой.



Если 0 в `insert` заменить на константу `END`, то вставляться будет в конец. Можно указать любое число-индекс знакоместа, тогда вставка будет производиться куда-либо в середину строки.

Метод `delete` принимает один или два аргумента. В первом случае удаляется один символ в указанной позиции. Во втором – срез между двумя указанными индексами, не включая последний. Если нужно полностью очистить поле, то первым аргументом должен быть 0, вторым – `END`.

Практическая работа

Напишите программу, состоящую из семи кнопок, цвета которых соответствуют цветам радуги. При нажатии на ту или иную кнопку в текстовое поле должен вставляться код цвета, а в метку – название цвета.

Коды цветов в шестнадцатеричной кодировке: #ff0000 – красный, #ff7d00 – оранжевый, #ffff00 – желтый, #00ff00 – зеленый, #007dff – голубой, #0000ff – синий, #7d00ff – фиолетовый.

Примерно должно получиться так:

Для выравнивания строки по центру в текстовом поле используется свойство `justify` со значением `CENTER`.

Урок 3. Метод pack

Прежде чем продолжить знакомство с виджетами GUI остановимся на вопросе их расположения в окне. Это важный вопрос, так как от продуманности интерфейса во многом зависит удобство использования программы. Организуя виджеты в пространстве, программист отчасти становится дизайнером, разработчиком интерфейсов.

В Tkinter существует три так называемых менеджера геометрии – упаковщик, сетка и размещение по координатам. В этом уроке будет рассмотрен первый как наиболее простой и часто используемый, остальные два упомянем позже.

Упаковщик (packer) вызывается методом `pack`, который имеется у всех виджетов-объектов. Мы уже использовали его. Если к элементу интерфейса не применить какой-либо из менеджеров геометрии, то он не отобразится в окне. При этом в одном окне (или любом другом родительском виджете) нельзя комбинировать разные менеджеры. Если вы начали размещать виджеты методом `pack`, не пытайтесь тут же использовать методы `grid` (сетка) и `place` (место).

Если в упаковщики не передавать аргументы, то виджеты будут располагаться вертикально, друг над другом. Тот объект, который первым вызовет `pack`, будет вверху. Который вторым – под первым, и так далее.

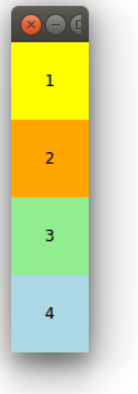
У метода `pack` есть параметр `side` (сторона), который принимает одно из четырех значений-констант `tkinter` – `TOP`, `BOTTOM`, `LEFT`, `RIGHT` (верх, низ, лево, право). По умолчанию, когда в `pack` не указывается `side`, его значение равняется `TOP`. Из-за этого виджеты располагаются вертикально.

Создадим четыре раскрашенные метки

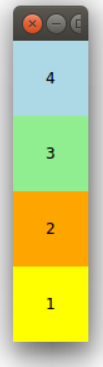
```
…
l1 = Label(width=7, height=4, bg='yellow', text="1")
l2 = Label(width=7, height=4, bg='orange', text="2")
l3 = Label(width=7, height=4, bg='lightgreen', text="3")
l4 = Label(width=7, height=4, bg='lightblue', text="4")
…
```

и рассмотрим разные комбинации значений сайда:

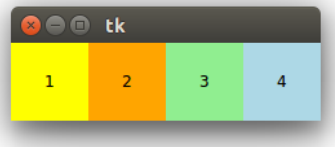
```
l1.pack()
l2.pack()
l3.pack()
l4.pack()
```



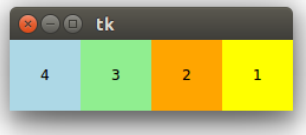
```
l1.pack(side=BOTTOM)
l2.pack(side=BOTTOM)
l3.pack(side=BOTTOM)
l4.pack(side=BOTTOM)
```



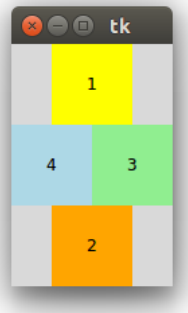

```
l1.pack(side=LEFT)
l2.pack(side=LEFT)
l3.pack(side=LEFT)
l4.pack(side=LEFT)
```



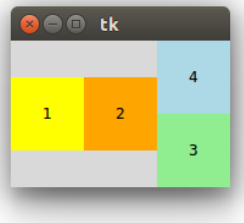

```
l1.pack(side=RIGHT)
l2.pack(side=RIGHT)
l3.pack(side=RIGHT)
l4.pack(side=RIGHT)
```




```
l1.pack(side=TOP)
l2.pack(side=BOTTOM)
l3.pack(side=RIGHT)
l4.pack(side=LEFT)
```




```
l1.pack(side=LEFT)
l2.pack(side=LEFT)
l3.pack(side=BOTTOM)
l4.pack(side=LEFT)
```



Проблема двух последних вариантов в том, что если надо разместить виджеты квадратом, т. е. два сверху, два снизу ровно под двумя верхними, то сделать это проблематично, если вообще возможно. Поэтому прибегают к вспомогательному виджету – фрейму (рамке), который порождается от класса `Frame`.

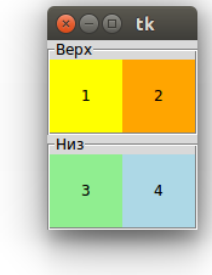
Фреймы размещают на главном окне, а уже в фреймах – виджеты:

```
from tkinter import *
root = Tk()
f_top = Frame(root)
f_bot = Frame(root)
l1 = Label(f_top, width=7, height=4,
           bg='yellow', text="1")
l2 = Label(f_top, width=7, height=4,
           bg='orange', text="2")
l3 = Label(f_bot, width=7, height=4,
           bg='lightgreen', text="3")
l4 = Label(f_bot, width=7, height=4,
           bg='lightblue', text="4")

f_top.pack()
f_bot.pack()
l1.pack(side=LEFT)
l2.pack(side=LEFT)
l3.pack(side=LEFT)
l4.pack(side=LEFT)

root.mainloop()
```

Результат:

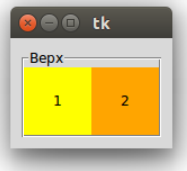
Кроме **`Frame`** существует похожий класс **`LabelFrame`** – фрейм с подписью. В отличие от простого фрейма у него есть свойство **`text`**.

```
…
f_top = LabelFrame(text="Верх")
f_bot = LabelFrame(text="Низ")
…
```

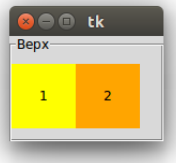


Кроме **`side`** у **`pack`** есть другие параметры-свойства. Можно задавать внутренние (**`ipadx`** и **`ipady`**) и внешние (**`padx`** и **`pady`**) отступы:

```
f_top.pack(padx=10, pady=10)

l1.pack(side=LEFT)
l2.pack(side=LEFT)
```




```
f_top.pack(ipadx=10, ipady=10)

l1.pack(side=LEFT)
l2.pack(side=LEFT)


root.mainloop()
```



Когда устанавливаются внутренние отступы, то из-за того, что **side** прибивает виджет к левой границе, справа получаем отступ в 20 пикселей, а слева – ничего. Можно частично решить проблему, заменив внутренние отступы рамки на внешние отступы у меток.

```
f_top.pack()
l1.pack(side=LEFT, padx=10, pady=10)
l2.pack(side=LEFT, padx=10, pady=10)
```

Но тут появляется промежуток между самими метками. Чтобы его убрать, пришлось бы каждый виджет укладывать в свой собственный фрейм. Отсюда делаем вывод, что упаковщик Tkinter удобен только для относительно простых интерфейсов.

Следующие два свойства – **fill** (заполнение) и **expand** (расширение). По-умолчанию **expand** равен нулю (другое значение – единица), а **fill** – **NONE** (другие значения **BOTH**, **X**, **Y**). Создадим окно с одной меткой:

```
from tkinter import *
root = Tk()
l1 = Label(text="This is a label",
           width=30, height=10,
           bg="lightgreen")
l1.pack()
root.mainloop()
```

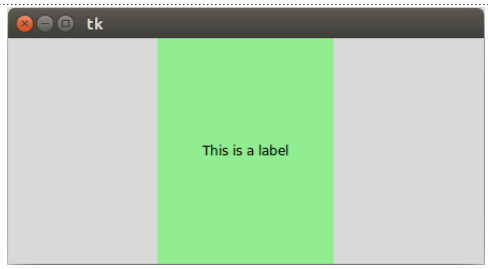
Если начать расширять окно или сразу раскрыть его на весь экран, то метка окажется вверху по вертикали и в середине по горизонтали. Причина, по которой метка не в середине по вертикали заключается в том, что **side** по-умолчанию равен TOP, и метку прибивает к верху.

Если установить свойство **expand** в 1, то при расширении окна метка будет всегда в середине:

```
…
l1.pack(expand=1)
…
```

Свойство **fill** заставляет виджет заполнять все доступное пространство. Заполнить его можно во всех направлениях или только по одной из осей:

```
…
l1.pack(expand=1, fill=Y)
…
```

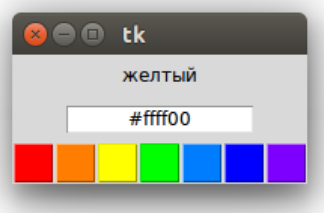


Последняя опция метода **pack** – **anchor** (якорь) – может принимать значения **N** (north – север), **S** (south – юг), **W** (west – запад), **E** (east – восток) и их комбинации:

```
…
l1.pack(expand=1, anchor=SE)
…
```

Практическая работа

Перепишите программу из практической работы предыдущего урока так, чтобы интерфейс выглядел примерно следующим образом:

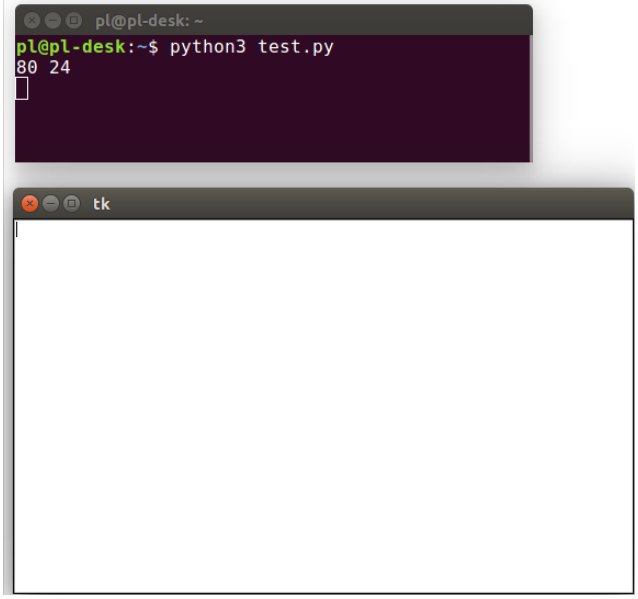


Урок 4. Text – многострочное текстовое поле

В этом уроке рассмотрим, как с помощью Tkinter запрограммировать такой элемент интерфейса как многострочное текстовое поле. Этот виджет часто встречается при заполнении веб-форм. В приложениях для десктопов он редок, если не считать программы "Терминал", где по-сути вы работаете в большом текстовом поле.

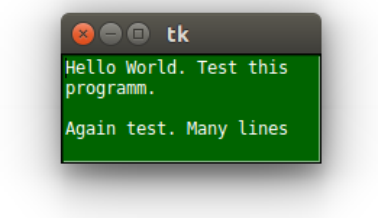
В `tkinter` многострочное текстовое поле создается от класса `Text`. По умолчанию его размер равен 80-ти знакоместам по горизонтали и 24-м по вертикали.

```
from tkinter import *
root = Tk()
text = Text()
print(text['width'], text['height'])
text.pack()
root.mainloop()
```



Однако эти свойства можно изменять с помощью опций `width` и `height`. Есть возможность конфигурировать шрифт, цвета и другое.

```
from tkinter import *
root = Tk()

text = Text(width=25, height=5, bg="darkgreen",
            fg='white', wrap=WORD)

text.pack()
root.mainloop()
```



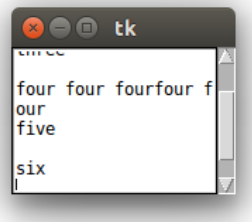
Значение `WORD` опции `wrap` позволяет переносить слова на новую строку целиком, а не по буквам.

Text и Scrollbar

Если в текстовое поле вводится больше линий текста, чем его высота, то оно само будет прокручиваться вниз. При просмотре прокручивать вверх-вниз можно с помощью колеса мыши и стрелками на клавиатуре. Однако бывает удобнее пользоваться скроллером – полосой прокрутки.

В `tkinter` скроллеры производятся от класса `Scrollbar`. Объект-скроллер связывают с виджетом, которому он требуется. Это не обязательно многострочное текстовое поле. Часто полосы прокрутки бывают нужны спискам, которые будут рассмотрены позже.

```
from tkinter import *
root = Tk()

text = Text(width=20, height=7)
text.pack(side=LEFT)

scroll = Scrollbar(command=text.yview)
scroll.pack(side=LEFT, fill=Y)

text.config(yscrollcommand=scroll.set)

root.mainloop()
```



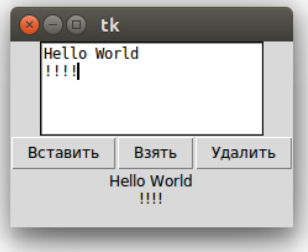
Здесь создается скроллер, к которому с помощью опции `command` привязывается прокрутка текстового поля по оси *y* – `text.yview`. В свою очередь текстовому полю опцией `yscrollcommand` устанавливается ранее созданный скроллер – `scroll.set`.

Методы Text

Основные методы у `Text` такие же как у `Entry` – `get`, `insert`, `delete`. Однако, если в случае однострочного текстового поля было достаточно указать один индекс элемента при вставке или удалении, то в случае многострочного надо указывать два – номер строки и номер символа в этой строке (другими словами, номер столбца). При этом нумерация строк начинается с единицы, а столбцов – с нуля.

```
from tkinter import *


def insert_text():
    s = "Hello World"
    text.insert(1.0, s)


def get_text():
```

```
    s = text.get(1.0, END)
    label['text'] = s


def delete_text():
    text.delete(1.0, END)


root = Tk()

text = Text(width=25, height=5)
text.pack()

frame = Frame()
frame.pack()
Button(frame, text="Вставить",
       command=insert_text).pack(side=LEFT)
Button(frame, text="Взять",
       command=get_text).pack(side=LEFT)
Button(frame, text="Удалить",
       command=delete_text).pack(side=LEFT)

label = Label()
label.pack()

root.mainloop()
```



Методы **`get`** и **`delete`** могут принимать не два, а один аргумент. В таком случае будет обрабатываться только один символ в указанной позиции.

Теги

Особенностью текстового поля библиотеки Tk является возможность форматировать текст в нем, то есть придавать его разным частям разное оформление. Делается это с помощью методов **`tag_add`** и **`tag_config`**. Первый добавляет тег, при этом надо указать его произвольное имя и отрезок текста, к которому он будет применяться. Метод **`tag_config`** настраивает тегу стили оформления.

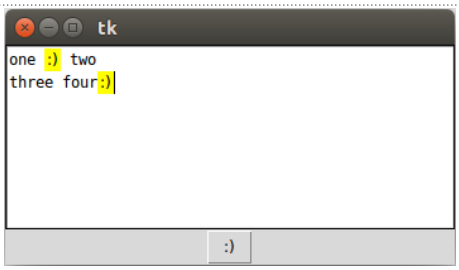
```
from tkinter import *
root = Tk()

text = Text(width=50, height=10)
text.pack()
text.insert(1.0, "Hello world!\nline two")
```

```
text.tag_add('title', 1.0, '1.end')
text.tag_config('title', justify=CENTER,
                font=("Verdana", 24, 'bold'))

root.mainloop()
```

Вставка виджетов в текстовое поле

В `Text` можно вставлять другие виджеты помощью метода `window_creat`. Потребность в этом не велика, однако может быть интересна с объектами типа `Canvas`. Данный класс будет изучен позже. В примере ниже вставляется метка в текущую (`INSERT`) позицию курсора.

```
from tkinter import *

def smile():
    label = Label(text=":)", bg="yellow")
    text.window_create(INSERT, window=label)

root = Tk()

text = Text(width=50, height=10)
text.pack()

button = Button(text=":)", command=smile)
button.pack()

root.mainloop()
```



Размещение метки в функции позволяет каждый раз при вызове функции создавать новую метку. Иначе, если бы метка была в основной ветке программы, предыдущая исчезала бы.

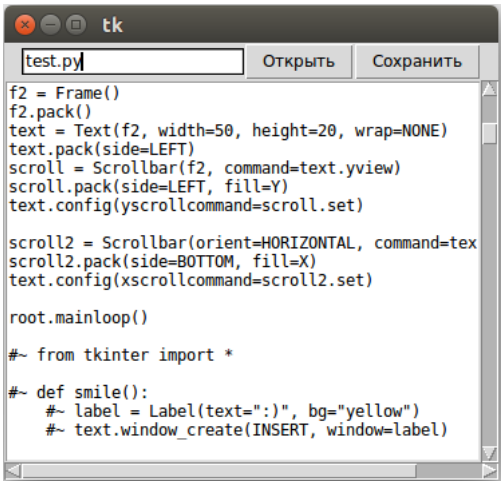
Практическая работа

Напишите программу, состоящую из однострочного и многострочного текстовых полей и двух кнопок "Открыть" и "Сохранить". При клике на первую должен открываться на чтение файл, чье имя указано в поле класса `Entry`, а содержимое файла должно загружаться в поле типа `Text`.

При клике на вторую кнопку текст, введенный пользователем в экземпляр `Text`, должен сохраняться в файле под именем, которое пользователь указал в однострочном текстовом поле.

Файлы будут читаться и записываться в том же каталоге, что и файл скрипта, если указывать имена файлов без адреса.

Для выполнения практической работы вам понадобится функция `open` языка Python и методы файловых объектов чтения и записи. Освежить знания о них можно здесь (https://younglinux.info/python/files).



Урок 5. Radiobutton и Checkbutton. Переменные Tkinter

В Tkinter от класса `Radiobutton` создаются радиокнопки, от класса `Checkbutton` – флажки.

Радиокнопки не создают по одной, а делают связанную группу, работающую по принципу переключателей. Когда включена одна, другие выключены.

Экземпляры `Checkbutton` также могут быть визуально оформлены в группу, но каждый флажок независим от остальных. Каждый может быть в состоянии "установлен" или "снят", независимо от состояний других флажков. Другими словами, в группе `Checkbutton` можно сделать множественный выбор, в группе `Radiobutton` – нет.

Radiobutton – радиокнопка

Если мы создадим две радиокнопки без соответствующих настроек, то обе они будут включены и выключить их будет невозможно:

```
from tkinter import *
root = Tk()

r1 = Radiobutton(text='First')
r2 = Radiobutton(text='Second')

r1.pack(anchor=W)
r2.pack(anchor=W)

root.mainloop()
```

Эти переключатели никак не связаны друг с другом. Кроме того для них не указано исходное значение, должны ли они быть в состоянии "вкл" или "выкл". По-умолчанию они включены.

Связь устанавливается через общую переменную, разные значения которой соответствуют включению разных радиокнопок группы. У всех кнопок одной группы свойство `variable` устанавливается в одно и то же значение – связанную с группой переменную. А свойству `value` присваиваются разные значения этой переменной.

В Tkinter нельзя использовать любую переменную для хранения состояний виджетов. Для этих целей предусмотрены специальные классы-переменные пакета `tkinter` – `BooleanVar`, `IntVar`, `DoubleVar`, `StringVar`. Первый класс позволяет принимать своим экземплярам только булевы значения (0 или 1 и `True` или `False`), второй – целые, третий – дробные, четвертый – строковые.

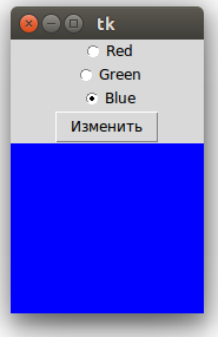
```
r_var = BooleanVar()
r_var.set(0)
r1 = Radiobutton(text='First',
                 variable=r_var, value=0)
r2 = Radiobutton(text='Second',
                 variable=r_var, value=1)
```

Здесь переменной *r_var* присваивается объект типа `BooleanVar`. С помощью метода `set` он устанавливается в значение 0.

При запуске программы включенной окажется первая радиокнопка, так как значение ее опции **value** совпадает с текущим значением переменной **r_var**. Если кликнуть по второй радиокнопке, то она включится, а первая выключится. При этом значение **r_var** станет равным 1.

В программном коде обычно требуется "снять" данные о том, какая из двух кнопок включена. Делается это с помощью метода **get** экземпляров переменных Tkinter.

```
from tkinter import *


def change():
    if var.get() == 0:
        label['bg'] = 'red'
    elif var.get() == 1:
        label['bg'] = 'green'
    elif var.get() == 2:
        label['bg'] = 'blue'


root = Tk()

var = IntVar()
var.set(0)
red = Radiobutton(text="Red",
                  variable=var, value=0)
green = Radiobutton(text="Green",
                    variable=var, value=1)
blue = Radiobutton(text="Blue",
                   variable=var, value=2)
button = Button(text="Изменить",
                command=change)
label = Label(width=20, height=10)
red.pack()
green.pack()
blue.pack()
button.pack()
label.pack()

root.mainloop()
```



В функции *change* в зависимости от считанного значения переменной *var* ход выполнения программы идет по одной из трех веток.

Мы можем избавиться от кнопки "Изменить", связав функцию *change* или любую другую со свойством `command` радиокнопок. При этом не обязательно, чтобы радиокнопки, объединенные в одну группу, вызывали одну и ту же функцию.

```
from tkinter import *


def red_label():
    label['bg'] = 'red'


def green_label():
    label['bg'] = 'green'


def blue_label():
    label['bg'] = 'blue'


root = Tk()

var = IntVar()
var.set(0)
Radiobutton(text="Red", command=red_label,
            variable=var, value=0).pack()
Radiobutton(text="Green", command=green_label,
            variable=var, value=1).pack()
Radiobutton(text="Blue", command=blue_label,
            variable=var, value=2).pack()
label = Label(width=20, height=10, bg='red')
label.pack()

root.mainloop()
```

Здесь метка будет менять цвет при клике по радиокнопкам.

Если посмотреть на пример выше, можно заметить, что код радиокнопок почти идентичный, как и код связанных с ними функций. В таких случаях более грамотно поместить однотипный код в класс.

```
from tkinter import *


def paint(color):
    label['bg'] = color


class RBColor:
    def __init__(self, color, val):
        Radiobutton(
            text=color.capitalize(),
            command=lambda i=color: paint(i),
            variable=var, value=val).pack()


root = Tk()
```

```
var = IntVar()
var.set(0)
RBColor('red', 0)
RBColor('green', 1)
RBColor('blue', 2)
label = Label(width=20, height=10, bg='red')
label.pack()

root.mainloop()
```

В двух последних вариациях кода мы не используем метод `get`, чтобы получить значение переменной *var*. В данном случае нам это не требуется, потому что цвет метки меняется в момент клика по соответствующей радиокнопке и не находится в зависимости от значения переменной. Несмотря на это использовать переменную в настройках радиокнопок необходимо, так как она обеспечивает связывание их в единую группу и выключение одной при включении другой.

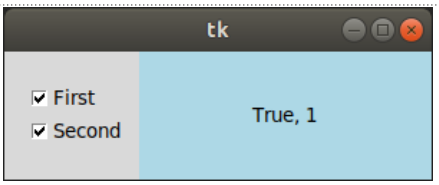
Checkbutton – флажок

Флажки не требуют установки между собой связи, поэтому может возникнуть вопрос, а нужны ли тут переменные Tkinter? Они нужны, чтобы снимать сведения о состоянии флажков. По значению связанной с `Checkbutton` переменной можно определить, установлен флажок или снят, что в свою очередь повлияет на ход выполнения программы.

У каждого флажка должна быть своя переменная Tkinter. Иначе при включении одного флажка, другой будет выключаться, так как значение общей tkinter-переменной изменится и не будет равно значению опции `onvalue` первого флажка.

```
from tkinter import *
root = Tk()


def show():
    s = f'{var1.get()}, ' \
        f'{var2.get()}'
    lab['text'] = s


frame = Frame()
frame.pack(side=LEFT)

var1 = BooleanVar()
var1.set(0)
c1 = Checkbutton(frame, text="First",
                 variable=var1,
                 onvalue=1, offvalue=0,
                 command=show)
c1.pack(anchor=W, padx=10)

var2 = IntVar()
var2.set(-1)
c2 = Checkbutton(frame, text="Second",
                 variable=var2,
                 onvalue=1, offvalue=0,
```

```
                  command=show)
c2.pack(anchor=W, padx=10)

lab = Label(width=25, height=5, bg="lightblue")
lab.pack(side=RIGHT)

root.mainloop()
```



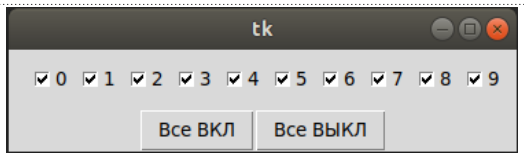
С помощью опции **onvalue** устанавливается значение, которое принимает связанная переменная при включенном флажке. С помощью свойства **offvalue** – при выключенном. В данном случае оба флажка при запуске программы будут выключены, так как методом **set** были установлены отличные от **onvalue** значения.

Опцию **offvalue** можно не указывать. Однако при ее наличии можно отследить, выключался ли флажок.

С помощью методов **select** и **deselect** флажков можно их программно включать и выключать. То же самое относится к радиокнопкам.

```
from tkinter import *


class CheckButton:
    def __init__(self, master, title):
        self.var = BooleanVar()
        self.var.set(0)
        self.title = title
        self.cb = Checkbutton(
            master, text=title, variable=self.var,
            onvalue=1, offvalue=0)
        self.cb.pack(side=LEFT)


def ch_on():
    for ch in checks:
        ch.cb.select()


def ch_off():
    for ch in checks:
        ch.cb.deselect()


root = Tk()

f1 = Frame()
f1.pack(padx=10, pady=10)
checks = []
for i in range(10):
    checks.append(CheckButton(f1, i))
```

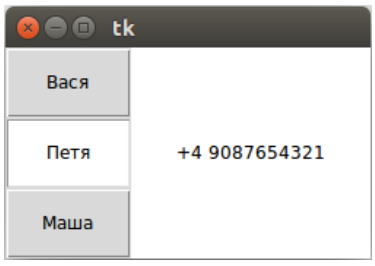
```
f2 = Frame()
f2.pack()
button_on = Button(f2, text="Все ВКЛ",
                   command=ch_on)
button_on.pack(side=LEFT)
button_off = Button(f2, text="Все ВЫКЛ",
                    command=ch_off)
button_off.pack(side=LEFT)

root.mainloop()
```



Практическая работа

Виджеты Radiobatton и Checkbutton поддерживают большинство свойств оформления внешнего вида, которые есть у других элементов графического интерфейса. При этом у Radiobutton есть особое свойство **`indicatoron`**. По-умолчанию он равен единице, в этом случае радиокнопка выглядит как нормальная радиокнопка. Однако если присвоить этой опции ноль, то виджет Radiobutton становится похожим на обычную кнопку по внешнему виду. Но не по смыслу.

Напишите программу, в которой имеется несколько объединенных в группу радиокнопок, индикатор которых выключен (`indicatoron=0`). Если какая-нибудь кнопка включается, то в метке должна отображаться соответствующая ей информация. Обычных кнопок в окне быть не должно.



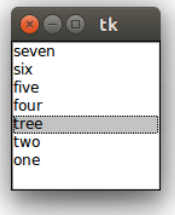
Помните, что свойство **`command`** есть не только у виджетов класса Button.

Урок 6. Виджет Listbox

От класса `Listbox` создаются списки – виджеты, внутри которых в столбик перечисляются элементы. При этом можно выбирать один или множество элементов списка.

В Tkinter сначала создается экземпляр `Listbox`, после этого он заполняется с помощью метода `insert`.

```
from tkinter import *
root = Tk()

lbox = Listbox(width=15, height=8)
lbox.pack()

for i in ('one', 'two', 'tree', 'four', 'five', 'six', 'seven'):
    lbox.insert(0, i)

root.mainloop()
```



Первым аргументом в `insert` передается индекс места, куда будет вставлен элемент. Если нужно вставлять в конец списка, то индекс обозначают константой `END`. Вторым аргументом передается вставляемый элемент.

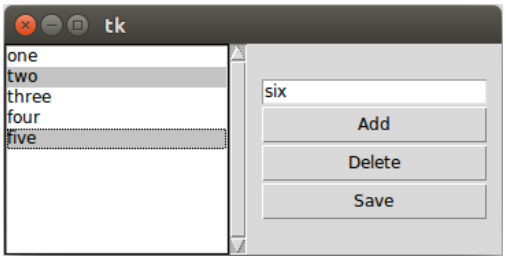
По-умолчанию в Listbox, кликая мышкой, можно выбирать только один элемент. Если необходимо обеспечить множественный выбор, то для свойства `selectmode` можно установить значение `EXTENDED`. В этом режиме можно выбрать сколь угодно элементов, зажав `Ctrl` или `Shift`.

Если для Listbox необходим скроллер, то он настраивается также как для текстового поля. В программу добавляется виджет `Scrollbar` и связывается с экземпляром `Listbox`.

С помощью метода `get` из списка можно получить один элемент по индексу, или срез, если указать два индекса. Метод `delete` удаляет один элемент или срез.

Метод `curselection` позволяет получить в виде кортежа **индексы** выбранных элементов экземпляра `Listbox`.

Ниже приводится пример программы, иллюстрирующей применение методов `get`, `insert`, `delete` и `curselection` класса `Listbox`. Первая кнопка добавляет введенную пользователем в текстовое поле строку в список, вторая кнопка удаляет выбранные элементы из списка, третья – сохраняет список в файл.



```
from tkinter import *


def add_item():
    box.insert(END, entry.get())
    entry.delete(0, END)


def del_list():
    select = list(box.curselection())
    select.reverse()
    for i in select:
        box.delete(i)


def save_list():
    f = open('list000.txt', 'w')
    f.writelines("\n".join(box.get(0, END)))
    f.close()


root = Tk()

box = Listbox(selectmode=EXTENDED)
box.pack(side=LEFT)
scroll = Scrollbar(command=box.yview)
scroll.pack(side=LEFT, fill=Y)
box.config(yscrollcommand=scroll.set)

f = Frame()
f.pack(side=LEFT, padx=10)
entry = Entry(f)
entry.pack(anchor=N)
Button(f, text="Add", command=add_item)\
    .pack(fill=X)
Button(f, text="Delete", command=del_list)\
    .pack(fill=X)
Button(f, text="Save", command=save_list)\
    .pack(fill=X)

root.mainloop()
```

В функции *del_list* кортеж выбранных элементов превращается в список, после чего выполняется его реверс, т. е. переворот. Это делается для того, чтобы удаление элементов происходило с конца списка. Иначе программа бы неверно работала, так как удаление элемента приводило бы к изменению индексов всех следующих за ним. Если же удалять с конца, то индексы впереди стоящих не меняются.

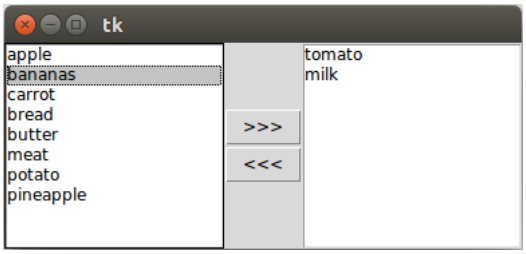
Метод **`curselection`** возвращает кортеж. Кортежи не имеют метода **`reverse`**, поэтому мы преобразуем его в список.

В функции *save_list* кортеж строк-элементов, который вернул метод **`get`**, преобразуется в одну строку с помощью строкового метода **`join`** через разделитель `'\n'`. Это делается для того, чтобы элементы списка записались в файл столбиком.

Listbox – достаточно сложный виджет. Кроме рассмотренных он обладает другими методами, а также множеством свойств.

Практическая работа

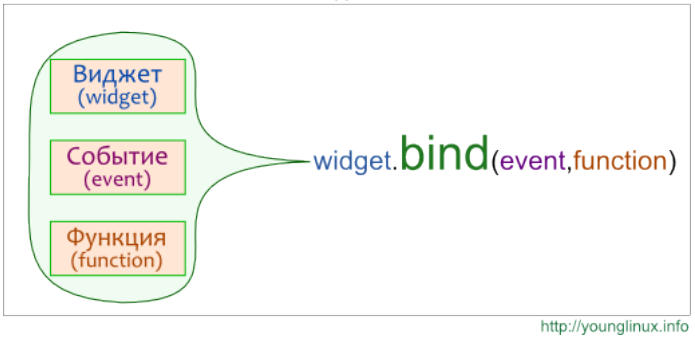
Напишите программу, состоящую из двух списков `Listbox`. В первом будет, например, перечень товаров, заданный программно. Второй изначально пуст, пусть это будет перечень покупок. При клике на одну кнопку товар должен переходить из одного списка в другой. При клике на вторую кнопку – возвращаться (человек передумал покупать). Предусмотрите возможность множественного выбора элементов списка и их перемещения.



Урок 7. Метод bind

В `tkinter` с помощью метода `bind` между собой связываются виджет, событие и действие. Например, виджет – кнопка, событие – клик по ней левой кнопкой мыши, действие – отправка сообщения. Другой пример: виджет – текстовое поле, событие – нажатие `Enter`, действие – получение текста из поля методом `get` для последующей обработки программой. Действие оформляют как функцию или метод, которые вызываются при наступлении события.

Добавление функциональности графическому элементу с помощью метода bind



Один и тот же виджет можно связать с несколькими событиями. В примере ниже используется одна и та же функция-обработчик, однако могут быть и разные:

```
from tkinter import *
root = Tk()

def change(event):
    b['fg'] = "red"
    b['activeforeground'] = "red"

b = Button(text='RED', width=10, height=3)
b.bind('<Button-1>', change)
b.bind('<Return>', change)

b.pack()

root.mainloop()
```

Здесь цвет текста на кнопке меняется как при клике по ней (событие `<Button-1>`), так и при нажатии клавиши `Enter` (событие `<Return>`). Однако `Enter` сработает, только если кнопка предварительно получила фокус. В данном случае для этого надо один раз нажать клавишу `Tab`. Иначе нажатие `Enter` будет относиться к окну, но не к кнопке.

У функций-обработчиков, которые вызываются через `bind`, а не через свойство `command`, должен быть обязательный параметр *event*, через который передается событие. Имя *event* – соглашение, идентификатор может иметь другое имя, но обязательно должен стоять на первом месте в функции, или может быть вторым в методе:

```
from tkinter import *
root = Tk()

class RedButton:
    def __init__(self):
```

```
        self.b = Button(text='RED', width=10, height=3)
        self.b.bind('<Button-1>', self.change)
        self.b.pack()

    def change(self, event):
        self.b['fg'] = "red"
        self.b['activeforeground'] = "red"

RedButton()
root.mainloop()
```

Что делать, если в функцию надо передать дополнительные аргументы? Например, клик левой кнопкой мыши по метке устанавливает для нее один шрифт, а клик правой кнопкой мыши – другой. Можно написать две разные функции:

```
from tkinter import *
root = Tk()

def font1(event):
    l['font'] = "Verdana"

def font2(event):
    l['font'] = "Times"

l = Label(text="Hello World")

l.bind('<Button-1>', font1) # ЛКМ
l.bind('<Button-3>', font2) # ПКМ
l.pack()

root.mainloop()
```

Но это не совсем правильно, так как код тела функций фактически идентичен, а имя шрифта можно передавать как аргумент. Лучше определить одну функцию:

```
…
def changeFont(event, font):
    l['font'] = font
…
```

Однако возникает проблема, как передать дополнительный аргумент функции в метод **`bind`**? Ведь в этот метод мы передаем объект-функцию, но не вызываем ее. Нельзя написать `l.bind('<Button-1>', changeFont(event, "Verdana"))`. Потому что как только вы поставили после имени функции скобки, значит вызвали ее, то есть заставили тело функции выполниться. Если в функции нет оператора **`return`**, то она возвращает **`None`**. Поэтому получается, что даже если правильно передать аргументы, то в метод **`bind`** попадет **`None`**, но не объект-функция.

На помощь приходят так называемые анонимные объекты-функции Python, которые создаются инструкцией **`lambda`**. Применительно к нашей программе выглядеть это будет так:

```
…
l.bind('<Button-1>',
       lambda e, f="Verdana": changeFont(e, f))
l.bind('<Button-3>',
       lambda e, f="Times": changeFont(e, f))
```

…

Лямбда-функции можно использовать не только с методом **`bind`**, но и опцией **`command`**, имеющейся у ряда виджет. Если функция передается через **`command`**, ей не нужен параметр *event*. Здесь обрабатывается только одно основное событие для виджета – клик левой кнопкой мыши.

У меток нет **`command`**, однако это свойство есть у кнопок:

```
from tkinter import *


def changeFont(font):
    l['font'] = font


root = Tk()
l = Label(text="Hello World")
l.pack()
Button(command=
       lambda f="Verdana": changeFont(f))\
    .pack()
Button(command=
       lambda f="Times": changeFont(f))\
    .pack()
root.mainloop()
```

Практическая работа

Напишите программу по следующему описанию. Нажатие `Enter` в однострочном текстовом поле приводит к перемещению текста из него в список (экземпляр **`Listbox`**). При двойном клике (**`<Double-Button-1>`**) по элементу-строке списка, она должна копироваться в текстовое поле.

Урок 8. События

Обычно, чтобы приложение с графическим интерфейсом что-то делало, должны происходить те или иные события, чаще всего представляющие собой воздействие человека на элементы GUI.

Можно выделить три основных типа событий: производимые мышью, нажатиями клавиш на клавиатуре, а также события, возникающие в результате изменения виджетов. Нередко обрабатываются сочетания. Например, клик мышью с зажатой клавишей на клавиатуре.

При вызове метода **bind** событие передается в качестве первого аргумента.

Название события заключается в кавычки, а также в угловые скобки < и >. События описывается с помощью зарезервированных ключевых слов.

Часто используемые события, производимые мышью:

- **<Button-1>** – клик левой кнопкой мыши
- **<Button-2>** – клик средней кнопкой мыши
- **<Button-3>** – клик правой кнопкой мыши
- **<Double-Button-1>** – двойной клик левой кнопкой мыши
- **<Motion>** – движение мыши
- и т. д.

Пример:

```
from tkinter import *


def b1(event):
    root.title("Левая кнопка мыши")


def b3(event):
    root.title("Правая кнопка мыши")
```

```
def move(event):
    x = event.x
    y = event.y
    s = "Движение мышью {}x{}".format(x, y)
    root.title(s)


root = Tk()
root.minsize(width=500, height=400)

root.bind('<Button-1>', b1)
root.bind('<Button-3>', b3)
root.bind('<Motion>', move)

root.mainloop()
```

В этой программе меняется надпись в заголовке главного окна в зависимости от того, двигается мышь, щелкают левой или правой кнопкой.

Событие (**`Event`**) – это один из объектов **`tkinter`**. У событий есть атрибуты, как и у многих других объектов. В примере в функции **`move`** извлекаются значения атрибутов **`x`** и **`y`** объекта *event*, в которых хранятся координаты местоположения курсора мыши в пределах виджета, по отношению к которому было сгенерировано событие. В данном случае виджетом является главное окно, а событием – **`<Motion>`**, т. е. перемещение мыши.

В программе ниже выводится информация об экземпляре **`Event`** и некоторым его свойствам. Все атрибуты можно посмотреть с помощью команды `dir(event)`. У разных событий они одни и те же, меняются только значения. Для тех или иных событий часть атрибутов не имеет смысла, такие свойства имеют значения по умолчанию.

В примере хотя обрабатывается событие нажатия клавиши клавиатуры, в поля **`x`**, **`y`**, **`x_root`**, **`y_root`** сохраняются координаты положения на экране курсора мыши.

```
from tkinter import *


def event_info(event):
    print(type(event))
    print(event)
    print(event.time)
    print(event.x_root)
    print(event.y_root)


root = Tk()
root.bind('a', event_info)
root.mainloop()
```

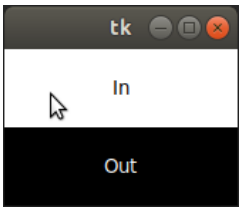
Пример выполнения программы:

```
<class 'tkinter.Event'>
<KeyPress event state=Mod2 keysym=a keycode=38 char='a' x=9 y=7>
8379853
37
92
```

Для событий с клавиатуры буквенные клавиши можно записывать без угловых скобок (например, `'a'`).

Для неалфавитных клавиш существуют специальные зарезервированные слова. Например, **`<Return>`** – нажатие клавиши `Enter`, **`<space>`** – `пробел`. Заметим, что есть событие **`<Enter>`**, которое не имеет отношения к нажатию клавиши `Enter`, а происходит, когда курсор заходит в пределы виджета.

Рассмотрим программу:




```
from tkinter import *


def enter_leave(event):
    if event.type == '7':
        event.widget['text'] = 'In'
    elif event.type == '8':
        event.widget['text'] = 'Out'


root = Tk()

lab1 = Label(width=20, height=3, bg='white')
lab1.pack()
lab1.bind('<Enter>', enter_leave)
lab1.bind('<Leave>', enter_leave)

lab2 = Label(width=20, height=3, bg='black',
             fg='white')
lab2.pack()
lab2.bind('<Enter>', enter_leave)
lab2.bind('<Leave>', enter_leave)
root.mainloop()
```

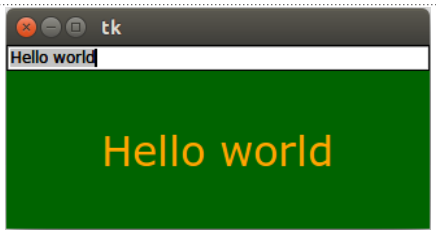
В ней две метки используют одну и ту же функцию, и каждая метка использует эту функцию для обработки двух разных событий: ввода курсора в пределы виджета и вывода во границы.

Функция, в зависимости от того, по отношению к какому виджету было зафиксировано событие, изменяет свойства только этого виджета. Как изменяет, зависит от произошедшего события.

Свойство **`event.widget`** содержит ссылку на виджет, сгенерировавший событие. Свойство **`event.type`** описывает, что это было за событие. У каждого события есть имя и номер. С помощью выражения `print(repr(event.type))` можно посмотреть его полное описание. При этом на одних платформах `str(event.type)` возвращает имя события (например, 'Enter'), на других – строковое представление номера события (например, '7').

Вернемся к событиям клавиатуры. Сочетания клавиш пишутся через тире. В случае использования так называемого модификатора, он указывается первым, детали на третьем месте. Например, `<Shift-Up>` - одновременное нажатие клавиш `Shift` и стрелки вверх, `<Control-B1-Motion>` – движение мышью с зажатой левой кнопкой и клавишей `Ctrl`.

```
from tkinter import *


def exit_win(event):
    root.destroy()


def to_label(event):
    t = ent.get()
    lbl.configure(text=t)


def select_all(event):

    def select_all2(widget):
        widget.selection_range(0, END)
        widget.icursor(END)  # курсор в конец

    root.after(10, select_all2, event.widget)


root = Tk()

ent = Entry(width=40)
ent.focus_set()
ent.pack()
lbl = Label(height=3, fg='orange',
            bg='darkgreen', font="Verdana 24")
lbl.pack(fill=X)

ent.bind('<Return>', to_label)
ent.bind('<Control-a>', select_all)
root.bind('<Control-q>', exit_win)

root.mainloop()
```



Здесь сочетание клавиш `Ctrl+a` выделяет текст в поле. Без `root.after()` выделение не работает. Метод `after` выполняет функцию, указанную во втором аргументе, через промежуток времени, указанный в первом аргументе. В третьем аргументе передается значение атрибута `widget` объекта *event*. В данном случае им будет поле *ent*. Именно оно будет передано как аргумент в функцию *select_all2* и присвоено параметру *widget*.

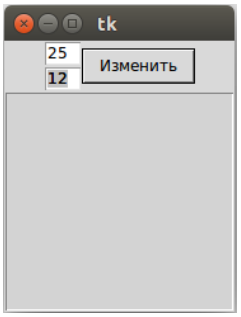
Практическая работа

Напишите программу по описанию. Размеры многострочного текстового поля определяются значениями, введенными в однострочные текстовые поля. Изменение размера происходит при нажатии мышью на кнопку, а также при нажатии клавиши `Enter`.

Цвет фона экземпляра **`Text`** светлосерый (**`lightgrey`**), когда поле не в фокусе, и белый, когда имеет фокус.

Событие получения фокуса обозначается как **`<FocusIn>`**, потери – как **`<FocusOut>`**.

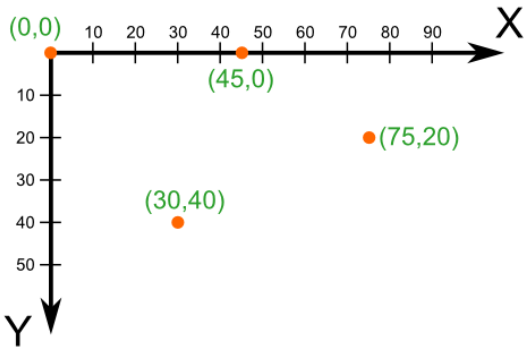
Для справки: фокус перемещается по виджетам при нажатии `Tab`, `Ctrl+Tab`, `Shift+Tab`, а также при клике по ним мышью (к кнопкам последнее не относится).



Урок 9. Canvas

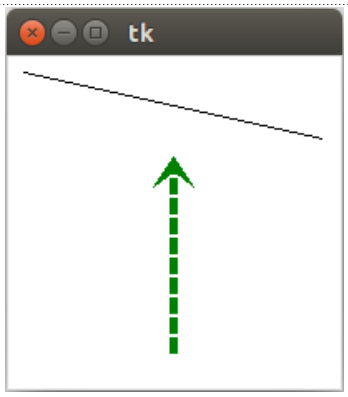
В `tkinter` от класса `Canvas` создаются объекты-холсты, на которых можно "рисовать", размещая различные фигуры и объекты. Делается это с помощью вызовов соответствующих методов.

При создании экземпляра `Canvas` необходимо указать его ширину и высоту. При размещении геометрических примитивов и других объектов указываются их координаты на холсте. Точкой отсчета является верхний левый угол.



В программе ниже создается холст. На нем с помощью метода `create_line` рисуются отрезки. Сначала указываются координаты начала (x_1, y_1), затем – конца (x_2, y_2).

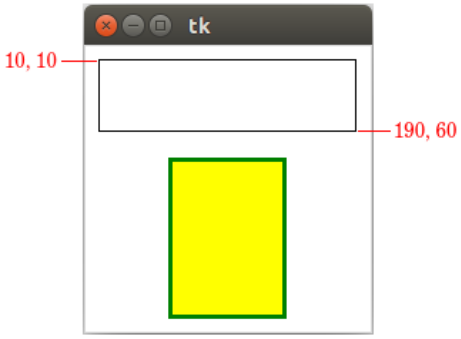
```
from tkinter import *
root = Tk()

c = Canvas(root, width=200, height=200, bg='white')
c.pack()

c.create_line(10, 10, 190, 50)

c.create_line(100, 180, 100, 60, fill='green',
                width=5, arrow=LAST, dash=(10,2),
                activefill='lightgreen',
                arrowshape="10 20 10")

root.mainloop()
```



Остальные свойства являются необязательными. Так `activefill` определяет цвет отрезка при наведении на него курсора мыши.

Создание прямоугольников методом `create_rectangle`:

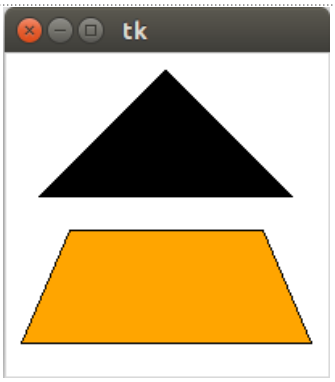
```
…
c.create_rectangle(10, 10, 190, 60)

c.create_rectangle(60, 80, 140, 190,
                   fill='yellow',
                   outline='green',
                   width=3,
                   activedash=(5, 4))
…
```



Первые координаты – верхний левый угол, вторые – правый нижний. В приведенном примере, когда на второй прямоугольник попадает курсор мыши, его рамка становится пунктирной, что определяется свойством `activedash`.

Методом `create_polygon` рисуется произвольный многоугольник путем задания координат каждой его точки:

```
…
c.create_polygon(100, 10, 20, 90, 180, 90)

c.create_polygon(40, 110, 160, 110,
                 190, 180, 10, 180,
                 fill='orange', outline='black')
…
```

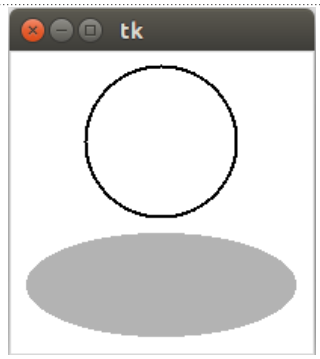


Для удобства координаты точек можно заключать в скобки:

```
…
c.create_polygon((40, 110), (160, 110),
                 (190, 180), (10, 180),
                 fill='orange', outline='black')
…
```

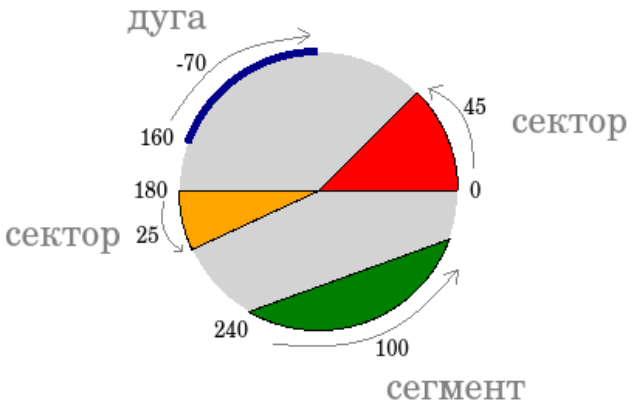
Метод `create_oval` создает эллипсы. При этом задаются координаты гипотетического прямоугольника, описывающего эллипс. Если нужно получить круг, то соответственно описываемый прямоугольник должен быть квадратом.

```
…
c.create_oval(50, 10, 150, 110, width=2)
c.create_oval(10, 120, 190, 190,
              fill='grey70', outline='white')
…
```



Более сложные для понимания фигуры получаются при использовании метода `create_arc`. В зависимости от значения опции `style` можно получить сектор (по умолчанию), сегмент (`CHORD`) или дугу (`ARC`). Также как в случае `create_oval` координаты задают прямоугольник, в который вписана окружность (или эллипс), из которой "вырезают" сектор, сегмент или дугу. Опции `start` присваивается градус начала фигуры, `extent` определяет угол поворота.

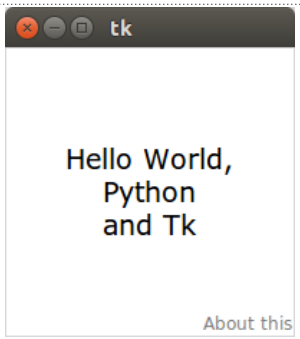
```
…
c.create_oval(10, 10, 190, 190,
              fill='lightgrey',
              outline='white')
c.create_arc(10, 10, 190, 190,
             start=0, extent=45,
             fill='red')
c.create_arc(10, 10, 190, 190,
             start=180, extent=25,
             fill='orange')
c.create_arc(10, 10, 190, 190,
             start=240, extent=100,
             style=CHORD, fill='green')
c.create_arc(10, 10, 190, 190,
             start=160, extent=-70,
             style=ARC, outline='darkblue',
             width=5)
…
```



В данном примере светло-серый круг используется исключительно для наглядности.

На холсте можно разместить текст. Делается это с помощью метода `create_text`:

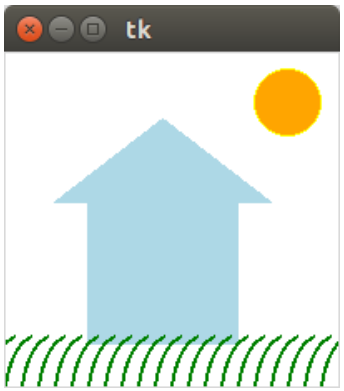
```
…
c.create_text(100, 100,
              text="Hello World,\nPython\nand Tk",
              justify=CENTER, font="Verdana 14")
c.create_text(200, 200, text="About this",
              anchor=SE, fill="grey")
…
```



По умолчанию в заданной координате располагается центр текстовой надписи. Чтобы изменить это и, например, разместить по указанной координате левую границу текста, используется якорь со значением `W` (от англ. west – запад). Другие значения: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`. Если букв, задающих сторону привязки, две, то вторая определяет вертикальную привязку (вверх или вниз «уйдет» текст от заданной координаты). Свойство `justify` определяет лишь выравнивание текста относительно себя самого.

Практическая работа

Создайте на холсте подобное изображение:



Для создания травы используется цикл.

Урок 10. Canvas. Идентификаторы, теги и анимация

Изучив размещение геометрических примитивов на экземпляре Canvas, в этом уроке рассмотрим, как можно обращаться к уже созданным фигурам для изменения их свойств, а также создадим анимацию.

В Tkinter существует два способа "пометить" фигуры, размещенные на холсте, – это идентификаторы и теги. Первые всегда уникальны для каждого объекта. Два объекта не могут иметь одни и тот же идентификатор. Теги не уникальны. Группа объектов на холсте может иметь один и тот же тег. Это дает возможность менять свойства всей группы. Отдельно взятая фигура на Canvas может иметь как идентификатор, так и тег.

Идентификаторы

Методы, создающие фигуры на холсте, возвращают численные идентификаторы этих объектов, которые можно присвоить переменным, через которые позднее обращаться к созданным фигурам.

```
from tkinter import *
root = Tk()
c = Canvas(width=300, height=300,
           bg='white')
c.focus_set()
c.pack()

ball = c.create_oval(140, 140, 160, 160,
                     fill='green')
c.bind('<Up>',
       lambda event: c.move(ball, 0, -2))
c.bind('<Down>',
       lambda event: c.move(ball, 0, 2))
c.bind('<Left>',
       lambda event: c.move(ball, -2, 0))
c.bind('<Right>',
       lambda event: c.move(ball, 2, 0))

root.mainloop()
```

В данном примере круг двигается по холсту с помощью стрелок на клавиатуре. Когда создавался круг, его идентификатор был присвоен переменной *ball*. Метод `move` объекта `Canvas` принимает идентификатор и смещение по осям.

С помощью метода `itemconfig` можно изменять другие свойства. Метод `coords` устанавливает новые координаты фигуры, если они заданы. Если указывается только идентификатор или тег, то `coords` возвращает текущие координаты.

```
from tkinter import *
root = Tk()
c = Canvas(width=200, height=200,
           bg='white')
c.pack()
```

```
rect = c.create_rectangle(
    80, 80, 120, 120, fill='lightgreen')


def in_focus(event):
    c.itemconfig(rect, fill='green', width=2)
    c.coords(rect, 70, 70, 130, 130)


c.bind('<FocusIn>', in_focus)

root.mainloop()
```

Здесь при получении холстом фокуса (нажать `Tab`) изменится цвет и размер квадрата.

Теги

В отличие от идентификаторов, которые являются уникальными для каждого объекта, один и тот же тег может присваиваться разным объектам. Дальнейшее обращение к такому тегу позволит изменить все объекты, в которых он был указан. В примере ниже эллипс и линия содержат один и тот же тег, а функция **`color`** изменяет цвет всех объектов с тегом **`group1`**. Обратите внимание, что в отличие от имени идентификатора (переменная), имя тега заключается в кавычки (строковое значение).

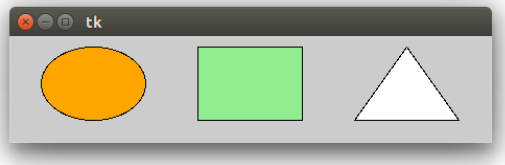
```
from tkinter import *


def color(event):
     c.itemconfig('group1', width=3,
                  fill="red")


root = Tk()
c = Canvas(width=460, height=150,
           bg='white')
c.pack()
oval = c.create_oval(30, 10, 130, 80,
                     tag="group1")
c.create_line(10, 100, 450, 100,
              tag="group1")
c.bind('<Button-3>', color)
root.mainloop()
```

Метод **`tag_bind`** позволяет привязать событие (например, щелчок кнопкой мыши) к определенной фигуре на Canvas. Таким образом, можно реализовать обращение к различным областям холста с помощью одного и того же события. Пример ниже иллюстрирует, как изменения на холсте зависят от того, где произведен клик.

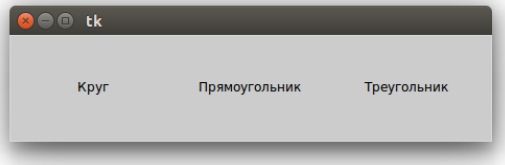
```
from tkinter import *


def oval_func(event):
    c.delete(oval)
    c.create_text(80, 50,
                  text="Круг")
```

```
def rect_func(event):
    c.delete("rect")
    c.create_text(230, 50,
                  text="Прямоугольник")


def triangle(event):
    c.delete(trian)
    c.create_text(380, 50,
                  text="Треугольник")


c = Canvas(width=460, height=100,
           bg='grey80')
c.pack()

oval = c.create_oval(30, 10, 130, 80,
                     fill="orange")
c.create_rectangle(180, 10, 280, 80,
                   tag="rect",
                   fill="lightgreen")
trian = c.create_polygon(
    330, 80, 380, 10, 430, 80,
    fill='white', outline="black")

c.tag_bind(oval, '<Button-1>', oval_func)
c.tag_bind("rect", '<Button-1>', rect_func)
c.tag_bind(trian, '<Button-1>', triangle)

mainloop()
```

До кликов



После кликов



Метод **`delete`** удаляет объект. Если нужно очистить холст, то вместо идентификаторов или тегов используется константа **`ALL`**.

Практическая работа. Анимация в tkinter

В данной программе создается анимация круга, который движется от левой границы холста до правой:

```
from tkinter import *


def motion():
    c.move(ball, 1, 0)
    if c.coords(ball)[2] < 300:
        root.after(10, motion)


root = Tk()
c = Canvas(root, width=300, height=200,
           bg="white")
c.pack()
ball = c.create_oval(0, 100, 40, 140,
                     fill='green')
motion()
root.mainloop()
```

Выражение `c.coords(ball)` возвращает список текущих координат объекта (в данном случае это *ball*). Третий элемент списка соответствует его второй координате *x*.

Метод **`after`** вызывает функцию, переданную вторым аргументом, через количество миллисекунд, указанных первым аргументом.

Изучите приведенную программу и самостоятельно запрограммируйте постепенное движение фигуры в ту точку холста, где пользователь кликает левой кнопкой мыши. Координаты события хранятся в его атрибутах **`x`** и **`y`** (`event.x`, `event.y`).

Урок 11. Окна

В этом уроке рассмотрим основные настройки окон, в которых располагаются виджеты. Обычные окна в Tkinter порождаются не только от класса **`Tk`**, но и **`Toplevel`**. От Tk принято создавать главное окно. Если создается многооконное приложение, то остальные окна создаются от Toplevel. Методы обоих классов схожи.

Размер и положение окна

По умолчанию окно приложения появляется в верхнем левом углу экрана. Его размер (ширина и высота) определяется совокупностью размеров расположенных в нем виджетов. В случае если окно пустое, то tkinter устанавливает его размер в 200 на 200 пикселей.

С помощью метода **`geometry`** можно изменить как размер окна, так и его положение. Метод принимает строку определенного формата.

```
from tkinter import *

root = Tk()

root.geometry('600x400+200+100')

root.mainloop()
```

Первые два числа в строке-аргументе **`geometry`** задают ширину и высоту окна. Вторая пара чисел обозначает смещение на экране по осям *x* и *y*. В примере окно размерностью 600 на 400 будет смещено от верхней левой точки экрана на 200 пикселей вправо и на 100 пикселей вниз.

Если перед обоими смещениями вместо плюса указывается минус, то расчет происходит от нижних правых углов экрана и окна. Так выражение `root.geometry('600x400-0-0')` заставит окно появиться в нижнем правом углу.

В аргументе метода **`geometry`** можно не указывать либо размер, либо смещение. Например, чтобы сместить окно, но не менять его размер, следует написать `root.geometry('+200+100')`.

Бывает удобно, чтобы окно появлялось в центре экрана. Методы **`winfo_screenwidth`** и **`winfo_screenheight`** возвращают количество пикселей экрана, на котором появляется окно. Рассмотрим, как поместить окно в центр, если размер окна известен:

```
…
w = root.winfo_screenwidth()
h = root.winfo_screenheight()
w = w//2 # середина экрана
h = h//2
w = w - 200 # смещение от середины
h = h - 200
root.geometry('400x400+{}+{}'.format(w, h))
…
```

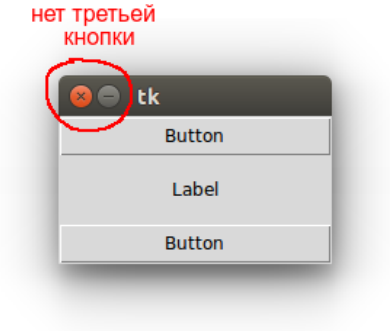
Здесь мы вычитаем половину ширины и высоты окна (по 200 пикселей). Иначе в центре экрана окажется верхний левый угол окна, а не его середина.

Если размер окна неизвестен, то его можно получить с помощью того же метода **`geometry`**, но без аргументов. В этом случае метод возвращает строку, содержащую сведения о размерах и смещении, из которой можно извлечь ширину и высоту окна.

```
from tkinter import *
root = Tk()

Button(text="Button", width=20).pack()
Label(text="Label", width=20, height=3).pack()
Button(text="Button", width=20).pack()

root.update_idletasks()
s = root.geometry()
s = s.split('+')
s = s[0].split('x')
width_root = int(s[0])
height_root = int(s[1])

w = root.winfo_screenwidth()
h = root.winfo_screenheight()
w = w // 2
h = h // 2
w = w - width_root // 2
h = h - height_root // 2
root.geometry('+{}+{}'.format(w, h))

root.mainloop()
```

Метод **`update_idletasks`** позволяет перезагрузить данные об окне после размещения на нем виджетов. Иначе **`geometry`** вернет строку, где ширина и высота равняются по одному пикселю. Видимо таковы параметры на момент запуска приложения.

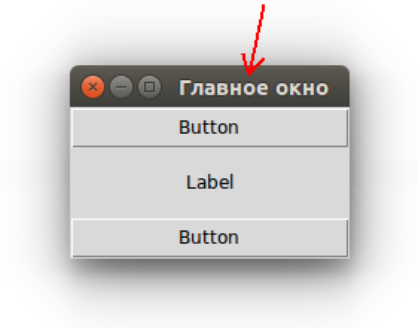
По умолчанию пользователь может разворачивать окно на весь экран, а также изменять его размер, раздвигая границы. Эти возможности можно отключить с помощью метода **`resizable`**. Так `root.resizable(False, False)` запретит изменение размеров главного окна как по горизонтали, так и вертикали. Развернуть на весь экран его также будет невозможно, при этом соответствующая кнопка разворота исчезает.



Заголовок окна

По умолчанию с стоке заголовка окна находится надпись "tk". Для установки собственного названия используется метод `title`.

```
…
root.title("Главное окно")
…
```

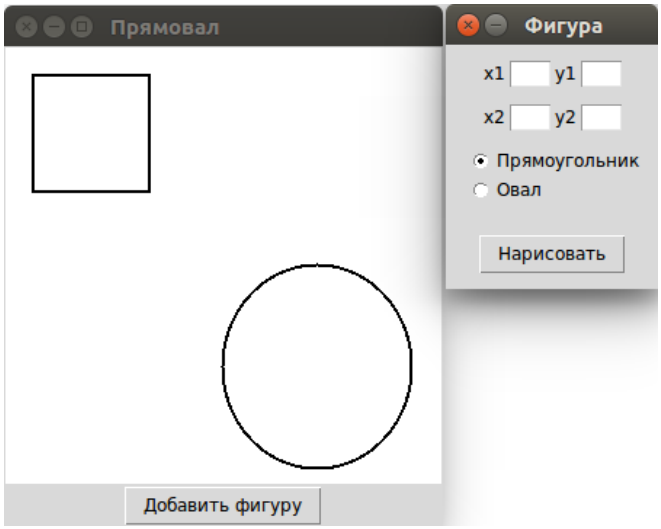


Если необходимо, заголовок окна можно вообще убрать. В программе ниже второе окно (`Toplevel`) открывается при клике на кнопку, оно не имеет заголовка, так как к нему был применен метод `overrideredirect` с аргументом `True`. Через пять секунд данное окно закрывается методом `destroy`.

```
from tkinter import *


def about():
    a = Toplevel()
    a.geometry('200x150')
    a['bg'] = 'grey'
    a.overrideredirect(True)
    Label(a, text="About this")\
        .pack(expand=1)
    a.after(5000, lambda: a.destroy())


root = Tk()
root.title("Главное окно")
Button(text="Button", width=20).pack()
Label(text="Label", width=20, height=3)\
    .pack()
Button(text="About", width=20, command=about)\
    .pack()

root.mainloop()
```

Практическая работа

Напишите программу, в которой на главном окне находятся холст и кнопка "Добавить фигуру". Кнопка открывает второе окно, включающее четыре поля для ввода координат и две радиокнопки для выбора, рисовать ли на холсте прямоугольник или овал. Здесь же находится кнопка "Нарисовать", при клике на которую соответствующая фигура добавляется на холст, а второе окно закрывается. Проверку корректности ввода в поля можно опустить.

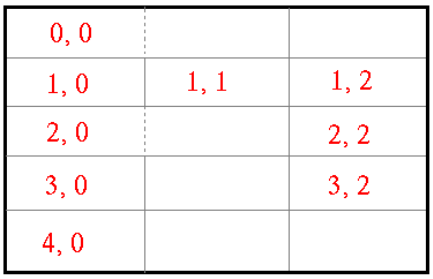


Урок 12. Метод grid

Grid является одним из трех менеджеров геометрии в Tkinter (другими являются уже рассмотренный ранее Pack, а также Place). У всех виджетов есть соответствующий данному менеджеру метод `grid`. "Grid" с английского переводится как "сетка", однако по смыслу правильнее говорить о таблице.

Табличный способ размещения предпочтителен из-за его гибкости и удобства, когда дело доходит до разработки относительно сложных интерфейсов. Grid позволяет избежать использования множества фреймов, что неизбежно в случае упаковщика Pack.

При размещении виджетов методом `grid` родительский контейнер (обычно это окно) условно разделяется на ячейки подобно таблице. Адрес каждой ячейки состоит из номера строки и номера столбца. Нумерация начинается с нуля. Ячейки можно объединять как по вертикали, так и по горизонтали.

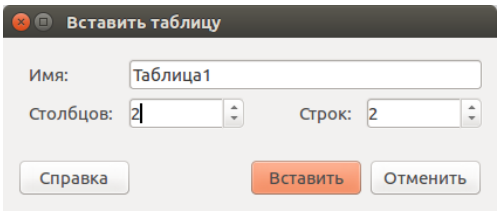


На рисунке пунктир обозначает объединение ячеек. Общая ячейка в таком случае обозначается адресом первой.

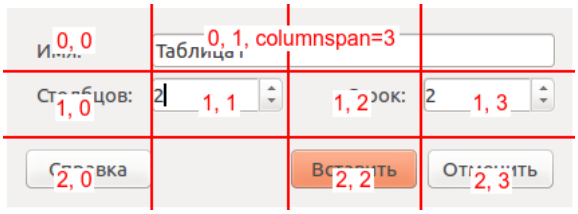
Никаких предварительных команд по разбиению родительского виджета на ячейки не выполняется. Tkinter делает это сам, исходя из указанных позиций виджетов.

Размещение виджета в той или иной ячейке задается через аргументы `row` и `column`, которым присваиваются соответственно номера строки и столбца. Чтобы объединить ячейки по горизонтали, используется атрибут `columnspan`, которому присваивается количество объединяемых ячеек. Опция `rowspan` объединяет ячейки по вертикали.

Пусть надо запрограммировать такой GUI:



Представим данный интерфейс в виде таблицы и пронумеруем ячейки, в которых будут располагаться виджеты:

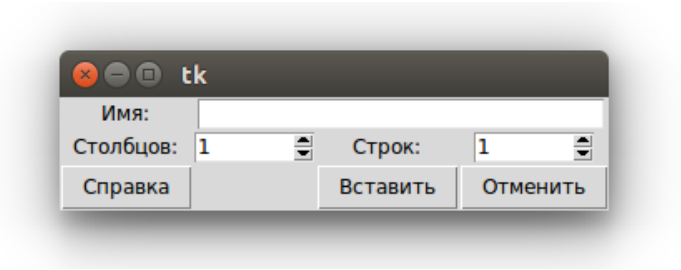


Теперь пишем код:

```
from tkinter import *
root = Tk()

Label(text="Имя:")\
    .grid(row=0, column=0)
Entry(width=30)\
    .grid(row=0, column=1, columnspan=3)

Label(text="Столбцов:")\
    .grid(row=1, column=0)
Spinbox(width=7, from_=1, to=50)\
    .grid(row=1, column=1)
Label(text="Строк:")\
    .grid(row=1, column=2)
Spinbox(width=7, from_=1, to=100)\
    .grid(row=1, column=3)

Button(text="Справка").grid(row=2, column=0)
Button(text="Вставить").grid(row=2, column=2)
Button(text="Отменить").grid(row=2, column=3)

root.mainloop()
```

Примечание. В примере используются виджеты класса **`Spinbox`**, которые не рассматривались в курсе. **`Spinbox`** похож на **`Entry`**, но для него задается список принимаемых значений, и имеется подобие скроллера.

Выполнив приведенный выше программный код, получим:

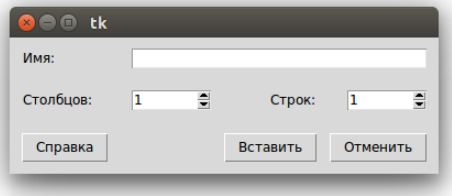


Похоже, но не совсем то, что хотелось. Теперь на помощь должны прийти другие свойства метода **`grid`**. У него, также как у **`pack`**, имеются атрибуты для задания внешних и внутренних отступов (**`padx`**, **`pady`**, **`ipadx`**, **`ipady`**).

Кроме этого есть атрибут **`sticky`** (липкий), который принимает значения направлений сторон света (N, S, W, E, NW, NE, SW, SE). Если, например, указать NW, то виджет прибьет к верхнему левому углу ячейки. Виджеты можно растягивать на весь объем ячейки (`sticky=N+S+W+E`) или только по одной из осей (`N+S` или `W+E`). Эффект от "липучки" заметен, только если виджет меньше ячейки.

```
from tkinter import *
root = Tk()

Label(text="Имя:").grid(row=0, column=0,
                        sticky=W,
                        pady=10, padx=10)
table_name = Entry()
```

```
table_name.grid(row=0, column=1,
                columnspan=3,
                sticky=W+E, padx=10)

Label(text="Столбцов:").grid(
    row=1, column=0, sticky=W,
    padx=10, pady=10)
Spinbox(width=7, from_=1, to=50)\
    .grid(row=1, column=1, padx=10)
Label(text="Строк:")\
    .grid(row=1, column=2, sticky=E)
Spinbox(width=7, from_=1, to=100)\
    .grid(row=1, column=3, sticky=E, padx=10)

Button(text="Справка")\
    .grid(row=2, column=0, pady=10, padx=10)
Button(text="Вставить")\
    .grid(row=2, column=2)
Button(text="Отменить")\
    .grid(row=2, column=3, padx=10)

root.mainloop()
```



С помощью методов **grid_remove** и **grid_forget** можно сделать виджет невидимым. Отличие между этими методами лишь в том, что **grid_remove** запоминает прежнее положение виджета. Поэтому для его отображения в прежней ячейки достаточно применить **grid** без аргументов. После **grid_forget** потребуется заново конфигурировать положение виджета.

```
from tkinter import *


def rem():
    global l1_flag
    if l1_flag == 1:
        l1.grid_remove()
        l1_flag = 0
    else:
        l1.grid()
        l1_flag = 1


def forg():
    global l2_flag
    if l2_flag == 1:
        l2.grid_forget()
        l2_flag = 0
    else:
        l2.grid(row=1)
```

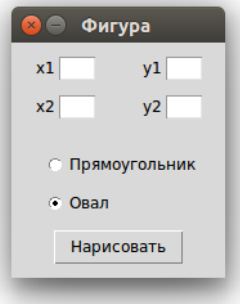
```
        l2_flag = 1


root = Tk()
l1_flag = 1
l2_flag = 1
l1 = Label(width=5, height=3, bg='blue')
l2 = Label(width=5, height=3, bg='green')
b1 = Button(bg='lightblue', command=rem)
b2 = Button(bg='lightgreen', command=forg)

l1.grid(row=0)
l2.grid(row=1)
b1.grid(row=2)
b2.grid(row=3)

root.mainloop()
```

В данной программе голубая метка после удаления будет появляться в том же месте, где была до этого, несмотря на то, что в функции *rem* к ней применяется метод `grid` без настроек. В отличие от нее зеленая метка, если ей не указать место размещения после удаления, появлялась бы под кнопками.

Скрытие виджетов бывает необходимо в тех случаях, когда, например, от выбора пользователя в одной части интерфейса зависит, какие виджеты появятся в другой.

Практическая работа

Перепрограммируйте второе окно из практической работы предыдущего урока, используя метод `grid`.



Урок 13. Диалоговые окна

Пакет **tkinter** содержит несколько модулей, предоставляющих доступ к уже готовым диалоговым окнам. Это окна различных сообщений, выбора по принципу "да-нет", открытия и сохранения файлов и др. В этом уроке рассмотрим примеры окон из модулей **messagebox** и **filedialog** пакета **tkinter**.

Модули пакета необходимо импортировать отдельно. То есть вы импортируете содержимое **tkinter** (например, `from tkinter import *`) и отдельно входящий в состав пакета **tkinter** модуль. Способы импорта на примере **messagebox** и пример вызова одной из функций модуля:

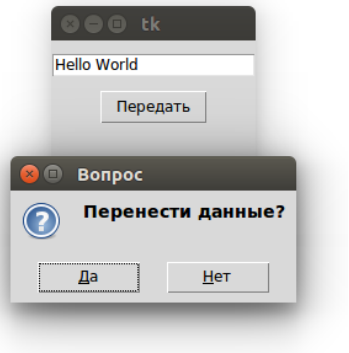
- `import tkinter.messagebox` → `tkinter.messagebox.askyesno()`
- `from tkinter.messagebox import *` → `askyesno()`
- `from tkinter import messagebox` → `messagebox.askyesno()`
- `from tkinter import messagebox as mb` (вместо *mb* может быть любой идентификатор) → `mb.askyesno()`

В уроке мы будем использовать последний вариант.

Модуль messagebox – стандартные диалоговые окна

Окно выбора "да" или "нет" – **askyesno**:

```
from tkinter import *
from tkinter import messagebox as mb


def check():
    answer = mb.askyesno(
        title="Вопрос",
        message="Перенести данные?")
    if answer:
        s = entry.get()
        entry.delete(0, END)
        label['text'] = s


root = Tk()
entry = Entry()
entry.pack(pady=10)
Button(text='Передать', command=check).pack()
label = Label(height=3)
label.pack()

root.mainloop()
```



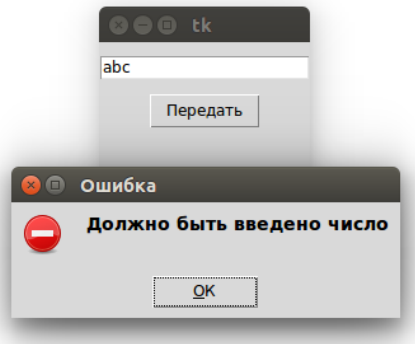
Нажатие "Да" в диалоговом окне возвращает в программу `True`, "Нет" вернет `False` (также как закрытие окна через крестик). Таким образом в коде можно обработать выбор пользователя. В данном случае если последний соглашается, то данные переносятся из поля в метку.

Опции `title` и `message` являются позиционными, так что можно указывать только значения: `askyesno("Вопрос", "Перенести данные?")`.

Подобные окна генерируются при использовании функции **askokcancel** с надписями на кнопках "OK" и "Отмена", **askquestion** (возвращает не True или False, а строки 'yes' или 'no'), **askretrycancel** ("Повторить", "Отмена"), **askyesnocancel** ("Да", "Нет", "Отмена").

Другую группу составляют окна с одной кнопкой, которые служат для вывода сообщений различного характера. Это **showerror**, **showinfo** и **showwarning**.

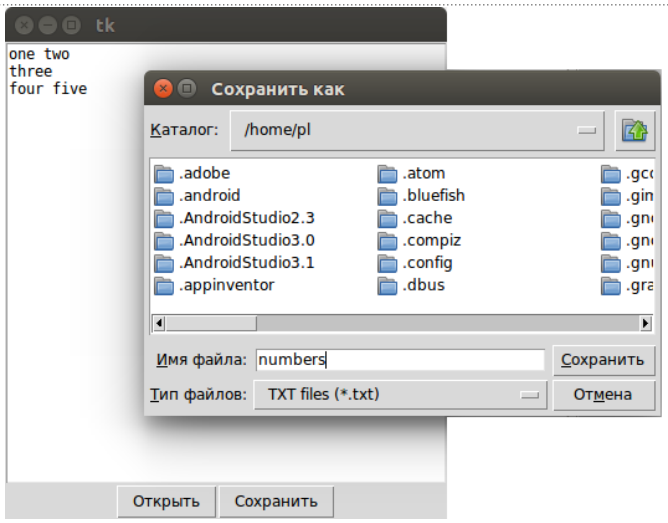
```
…
def check():
    s = entry.get()
    if not s.isdigit():
        mb.showerror(
            "Ошибка",
            "Должно быть введено число")
    else:
        entry.delete(0, END)
        label['text'] = s
…
```



Модуль filedialog – диалоговые окна открытия и сохранения файлов

Рассмотрим две функции из модуля `filedialog` – `askopenfilename` и `asksaveasfilename`. Первая предоставляет диалоговое окно для открытия файла, вторая – для сохранения. Обе возвращают имя файла, который должен быть открыт или сохранен, но сами они его не открывают и не сохраняют. Делать это уже надо программными средствами самого Python.

```
from tkinter import *
from tkinter import filedialog as fd


def insert_text():
    file_name = fd.askopenfilename()
    f = open(file_name)
    s = f.read()
    text.insert(1.0, s)
    f.close()


def extract_text():
    file_name = fd.asksaveasfilename(
        filetypes=(("TXT files", "*.txt"),
                   ("HTML files", "*.html;*.htm"),
                   ("All files", "*.*")))
    f = open(file_name, 'w')
    s = text.get(1.0, END)
    f.write(s)
    f.close()


root = Tk()
text = Text(width=50, height=25)
text.grid(columnspan=2)
b1 = Button(text="Открыть", command=insert_text)
b1.grid(row=1, sticky=E)
b2 = Button(text="Сохранить", command=extract_text)
b2.grid(row=1, column=1, sticky=W)

root.mainloop()
```



Опция `filetype` позволяет перечислить типы файлов, которые будут сохраняться или открываться, и их расширения.

Примечание. В приведенном коде при размещении текстового поля методом `grid` не указаны аргументы `row` и `column`. В таких случаях подразумевается, что их значениями являются нули.

Практическая работа

В приведенной в уроке программе с функциями `askopenfilename` и `asksaveasfilename` генерируются исключения, если диалоговые окна были закрыты без выбора или указания имени файлов.

Напишите код обработки данных исключений. При этом для пользователя должно появляться информационное диалоговое окно с сообщением о том, что файл не загружен или не сохранен.

Добавьте кнопку "Очистить", которая удаляет текст из поля. Перед удалением пользователь должен подтвердить свои намерения через соответствующее диалоговое окно.

Урок 14. Виджет Menu

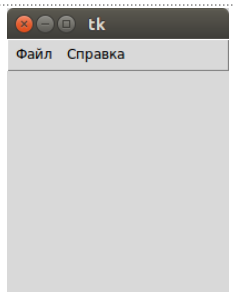
Меню – это виджет, который присутствует во многих пользовательских приложениях. Находится оно под строкой заголовка и представляет собой выпадающие списки под словами-пунктами меню. Пункты конечных списков представляют собой команды, обычно выполняющие какие-либо действия или открывающие диалоговые окна.

В `tkinter` экземпляр меню создается от класса `Menu`, далее его надо привязать к виджету, на котором оно будет расположено. Обычно таковым выступает главное окно приложения. Его опции `menu` присваивается экземпляр `Menu` через имя связанной с экземпляром переменной.

```
from tkinter import *

root = Tk()
mainmenu = Menu(root)
root.config(menu=mainmenu)

root.mainloop()
```

Если выполнить данный код, то никакого меню вы не увидите. Только тонкую полоску под заголовком окна, ведь ни одного пункта меню не было создано. Метод `add_command` добавляет пункт меню:

```
…
mainmenu.add_command(label='Файл')
mainmenu.add_command(label='Справка')
…
```

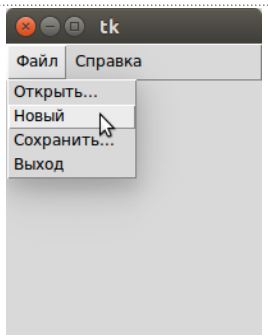


В данном случае "Файл" и "Справка" – это команды. К ним можно добавить опцию `command`, связав тем самым с какой-либо функцией-обработчиком клика. Хотя такой вариант меню имеет право на существование, в большинстве приложений панель меню содержит выпадающие списки команд, а сами пункты на панели командами по сути не являются. Клик по ним приводит лишь к раскрытию соответствующего списка.

В Tkinter проблема решается созданием новых экземпляров `Menu` и подвязыванием их к главному меню с помощью метода `add_cascade`.

```
from tkinter import *

root = Tk()

mainmenu = Menu(root)
root.config(menu=mainmenu)

filemenu = Menu(mainmenu, tearoff=0)
```

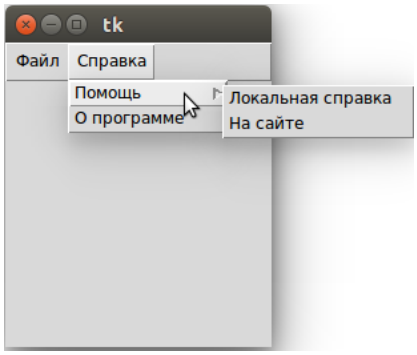
```
filemenu.add_command(label="Открыть...")
filemenu.add_command(label="Новый")
filemenu.add_command(label="Сохранить...")
filemenu.add_command(label="Выход")

helpmenu = Menu(mainmenu, tearoff=0)
helpmenu.add_command(label="Помощь")
helpmenu.add_command(label="О программе")

mainmenu.add_cascade(label="Файл",
                     menu=filemenu)
mainmenu.add_cascade(label="Справка",
                     menu=helpmenu)

root.mainloop()
```



На основное меню *mainmenu*, добавляются не команды, а другие меню. У *filemenu* и *helpmenu* в качестве родительского виджета указывается не *root*, а *mainmenu*. Команды добавляются только к дочерним меню. Значение 0 опции **`tearoff`** отключает возможность откреплення подменю, иначе его можно было бы делать плавающим кликом мыши по специальной линии. В случае `tearoff=0` она отсутствует.

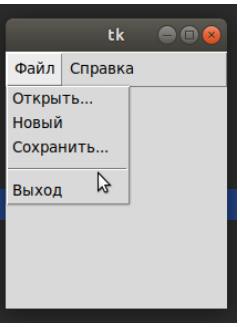
Точно также можно подвязывать дочерние меню к *filemenu* и *helpmenu*, создавая многоуровневые списки пунктов меню.

```
…
helpmenu = Menu(mainmenu, tearoff=0)

helpmenu2 = Menu(helpmenu, tearoff=0)
helpmenu2.add_command(
    label="Локальная справка")
helpmenu2.add_command(
    label="На сайте")

helpmenu.add_cascade(label="Помощь",
                     menu=helpmenu2)

helpmenu.add_command(label="О программе")
…
```



Метод `add_separator` добавляет линию разделитель в меню. Используется для визуального разделения групп команд.

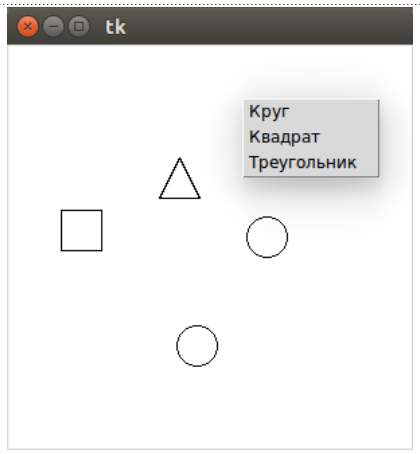
```
root.config(menu=mainmenu)

filemenu = Menu(mainmenu, tearoff=0)
filemenu.add_command(label="Открыть...")
filemenu.add_command(label="Новый")
filemenu.add_command(label="Сохранить...")
filemenu.add_separator()
filemenu.add_command(label="Выход")

helpmenu = Menu(mainmenu, tearoff=0)
```



В `tkinter` можно создать всплывающее меню, оно же контекстное (если настроить его появление по клику правой кнопкой мыши). Для этого экземпляр меню подвязывается не через опцию `menu` к родительскому виджету, а к меню применяется метод `post`, аргументами которого являются координаты того места, где должно появляться меню.

```
from tkinter import *


def circle():
    c.create_oval(
        x, y, x + 30, y + 30)


def square():
    c.create_rectangle(
        x, y, x + 30, y + 30)


def triangle():
    c.create_polygon(
        x, y, x - 15, y + 30, x + 15, y + 30,
        fill='white', outline='black')


def popup(event):
    global x, y
    x = event.x
    y = event.y
    menu.post(event.x_root, event.y_root)
```

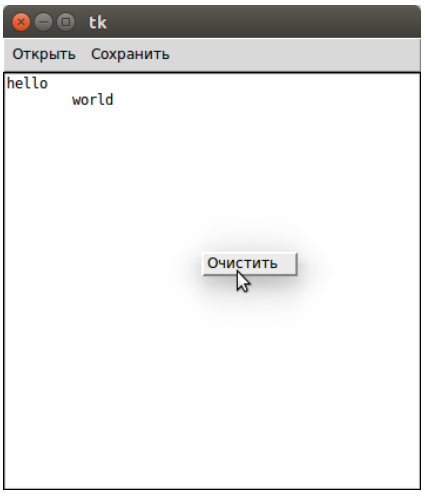
```
x = 0
y = 0

root = Tk()
c = Canvas(width=300, height=300, bg='white')
c.pack()
c.bind("<Button-3>", popup)
menu = Menu(tearoff=0)
menu.add_command(label="Круг",
                 command=circle)
menu.add_command(label="Квадрат",
                 command=square)
menu.add_command(label="Треугольник",
                 command=triangle)
root.mainloop()
```



Здесь при клике на холсте правой кнопкой мыши в этой точке всплывает меню. При выборе пункта меню рисуется соответствующая фигура в этом же месте.

Практическая работа

Измените программу из практической работы предыдущего урока так, чтобы открытие и сохранение файлов выполнялось не через экземпляры `Button`, а через `Menu`. Команду очистки текстового поля поместите в контекстное меню.



Урок 15. Метод place

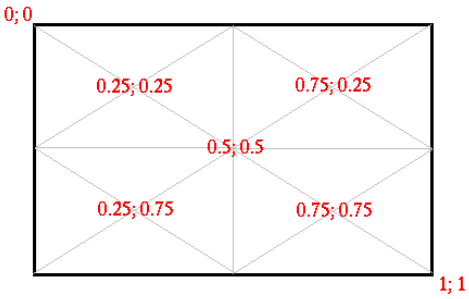
В конце курса рассмотрим третий и последний менеджер геометрии библиотеки **tk** – `Place`, который размещает виджеты по координатам. В `tkinter` использование данного управляющего размещением реализуется через метод `place` виджетов.

Методом `place` виджету указывается его положение либо в абсолютных значениях (в пикселях), либо в долях родительского окна, то есть относительно. Также абсолютно и относительно можно задавать размер самого виджета.

Основными параметрами `place` являются:

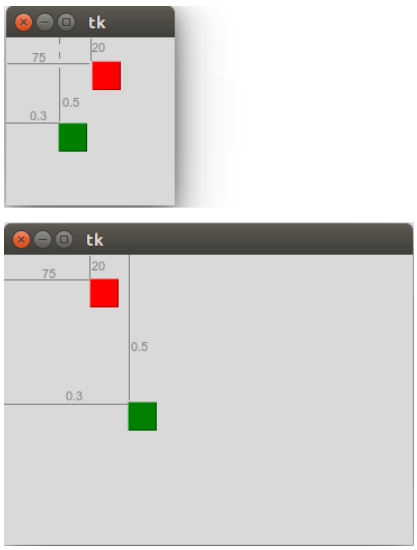
- `anchor` (якорь) – определяет часть виджета, для которой задаются координаты. Принимает значения N, NE, E, SE, SW, W, NW или CENTER. По умолчанию NW (верхний левый угол).
- `relwidth`, `relheight` (относительные ширина и высота) – определяют размер виджета в долях его родителя.
- `relx`, `rely` – определяют относительную позицию в родительском виджете. Координата (0; 0) – у левого верхнего угла, (1; 1) – у правого нижнего.
- `width`, `height` – абсолютный размер виджета в пикселях. Значения по умолчанию (когда данные опции опущены) приравниваются к естественному размеру виджета, то есть к тому, который определяется при его создании и конфигурировании.
- `x`, `y` – абсолютная позиция в пикселях. Значения по умолчанию приравниваются к нулю.

Схема с указанием относительных координат:



Для лучшего понимания разницы между абсолютным и относительным позиционированием рассмотрим пример:

```
from tkinter import *

root = Tk()
root.geometry('150x150')

Button(bg='red').place(x=75, y=20)
Button(bg='green').place(relx=0.3, rely=0.5)

root.mainloop()
```



Кнопка, позиция которой была жестко задана, не изменяет своего положения, когда окно раздвигается. Кнопка с относительными координатами смещается. Опции `relx` и `rely` для нее по-прежнему 0.3 и 0.5, но уже относительно нового размера окна.

Тоже самое с размерами виджетов. Если они относительны, то с изменением размера родительского виджета, их размеры также будут изменяться. Если требуется, чтобы виджет всегда сохранял свой размер, последний должен указываться как абсолютный. Или не указываться вовсе, тогда будет приравнен к естественному.

```
…
Label(bg='white').place(x=10, y=10,
                        width=50, height=30)
Label(bg='green').place(x=10, y=50,
                        relwidth=0.3, relheight=0.15)
…
```

Комбинируя различные варианты позиционирования и установки размеров, можно добиться интересных эффектов, а также неожиданных спецэффектов. Поэтому методом `place` следует пользоваться с осторожностью, хорошо понимая, что вы делаете.

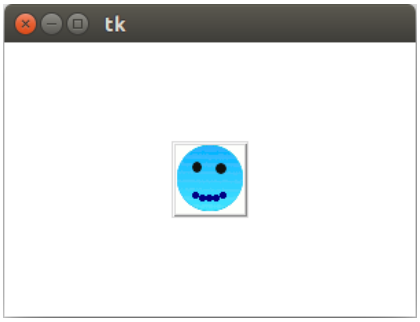
Если размер окна не меняется, а интерфейс сложен, то вероятно управляющий размещением `Place` может быть лучшим выбором, так как с его помощью можно выполнить тонкую настройку и создать наиболее аккуратный интерфейс.

Практическая работа

За рамками данного курса было оставлено несколько классов пакета **`tkinter`**. Среди них **`PhotoImage`**, позволяющий использовать в программе внешние изображения форматов GIF и PGM. Экземпляры **`PhotoImage`** можно размещать на различных виджетах через опцию **`image`**.

```
…
img = PhotoImage(file='smile.gif')
l=Label(image=img)
…
```

Напишите программу, состоящую из главного окна и кнопки, на которой изображен смайлик. При клике на кнопку она должна оказываться в новом случайном месте окна. Размер окна может меняться.



Урок 16. Модуль tkinter.ttk

В состав пакета `tkinter` входит модуль `ttk`, содержащий классы более стилизованных и современных виджет. По умолчанию их внешний вид зависит от операционной системы.

В коде ниже для сравнения в окне размещено несколько пар аналогичных виджетов из модулей `tkinter` и `tkinter.ttk`.

```
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()

tk.Button(text="Hello").pack()
ttk.Button(text="Hello").pack()

tk.Checkbutton(text="Hello").pack()
ttk.Checkbutton(text="Hello").pack()

tk.Radiobutton(text="Hello").pack()
ttk.Radiobutton(text="Hello").pack()

root.mainloop()
```

В операционной системе Ubuntu они будут выглядеть так:

Поскольку имена классов аналогичных виджетов совпадают, мы импортируем модули так, чтобы их пространства имен не перекрывались. При этом конструкторы классов вызываются через имена (в данном случае псевдонимы) модулей.

Если же нам не нужны виджеты модуля `tkinter`, подобные которым есть в `tkinter.ttk`, удобнее импортировать всё пространство имен как одного, так и второго модуля.

```
from tkinter import *
from tkinter.ttk import *

root = Tk()

b = Button(text="Hello")
b.pack()

t = Text(width=10, height=5)
t.pack()

print(root.__class__)
print(b.__class__)
```

```
print(t.__class__)

root.mainloop()
```

В данном случае пространство имен **tkinter.ttk**, который импортируется вторым, перекрывает часть имен **tkinter**. Поэтому выражение `Button(text="Hello")` вызывает конструктор класса **Button**, находящийся в модуле **tkinter.ttk**. В то же время в этом модуле нет классов **Tk** и **Text**. Следовательно, экземпляры этих классов создаются от базовых классов **tkinter**. В консоль будет выведено:

```
<class 'tkinter.Tk'>
<class 'tkinter.ttk.Button'>
<class 'tkinter.Text'>
```

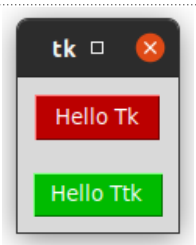
Как уже должно быть понятно, наборы виджетов обоих модулей не полностью перекрываются. Есть общие (такие как **Label**, **Button**, **Entry**), есть характерные только для **tkinter** (например, **Listbox** и **Canvas**) и только для **ttk** (например, **Combobox**, **Notebook**).

Сложность заключается в том, что в **ttk** свойства виджетов программируются не совсем так как в **tkinter**. Если в приложении сочетаются элементы сразу обоих модулей, код программы становится менее ясным.

Для виджетов из **tkinter.ttk** многие свойства задаются с помощью экземпляра, созданного от класса **ttk.Style**. Основная идея **ttk** – отделить оформление виджета от описания его поведения.

Импортируем модули, не перекрывая пространства их имен, и сравним установку свойств для двух кнопок.

```
from tkinter import *
from tkinter import ttk

root = Tk()

b_tk = Button(text="Hello Tk")
b_ttk = ttk.Button(text="Hello Ttk")

b_tk.config(background="#b00",
            foreground="#fff")

style = ttk.Style()
style.configure("TButton",
                background="#0b0",
                foreground="#fff")

b_tk.pack(padx=10, pady=10)
b_ttk.pack(padx=10, pady=10)
root.mainloop()
```



Мы задаем свойства для экземпляра `Button` из модуля `tkinter` с помощью метода кнопки `config`. Однако то же самое могли бы сделать, передав значения в конструктор:

```
b_tk = Button(text="Hello Tk",
              background="#b00",
              foreground="#fff")
```

Настроить так кнопку из модуля `ttk` нельзя. Вместо этого мы должны создать экземпляр от класса `Style` и уже через него изменять свойства по умолчанию.

В коде выше используется метод `configure`. Здесь первым аргументом в него передается имя стиля (`TButton`), связанного с классом объектов, для которых производится настройка. В данном случае это кнопки. Для уточнения имени стиля можно воспользоваться методом `winfo_class`:

```
…
print(ttk.Label().winfo_class())
print(ttk.Button().winfo_class())
…
```

Вывод:

```
TLabel
TButton
```

Что делать, если в программе нужны, например, кнопки разного стиля? Можно создать свой стиль, унаследовав его от исходного, и изменять лишь отдельные свойства.

```
from tkinter import *
from tkinter.ttk import *

root = Tk()

style = Style()
style.configure("G.TButton", foreground="green")

Button(text="First", style="G.TButton").pack()
Button(text="Second").pack()

root.mainloop()
```

В примере выше созданный стиль называется "G". После точки указывается его родитель (в данном случае это стиль `TButton`. При создании первой кнопки через опцию `style` указываем применяемый стиль.

По-умолчанию для второй кнопки используется стиль `TButton`. Мы его не меняли, хотя могли бы изменить.

На самом деле острой или частой необходимости изменять внешний вид виджетов нет. Наоборот, в приложениях приветствуется стандартный и единообразный внешний вид и поведение. Это дает пользователю интуитивно понятный интерфейс.

Кроме того, в `ttk` есть разные темы. Можно менять не отдельные виджеты, а целиком тему оформления приложения. С помощью методов `theme_names` и `theme_use` можно выяснить список тем (который зависит от вашей ОС) и текущую тему:

```
from tkinter.ttk import *

style = Style()
print(style.theme_names())
print(style.theme_use())
```

Вывод:

```
('clam', 'alt', 'default', 'classic')
default
```

Если в метод **`theme_use`** передать название темы, она будет применена. В программе ниже при нажатии клавиши `Enter` изменяется тема приложения:

```
from tkinter import *
from tkinter.ttk import *

root = Tk()
style = Style()

e = Entry(justify='center')
e.pack()
Button(text="Hello").pack()
Label(text="Hello").pack()

themes = style.theme_names()
i = 0


def theme_next(event):
    global i
    style.theme_use(themes[i])
    e.delete(0, END)
    e.insert(0, themes[i])
    if i < len(themes)-1:
        i += 1
    else:
        i = 0


root.bind('<Return>', theme_next)
root.mainloop()
```

Описание свойств виджетов в **`ttk`** не всегда совпадает с тем, как это делается в базовом **`tkinter`**. Выше в примере с красной и зеленой кнопками с помощью опций **`foreground`** и **`background`** мы устанавливаем цвета текста и фона кнопки, когда она не находится под курсором мыши, то есть когда она не активна.

Чтобы переопределить цвета по умолчанию при наводе курсора, для кнопок **`tkinter`** мы должны использовать опции **`activeforeground`** и **`activebackground`**. Однако эти свойства не работают для виджетов из **`ttk`**:

```
…
b_tk.config(background="red",
            foreground="white",
            activebackground="orange",
            activeforeground="white")
```

```
style = ttk.Style()
style.configure("TButton",
                background="green",
                foreground="white",
                activebackground="lightgreen",
                activeforeground="black")
…
```

Код выше выполнится без ошибок. Однако фон зеленой кнопки при нажатии на нее останется светло-серым, он не станет светло-зеленым.

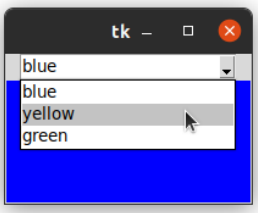
В данном случае вместо метода `configure` следует использовать метод `map` объектов типа `Style`:

```
from tkinter import *
from tkinter.ttk import *

root = Tk()

Button(text="Hello World").pack(
    padx=40, pady=40,
    ipadx=20, ipady=20
)

st = Style()
st.map('TButton',
       foreground=[('!active', 'purple'),
                   ('pressed', 'orange'),
                   ('active', 'red')],
       background=[
                   ('pressed', 'brown'),
                   ('active', 'white')]
       )

root.mainloop()
```

Здесь мы описываем для кнопки состояния двух свойств: текста (`foreground`) и фона (`background`). При этом у каждого свойства есть три состояния: неактивно, активно и нажато.

Практическая работа

Напишите программу, состоящую из выпадающего списка (`Combobox`) и холста. В списке перечисляются цвета. При выборе в списке того или иного значения цвет холста должен изменяться на соответствующий.



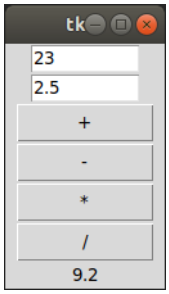
Для справки:

- Список значений передается в `Combobox` через опцию `values`.
- Метод `current` экземпляра `Combobox` позволяет указать значение, которое будет выбрано в списке изначально. Метод принимает индекс элемента из списка значений.
- Событие смены значения в выпадающем списке – `'<<ComboboxSelected>>'`
- Метод `get` экземпляра `Combobox` возвращает выбранный на данный момент элемент списка.

Примеры решений практических работ

1. Что такое Tkinter

Напишите простейший калькулятор, состоящий из двух текстовых полей, куда пользователь вводит числа, и четырех кнопок "+", "-", "*", "/". Результат вычисления должен отображаться в метке. Если арифметическое действие выполнить невозможно (например, если были введены буквы, а не числа), то в метке должно появляться слово "ошибка".




```
from tkinter import *


class MathButton:
    def __init__(self, action):
        self.but = Button(text=action, width=10, command=self.math)
        self.operator = action
        self.but.pack()

    def math(self):
        try:
            n1 = float(ent_n1.get())
            n2 = float(ent_n2.get())
        except ValueError:
            lab_result['text'] = 'Error'
        else:
            if self.operator == '+':
                lab_result['text'] = n1 + n2
            elif self.operator == '-':
                lab_result['text'] = round(n1 - n2, 2)
            elif self.operator == '*':
                lab_result['text'] = round(n1 * n2, 2)
            elif self.operator == '/':
                if n2 != 0:
                    lab_result['text'] = round(n1 / n2, 2)
                else:
                    lab_result['text'] = 'ZeroDivisionError'


root = Tk()

ent_n1 = Entry(width=10)
ent_n1.pack()
ent_n2 = Entry(width=10)
ent_n2.pack()
```

```
but_summa = MathButton('+')
but_diff = MathButton('-')
but_prod = MathButton('*')
but_div = MathButton('/')
lab_result = Label(width=15)
lab_result.pack()

root.mainloop()
```

Создание класса оправдано тем, что без него придется писать код проверки ввода в текстовые поля в каждой функции или выполнять проверку в отдельной функции:

```
from tkinter import *


def test():
    try:
        n1 = float(ent_n1.get())
        n2 = float(ent_n2.get())
        return n1, n2
    except ValueError:
        lab_result['text'] = 'Error'
        return False


def summa():
    nums = test()
    if nums:
        lab_result['text'] = nums[0] + nums[1]


def diff():
    nums = test()
    if nums:
        lab_result['text'] = nums[0] - nums[1]


def product():
    nums = test()
    if nums:
        lab_result['text'] = nums[0] * nums[1]


def div():
    nums = test()
    if nums:
        try:
            n = nums[0] / nums[1]
            lab_result['text'] = round(n, 2)
        except ZeroDivisionError:
            lab_result['text'] = 'ZeroDivisionError'


root = Tk()

ent_n1 = Entry(width=10)
```

```
ent_n1.pack()
ent_n2 = Entry(width=10)
ent_n2.pack()
but_summa = Button(text='+', width=10, command=summa)
but_summa.pack()
but_diff = Button(text='-', width=10, command=diff)
but_diff.pack()
but_prod = Button(text='*', width=10, command=product)
but_prod.pack()
but_div = Button(text='/', width=10, command=div)
but_div.pack()
lab_result = Label(width=15)
lab_result.pack()

root.mainloop()
```

Разрозненные функции также могут быть помещены в один класс как разные методы объекта:

```
from tkinter import *


class MathButton:
    def __init__(self, sign, action):
        self.but = Button(text=sign, width=10,
                          command=getattr(self, action))
        self.but.pack()

    @staticmethod
    def test():
        try:
            n1 = float(ent_n1.get())
            n2 = float(ent_n2.get())
            return n1, n2
        except ValueError:
            lab_result['text'] = 'Error'
            return False

    def summa(self):
        nums = MathButton.test()
        if nums:
            lab_result['text'] = nums[0] + nums[1]

    def diff(self):
        nums = MathButton.test()
        if nums:
            lab_result['text'] = nums[0] - nums[1]

    def product(self):
        nums = MathButton.test()
        if nums:
            lab_result['text'] = nums[0] * nums[1]

    def div(self):
        nums = MathButton.test()
        if nums:
            try:
```

```
                n = nums[0] / nums[1]
                lab_result['text'] = round(n, 2)
            except ZeroDivisionError:
                lab_result['text'] = 'ZeroDivisionError'


root = Tk()

ent_n1 = Entry(width=10)
ent_n1.pack()
ent_n2 = Entry(width=10)
ent_n2.pack()

but_summa = MathButton('+', 'summa')
but_diff = MathButton('-', 'diff')
but_prod = MathButton('*', 'product')
but_div = MathButton('/', 'div')
lab_result = Label(width=15)
lab_result.pack()

root.mainloop()
```

2. Виджеты Button, Label, Entry

Напишите программу, состоящую из семи кнопок, цвета которых соответствуют цветам радуги. При нажатии на ту или иную кнопку в текстовое поле должен вставляться код цвета, а в метку – название цвета.

Коды цветов в шестнадцатеричной кодировке: #ff0000 – красный, #ff7d00 – оранжевый, #ffff00 – желтый, #00ff00 – зеленый, #007dff – голубой, #0000ff – синий, #7d00ff – фиолетовый.

Примерно должно получиться так:

Для выравнивания строки по центру в текстовом поле используется свойство `justify` со значением `CENTER`.

Если решать задачу по-простому, придется создать семь функций и семь кнопок.

```
from tkinter import *

root = Tk()

rainbow = (('#ff0000', 'красный'), ('#ff7d00', 'оранжевый'),
           ('#ffff00', 'желтый'), ('#00ff00', 'зеленый'),
           ('#007dff', 'голубой'), ('#0000ff', 'синий'),
           ('#7d00ff', 'фиолетовый'))


def insert_red():
    label['text'] = rainbow[0][1]
    entry.delete(0, END)
    entry.insert(0, rainbow[0][0])


def insert_orange():
    label['text'] = rainbow[1][1]
    entry.delete(0, END)
    entry.insert(0, rainbow[1][0])


def insert_yellow():
    label['text'] = rainbow[2][1]
    entry.delete(0, END)
    entry.insert(0, rainbow[2][0])
```

```
def insert_green():
    label['text'] = rainbow[3][1]
    entry.delete(0, END)
    entry.insert(0, rainbow[3][0])


def insert_lightblue():
    label['text'] = rainbow[4][1]
    entry.delete(0, END)
    entry.insert(0, rainbow[4][0])


def insert_blue():
    label['text'] = rainbow[5][1]
    entry.delete(0, END)
    entry.insert(0, rainbow[5][0])


def insert_violet():
    label['text'] = rainbow[6][1]
    entry.delete(0, END)
    entry.insert(0, rainbow[6][0])


label = Label(width=10)
label.pack()
entry = Entry(width=15, justify=CENTER)
entry.pack()
Button(width=12, bg=rainbow[0][0], command=insert_red).pack()
Button(width=12, bg=rainbow[1][0], command=insert_orange).pack()
Button(width=12, bg=rainbow[2][0], command=insert_yellow).pack()
Button(width=12, bg=rainbow[3][0], command=insert_green).pack()
Button(width=12, bg=rainbow[4][0], command=insert_lightblue).pack()
Button(width=12, bg=rainbow[5][0], command=insert_blue).pack()
Button(width=12, bg=rainbow[6][0], command=insert_violet).pack()
root.mainloop()
```

Поскольку код функций почти идентичен, будет уместным создать для кнопок общий класс, в конструктор которого передавать индекс элемента кортежа или сами цвета.

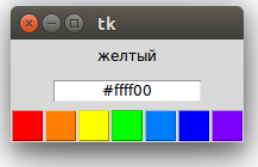
```
from tkinter import *

root = Tk()

rainbow = (('#ff0000', 'красный'), ('#ff7d00', 'оранжевый'),
           ('#ffff00', 'желтый'), ('#00ff00', 'зеленый'),
           ('#007dff', 'голубой'), ('#0000ff', 'синий'),
           ('#7d00ff', 'фиолетовый'))


class ColorButton:
    def __init__(self, i):
        self.i = i
        self.button = Button(width=12, bg=rainbow[i][0],
                             command=self.insert_color)
        self.button.pack()
```

```
    def insert_color(self):
        label['text'] = rainbow[self.i][1]
        entry.delete(0, END)
        entry.insert(0, rainbow[self.i][0])


label = Label(width=10)
label.pack()
entry = Entry(width=15, justify=CENTER)
entry.pack()
for i in range(len(rainbow)):
    ColorButton(i)

root.mainloop()
```

У каждого объекта типа *ColorButton* есть свое свойство *i* и своя кнопка. Метка и текстовое поле остаются "общими", они не являются атрибутами экземпляров *ColorButton*. Поэтому обращение к ним происходит без селфа.

3. Метод pack

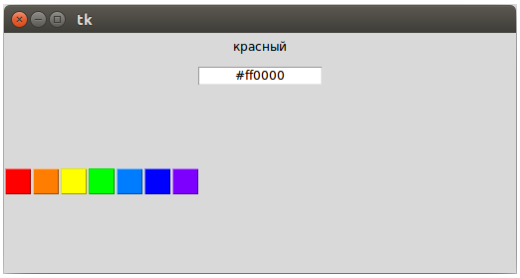
Перепишите программу из практической работы предыдущего урока так, чтобы интерфейс выглядел примерно следующим образом:



Необходимые изменения по-сравнению с предыдущей версией минимальны:

```
from tkinter import *

root = Tk()

rainbow = (('#ff0000', 'красный'), ('#ff7d00', 'оранжевый'),
           ('#ffff00', 'желтый'), ('#00ff00', 'зеленый'),
           ('#007dff', 'голубой'), ('#0000ff', 'синий'),
           ('#7d00ff', 'фиолетовый'))


class ColorButton:
    def __init__(self, master, i):
        self.i = i
        self.button = Button(master, bg=rainbow[i][0],
                             command=self.insert_color)
        self.button.pack(side=LEFT)

    def insert_color(self):
        label['text'] = rainbow[self.i][1]
        entry.delete(0, END)
        entry.insert(0, rainbow[self.i][0])


label = Label(width=10)
label.pack(pady=5)
entry = Entry(width=15, justify=CENTER)
entry.pack(pady=5)
frame = Frame()
frame.pack()
for i in range(len(rainbow)):
    ColorButton(frame, i)

root.mainloop()
```

Кнопки должны прибиваться к левой или правой стороне. Все кнопки лучше положить во фрейм, так как без него при раскрытии окна будет происходить искажение интерфейса:


tk
красный
#ff0000

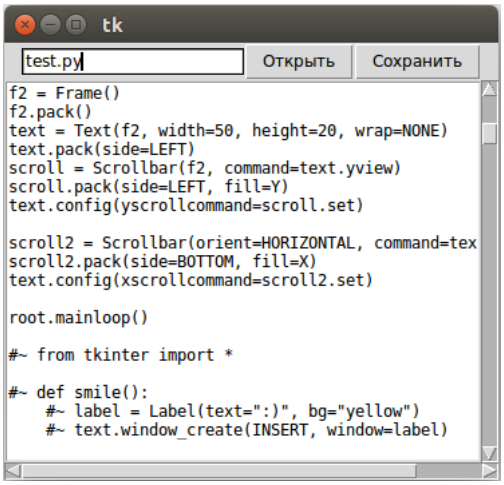
4. Text – многострочное текстовое поле

Напишите программу, состоящую из однострочного и многострочного текстовых полей и двух кнопок "Открыть" и "Сохранить". При клике на первую должен открываться на чтение файл, чье имя указано в поле класса **`Entry`**, а содержимое файла должно загружаться в поле типа **`Text`**.

При клике на вторую кнопку текст, введенный пользователем в экземпляр **`Text`**, должен сохраняться в файле под именем, которое пользователь указал в однострочном текстовом поле.

Файлы будут читаться и записываться в том же каталоге, что и файл скрипта, если указывать имена файлов без адреса.

Для выполнения практической работы вам понадобится функция **`open`** языка Python и методы файловых объектов чтения и записи. Освежить знания о них можно здесь.




```
from tkinter import *


def file_load():
    f = open(entry.get())
    text.delete(1.0, END)
    text.insert(1.0, f.read())


def file_save():
    f = open(entry.get(), 'x')
    f.write(text.get(1.0, END))
    text.delete(1.0, END)


root = Tk()

f1 = Frame()
f1.pack()
entry = Entry(f1, width=20)
entry.pack(side=LEFT)
Button(f1, text="Открыть", command=file_load)\
    .pack(side=LEFT)
Button(f1, text="Сохранить", command=file_save)\
```

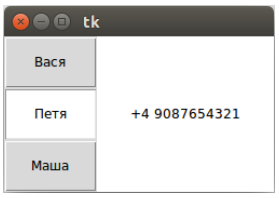
```
    .pack(side=LEFT)

f2 = Frame()
f2.pack()
text = Text(f2, width=50, height=20, wrap=NONE)
text.pack(side=LEFT)
scroll = Scrollbar(f2, command=text.yview)
scroll.pack(side=LEFT, fill=Y)
text.config(yscrollcommand=scroll.set)

scroll2 = Scrollbar(orient=HORIZONTAL, command=text.xview)
scroll2.pack(side=BOTTOM, fill=X)
text.config(xscrollcommand=scroll2.set)

root.mainloop()
```

Если планируется использовать горизонтальную прокрутку, то для текстового поля необходимо свойство **`wrap`** установить в значение **`NONE`**, т. е. "не переносить". Иначе текст будет сам переносится при достижении конца строки, и смысла в горизонтальной прокрутке не будет (скроллер останется неактивным).

5. Radiobutton и Checkbutton. Переменные Tkinter

Виджеты Radiobatton и Checkbutton поддерживают большинство свойств оформления внешнего вида, которые есть у других элементов графического интерфейса. При этом у Radiobutton есть особое свойство **`indicatoron`**. По-умолчанию он равен единице, в этом случае радиокнопка выглядит как нормальная радиокнопка. Однако если присвоить этой опции ноль, то виджет Radiobutton становится похожим на обычную кнопку по внешнему виду. Но не по смыслу.

Напишите программу, в которой имеется несколько объединенных в группу радиокнопок, индикатор которых выключен (`indicatoron=0`). Если какая-нибудь кнопка включается, то в метке должна отображаться соответствующая ей информация. Обычных кнопок в окне быть не должно.



Помните, что свойство **`command`** есть не только у виджетов класса **`Button`**.

```
from tkinter import *


def get_phone():
    v = var.get()
    if v == 0:
        label['text'] = '+7 1234567890'
    elif v == 1:
        label['text'] = '+4 9087654321'
    elif v == 2:
        label['text'] = '+9 1212121212'


root = Tk()

f = Frame()
f.pack(side=LEFT)

var = IntVar()
var.set(-1)
Radiobutton(f, text="Вася", width=10, height=3,
            indicatoron=0, variable=var,
            value=0, command=get_phone).pack()
Radiobutton(f, text="Петя", width=10, height=3,
            indicatoron=0, variable=var,
            value=1, command=get_phone).pack()
Radiobutton(f, text="Маша", width=10, height=3,
            indicatoron=0, variable=var,
            value=2, command=get_phone).pack()

label = Label(bg="white", width=20)
label.pack(side=LEFT, fill=Y)

root.mainloop()
```

Поскольку радиокнопки однотипные, может иметь смысл создать класс (представьте, что у вас не три персоны, а сто):

```
from tkinter import *

db = {'Вася': '+7 1234567890',
      'Петя': '+4 9087654321',
      'Маша': '+9 1212121212'}


class RadioIndicatorOff:
    __val = 0

    def __init__(self, master, person):
        RadioIndicatorOff.__val += 1
        self.name = person
        self.rb = Radiobutton(master, text=person, width=10,
                              height=3, indicatoron=0,
                              variable=var,
                              value=RadioIndicatorOff.__val,
                              command=self.get_phone)
        self.rb.pack()

    def get_phone(self):
        label['text'] = db[self.name]


root = Tk()

f = Frame()
f.pack(side=LEFT)

var = IntVar()
var.set(-1)
for key in db.keys():
    RadioIndicatorOff(f, key)

label = Label(bg="white", width=20)
label.pack(side=LEFT, fill=Y)

root.mainloop()
```

6. Виджет Listbox

Напишите программу, состоящую из двух списков Listbox. В первом будет, например, перечень товаров, заданный программно. Второй изначально пуст, пусть это будет перечень покупок. При клике на одну кнопку товар должен переходить из одного списка в другой. При клике на вторую кнопку – возвращаться (человек передумал покупать). Предусмотрите возможность множественного выбора элементов списка и их перемещения.

```
from tkinter import *


def buy(first, second):
    select = first.curselection()
    for i in select:
        second.insert(END, first.get(i))
    select = list(select)
    select.reverse()
    for i in select:
        first.delete(i)


root = Tk()

food = ['apple', 'bananas', 'tomato', 'potato', 'carrot',
        'pineapple', 'bread', 'butter', 'milk', 'meat']

shop = Listbox(selectmode=EXTENDED)
shop.pack(side=LEFT)
for i in food:
    shop.insert(END, i)
goods = Listbox(selectmode=EXTENDED)
goods.pack(side=RIGHT)

f = Frame()
f.pack(side=LEFT)
Button(f, text=">>>", command=lambda
    i=shop, j=goods: buy(i, j)).pack()
Button(f, text="<<<", command=lambda
    i=goods, j=shop: buy(i, j)).pack()

root.mainloop()
```

7. Метод bind

Напишите программу по следующему описанию. Нажатие `Enter` в однострочном текстовом поле приводит к перемещению текста из него в список (экземпляр **`Listbox`**). При двойном клике (**`<Double-Button-1>`**) по элементу-строке списка, она должна копироваться в текстовое поле.

```
from tkinter import *

root = Tk()


def take_item(event):
    s = e.get()
    e.delete(0, END)
    l.insert(END, s)


def give_item(event):
    tup = l.curselection()
    if tup:
        s = l.get(tup[0])
        e.delete(0, END)
        e.insert(0, s)


e = Entry()
e.pack()
e.bind('<Return>', take_item)
l = Listbox()
l.pack()
l.bind('<Double-Button-1>', give_item)

root.mainloop()
```

Необходимость использовать двойной клик в основном обусловлена тем, что при одинарном сначала происходит событие (клик), и только после этого выделение элемента. Таким образом, функция *give_item* запускается раньше, чем элемент выделяется. В ней метод **`curselection`** возвращает либо пустой кортеж (если перед этим никакой элемент не выделялся), либо индекс предыдущего выделенного элемента. В результате чего программа либо генерирует исключение, либо работает неправильно.

Двойной клик решает проблему, так как функция вызывается, когда необходимый элемент уже однозначно выделен.

В функции *give_item* кортеж, который возвращает **`curselection`** может быть пустым, если никакой элемент списка выделен не был. Попытка взять из пустого кортежа элемент по индексу приведет к выбросу исключения, поэтому мы проверяем *tup* на пустоту с помощью **`if`**.

В приведенной программе в список можно добавить пустую строку, если в поле ничего не было введено, но был нажат `Enter`. Можно было бы реализовать и обработку этого случая.

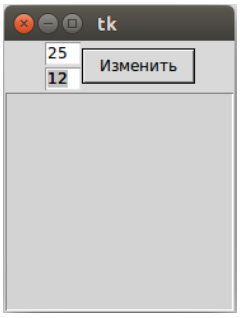
8. События

Напишите программу по описанию. Размеры многострочного текстового поля определяются значениями, введенными в однострочные текстовые поля. Изменение размера происходит при нажатии мышью на кнопку, а также при нажатии клавиши `Enter`.

Цвет фона экземпляра **`Text`** светлосерый (**`lightgrey`**), когда поле не в фокусе, и белый, когда имеет фокус.

Событие получения фокуса обозначается как **`<FocusIn>`**, потери – как **`<FocusOut>`**.

Для справки: фокус перемещается по виджетам при нажатии `Tab`, `Ctrl+Tab`, `Shift+Tab`, а также при клике по ним мышью (к кнопкам последнее не относится).




```
from tkinter import *
root = Tk()


def change_side(event):
    if ew.get().isdigit():
        t['width'] = ew.get()
    if eh.get().isdigit():
        t['height'] = eh.get()


def white_color(event):
    t['bg'] = 'white'


def grey_color(event):
    t['bg'] = 'lightgrey'


f1 = Frame()
f1.pack()

f = Frame(f1)
f.pack(side=LEFT)
ew = Entry(f, width=3)
ew.pack()
ew.focus_set()
eh = Entry(f, width=3)
eh.pack()
```

```
b = Button(f1, text="Изменить")
b.pack(side=LEFT)
b.bind('<Button-1>', change_side)
b.bind('<Return>', change_side)
ew.bind('<Return>', change_side)
eh.bind('<Return>', change_side)

f2 = Frame()
f2.pack()

t = Text(f2, width=20, height=10, bg='lightgrey')
t.pack()
t.bind('<FocusIn>', white_color)
t.bind('<FocusOut>', grey_color)

root.mainloop()
```

Поскольку логика функций *white_color* и *grey_color* идентична, их можно заменить одной с дополнительным параметром, а вызов функций оформить через лямбду:

```
…
def bg_color(event, color):
    t['bg'] = color
…
t.bind('<FocusIn>',
       lambda event, i='white': bg_color(event, i))
t.bind('<FocusOut>',
       lambda event, i='lightgrey': bg_color(event, i))
…
```

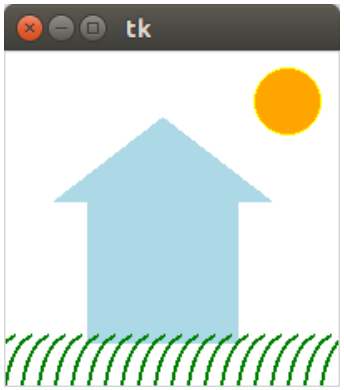
С другой стороны, тело *bg_color* состоит из одного выражения. Следовательно, его можно непосредственно поместить в тело лямбда-функции:

```
…
t.bind('<FocusIn>',
       lambda event, i='white': t.config(bg=i))
t.bind('<FocusOut>',
       lambda event, i='lightgrey': t.config(bg=i))
…
```

При этом в теле лямбды нельзя выполнить присваивание, поэтому меняем настройки виджета через метод **`config`**.

9. Canvas

Создайте на холсте подобное изображение:



Для создания травы используется цикл.

```
from tkinter import *
root = Tk()

c = Canvas(root, width=200, height=200, bg='white')
c.pack()

c.create_rectangle(50, 90, 140, 175,
            fill='lightblue', outline='lightblue')
c.create_polygon(30, 90, 160, 90, 95, 40,
            fill='lightblue', outline='lightblue')
c.create_oval(150, 10, 190, 50,
            fill='orange', outline='yellow')

x = -20
while x < 200:
    c.create_arc(x, 170, x+40, 250, outline='green',
            style=ARC, start=180, extent=-80,
            width=2)
    x += 10

root.mainloop()
```

10. Canvas. Идентификаторы, теги и анимация

В данной программе создается анимация круга, который движется от левой границы холста до правой:

```
from tkinter import *


def motion():
    c.move(ball, 1, 0)
    if c.coords(ball)[2] < 300:
        root.after(10, motion)


root = Tk()
c = Canvas(root, width=300, height=200,
           bg="white")
c.pack()
ball = c.create_oval(0, 100, 40, 140,
                     fill='green')
motion()
root.mainloop()
```

Выражение `c.coords(ball)` возвращает список текущих координат объекта (в данном случае это *ball*). Третий элемент списка соответствует его второй координате *x*.

Метод **`after`** вызывает функцию, переданную вторым аргументом, через количество миллисекунд, указанных первым аргументом.

Изучите приведенную программу и самостоятельно запрограммируйте постепенное движение фигуры в ту точку холста, где пользователь кликает левой кнопкой мыши. Координаты события хранятся в его атрибутах **`x`** и **`y`** (`event.x`, `event.y`).

В приведенной ниже программе реализована возможность множественных кликов и движения фигуры в разных направлениях.

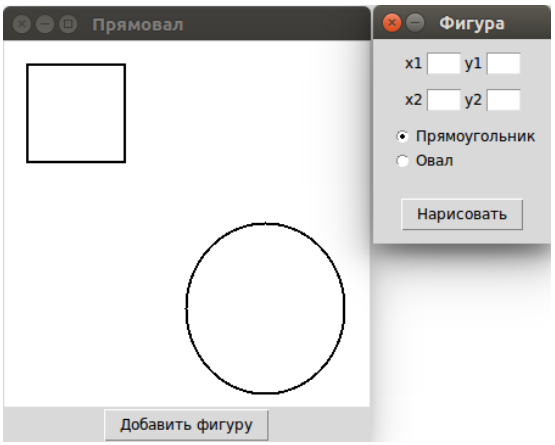
```
from tkinter import *

root = Tk()
c = Canvas(root, width=300, height=200, bg="white")
c.pack()

ball = c.create_oval(0, 100, 40, 140, fill='green')

def motion(x_new, x_step, y_step, flag=1):
    c.unbind('<Button-1>')
    c.itemconfig(ball, fill='green2')
    c.move(ball, x_step, y_step)
    if flag == 1:
        if c.coords(ball)[0] + 20 < x_new:
            root.after(10, lambda: motion(x_new, x_step, y_step))
        else:
            c.bind('<Button-1>', click)
            c.itemconfig(ball, fill='green')
```

```
    else:
        if c.coords(ball)[0] + 20 > x_new:
            root.after(10, lambda: motion(x_new, x_step, y_step, -1))
        else:
            c.bind('<Button-1>', click)
            c.itemconfig(ball, fill='green')

def click(event):
    p = c.coords(ball)
    x1 = p[0]
    y1 = p[1]
    x_ball = x1 + 20 # координаты центра шара
    y_ball = y1 + 20
    dx = abs(event.x - x_ball) # расстояние, которое надо пройти ...
    dy = abs(event.y - y_ball) # ... по осям X и Y
    if dx > dy: # Если расстояние по X больше, чем по Y,
        x_step = 1
        y_step = dy / dx # то шаг по Y должен быть меньше
    else: # Иначе
        x_step = dx / dy # должен быть меньше шаг по X
        y_step = 1
    if event.x < x_ball: # Если двигаться назад по X
        x_step *= -1
    if event.y < y_ball: # Если двигаться назад по Y
        y_step *= -1
    if event.x > x_ball:
        motion(event.x, x_step, y_step, 1)
    else:
        motion(event.x, x_step, y_step, -1)

c.bind('<Button-1>', click)

root.mainloop()
```

Метод **`unbind`** отвязывает событие. Если этого не сделать, то появляется баг, когда совершается клик при уже движущемся шаре. Хотя шар начинает двигаться в новом направлении, в отдельных случаях уже не может остановиться. Видимо это происходит из-за того, что новый клик запускает новый цикл рекурсивных вызовов *motion*, в то время как старый продолжает выполняться. При этом данные в последнем оказываются измененными, и выражение при **`if`** всегда возвращает **`True`**. Вероятно, данная проблема может быть решена через использование глобальных переменных.

11. Окна

Напишите программу, в которой на главном окне находятся холст и кнопка "Добавить фигуру". Кнопка открывает второе окно, включающее четыре поля для ввода координат и две радиокнопки для выбора, рисовать ли на холсте прямоугольник или овал. Здесь же находится кнопка "Нарисовать", при клике на которую соответствующая фигура добавляется на холст, а второе окно закрывается. Проверку корректности ввода в поля можно опустить.



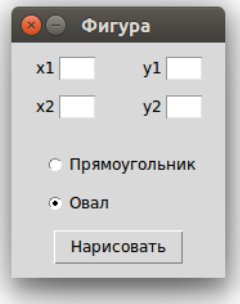

```
from tkinter import *
root = Tk()
root.title('Прямовал')

def addFigure():
    fig = Toplevel()
    fig.title('Фигура')
    fig.resizable(0, 0)

    frame1 = Frame(fig)
    frame1.pack(padx=10, pady=10)
    Label(frame1, text='x1').pack(side=LEFT)
    x1 = Entry(frame1, width=3)
    x1.pack(side=LEFT)
    Label(frame1, text='y1').pack(side=LEFT)
    y1 = Entry(frame1, width=3)
    y1.pack(side=LEFT)

    frame2 = Frame(fig)
    frame2.pack()
    Label(frame2, text='x2').pack(side=LEFT)
    x2 = Entry(frame2, width=3)
    x2.pack(side=LEFT)
    Label(frame2, text='y2').pack(side=LEFT)
    y2 = Entry(frame2, width=3)
    y2.pack(side=LEFT)

    frame3 = Frame(fig)
    frame3.pack(padx=10, pady=10)
    v = IntVar()
    v.set(1)
    Radiobutton(frame3, text='Прямоугольник', variable=v,
```

```
                value=1).pack(anchor=W)
    Radiobutton(frame3, text='Овал', variable=v,
                value=0).pack(anchor=W)

    def paint():
        x = int(x1.get())
        y = int(y1.get())
        xx = int(x2.get())
        yy = int(y2.get())
        if v.get() == 0:
            c.create_oval(x, y, xx, yy, width=2)
        elif v.get() == 1:
            c.create_rectangle(x, y, xx, yy, width=2)
        fig.destroy()
    Button(fig, text='Нарисовать', command=paint).pack(pady=10)

c = Canvas(width=300, height=300, bg='white')
c.pack()
Button(text='Добавить фигуру', command=addFigure).pack()

root.mainloop()
```

При таком варианте решения мы вынуждены функцию *paint* оставить внутри функции *addFigure*, иначе при выходе из последней переменные *x1-y2* окажутся недоступными.

12. Метод grid

Перепрограммируйте второе окно из практической работы предыдущего урока, используя метод **`grid`**.




```
from tkinter import *
root = Tk()
root.title('Прямовал')

def addFigure():
    fig = Toplevel()
    fig.title('Фигура')
    fig.resizable(0, 0)

    Label(fig, text='x1').grid(row=0, column=0, sticky=E)
    x1 = Entry(fig, width=3)
    x1.grid(row=0, column=1, pady=10, sticky=W)
    Label(fig, text='y1').grid(row=0, column=2, sticky=E)
    y1 = Entry(fig, width=3)
    y1.grid(row=0, column=3, sticky=W)

    Label(fig, text='x2').grid(row=1, column=0, sticky=E)
    x2 = Entry(fig, width=3)
    x2.grid(row=1, column=1, sticky=W)
    Label(fig, text='y2').grid(row=1, column=2, sticky=E)
    y2 = Entry(fig, width=3)
    y2.grid(row=1, column=3, sticky=W)

    Frame(fig, height=15).grid(row=2, column=0)

    v = IntVar()
    v.set(1)
    r1 = Radiobutton(fig, text='Прямоугольник', variable=v, value=1)
    r1.grid(row=3, column=0, columnspan=4, pady=10, padx=20)
    r2 = Radiobutton(fig, text='Овал', variable=v, value=0)
    r2.grid(row=4, column=0, columnspan=4, sticky=W, padx=20)

    def paint():
        x = int(x1.get())
        y = int(y1.get())
        xx = int(x2.get())
```

```
        yy = int(y2.get())
        if v.get() == 0:
            c.create_oval(x, y, xx, yy, width=2)
        elif v.get() == 1:
            c.create_rectangle(x, y, xx, yy, width=2)
        fig.destroy()
    button_draw = Button(fig, text='Нарисовать', command=paint)
    button_draw.grid(row=5, column=0, columnspan=4, pady=10)

c = Canvas(width=300, height=300, bg='white')
c.pack()
Button(text='Добавить фигуру', command=addFigure).pack()

root.mainloop()
```

Опция **`pady`** (как и **`padx`**) делает отступ с обеих сторон, в данном случае сверху и снизу. Бывает, что нужен отступ только с одной стороны. Здесь между второй строкой, где задаются вторые координаты (x_2; y_2), и третьей, в которой расположена радиокнопка "Прямоугольник", должен быть отступ больше, чем между соседними строками.

Однако, если установить достаточно большое значение **`pady`** у радиокнопки или у одной из координат, то также друг от друга далеко раздвинутся либо радиокнопки, либо координаты, чего быть не должно. В таких случаях приходится использовать "костыль" в виде фрейма или метки определенной высоты или ширины.

13. Диалоговые окна

В приведенной в уроке программе с функциями **`askopenfilename`** и **`asksaveasfilename`** генерируются исключения, если диалоговые окна были закрыты без выбора или указания имени файлов.

Напишите код обработки данных исключений. При этом для пользователя должно появляться информационное диалоговое окно с сообщением о том, что файл не загружен или не сохранен.

Добавьте кнопку "Очистить", которая удаляет текст из поля. Перед удалением пользователь должен подтвердить свои намерения через соответствующее диалоговое окно.

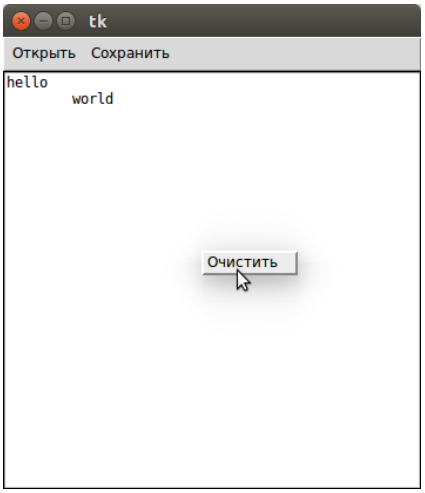
Обратите внимание, исключения выбрасываются на уровне функции **`open`** Питона, но не функций модуля **`filedialog`**, которые просто возвращают в программу строку или кортеж.

```
from tkinter import *
from tkinter import filedialog as fd
from tkinter import messagebox as mb


def del_text():
    answer = mb.askokcancel('Удаление текста',
                            'Реально удалить?')
    if answer:
        text.delete(1.0, END)


def insert_text():
    file_name = fd.askopenfilename()
    try:
        f = open(file_name)
    except (FileNotFoundError, TypeError):
        mb.showinfo("Открытие файла",
                    "Файл не выбран!")
    else:
        s = f.read()
        text.insert(1.0, s)
        f.close()


def extract_text():
    file_name = fd.asksaveasfilename()
    try:
        f = open(file_name, 'w')
    except (FileNotFoundError, TypeError):
        mb.showinfo("Сохранение файла",
                    "Файл не сохранен!")
    else:
        s = text.get(1.0, END)
        f.write(s)
        f.close()


root = Tk()
b3 = Button(text="Очистить", command=del_text)
```

```
b3.grid(column=1, sticky=E)
text = Text(width=50, height=25)
text.grid(row=1, columnspan=2)
b1 = Button(text="Открыть", command=insert_text)
b1.grid(row=2, sticky=E)
b2 = Button(text="Сохранить", command=extract_text)
b2.grid(row=2, column=1, sticky=W)

root.mainloop()
```

14. Виджет Menu

Измените программу из практической работы предыдущего урока так, чтобы открытие и сохранение файлов выполнялось не через экземпляры **`Button`**, а через **`Menu`**. Команду очистки текстового поля поместите в контекстное меню.



```
from tkinter import *
from tkinter import filedialog as fd
from tkinter import messagebox as mb


def del_text():
    answer = mb.askokcancel('Удаление текста', 'Реально удалить?')
    if answer:
        text.delete(1.0, END)


def insert_text():
    file_name = fd.askopenfilename()
    try:
        f = open(file_name)
    except (FileNotFoundError, TypeError):
        mb.showinfo("Открытие файла", "Файл не выбран!")
    else:
        s = f.read()
        text.insert(1.0, s)
        f.close()


def extract_text():
    file_name = fd.asksaveasfilename()
    try:
        f = open(file_name, 'w')
    except (FileNotFoundError, TypeError):
        mb.showinfo("Сохранение файла", "Файл не сохранен!")
    else:
        s = text.get(1.0, END)
        f.write(s)
        f.close()
```

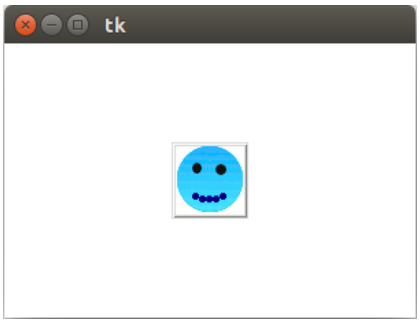
```
root = Tk()

mainmenu = Menu()
root['menu'] = mainmenu
mainmenu.add_command(label="Открыть", command=insert_text)
mainmenu.add_command(label="Сохранить", command=extract_text)

text = Text(width=50, height=25)
text.pack()

popupmenu = Menu(text, tearoff=0)
popupmenu.add_command(label="Очистить", command=del_text)

text.bind('<Button-3>',
          lambda event: popupmenu.post(event.x_root, event.y_root))

root.mainloop()
```

15. Метод place

За рамками данного курса было оставлено несколько классов пакета **tkinter**. Среди них **PhotoImage**, позволяющий использовать в программе внешние изображения форматов GIF и PGM. Экземпляры **PhotoImage** можно размещать на различных виджетах через опцию **image**.

```
…
img = PhotoImage(file='smile.gif')
l=Label(image=img)
…
```

Напишите программу, состоящую из главного окна и кнопки, на которой изображен смайлик. При клике на кнопку она должна оказываться в новом случайном месте окна. Размер окна может меняться.

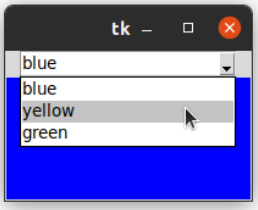



```
from tkinter import *
from random import random

def move():
    b.place(relx=random(), rely=random())

root = Tk()
root['bg'] = 'white'
root.geometry('300x200')
img = PhotoImage(file='smile.gif')
b = Button(image=img, command=move)
b.place(relx=0.5, rely=0.5, anchor=CENTER)

root.mainloop()
```

Задача решается так просто, благодаря опциям **relx** и **rely** метода **place**, с помощью которых устанавливается позиция кнопки в относительных единицах, то есть долях единицы, где за единицу принимается текущий размер окна по той или иной оси. Со своей стороны функция **random** возвращает случайное вещественное число от 0 до 1, которое также можно рассматривать как долю от единицы.

16. Модуль tkinter.ttk

Напишите программу, состоящую из выпадающего списка (**`Combobox`**) и холста. В списке перечисляются цвета. При выборе в списке того или иного значения цвет холста должен изменяться на соответствующий.



Для справки:

- Список значений передается в **`Combobox`** через опцию **`values`**.
- Метод **`current`** экземпляра **`Combobox`** позволяет указать значение, которое будет выбрано в списке изначально. Метод принимает индекс элемента из списка значений.
- Событие смены значения в выпадающем списке – `'<<ComboboxSelected>>'`
- Метод **`get`** экземпляра **`Combobox`** возвращает выбранный на данный момент элемент списка.

```
from tkinter import *
from tkinter.ttk import *

colors = ['blue', 'yellow', 'green']

root = Tk()
combo = Combobox(values=colors)
combo.current(0)
canvas = Canvas(width=200, height=100,
                bg=colors[0])

combo.bind('<<ComboboxSelected>>',
           lambda event:
           canvas.config(bg=combo.get())
           )
combo.pack()
canvas.pack()
root.mainloop()
```