МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
"Юго-Западный государственный университет"
(ЮЗГУ)

Кафедра биомедицинской инженерии

УТВЕРЖДАЮ
Проректор по учебной работе
Локтионова О.Г.
«05» 03 2021 г.

ЯЗЫК PYTHON

Методические указания к самостоятельной работе студентов
направления подготовки 30.05.03 - "Медицинская кибернетика"

Курск 2021

УДК 004.432

Составитель: Л.В. Стародубцева

Рецензент
Доктор технических наук, профессор *И.Е. Чернецкая*

Язык Python: методические указания к самостоятельной работе / Юго-Зап. гос. ун-т; сост.: Л.В. Стародубцева. - Курск, 2021. - 48 с.

Содержатся сведения, необходимые для выполнения самостоятельных работ по языку Python.

Методические указания по структуре, содержанию и стилю изложения материала соответствуют методическим и научным требованиям, предъявляемым к учебным и методическим пособиям.

Предназначены для студентов направления подготовки 30.05.03 очной и заочной форм обучений.

Текст печатается в авторской редакции

Подписано в печать 25.03.21 Формат 60x84 1/16. Бумага офсетная. Усл. печ.л. 2,79 . Уч. -изд.л. 2,53 Тираж 100 экз. Заказ. Бесплатно.
Юго-Западный государственный университет
305040, г.Курск, ул. 50 лет Октября, 94

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.

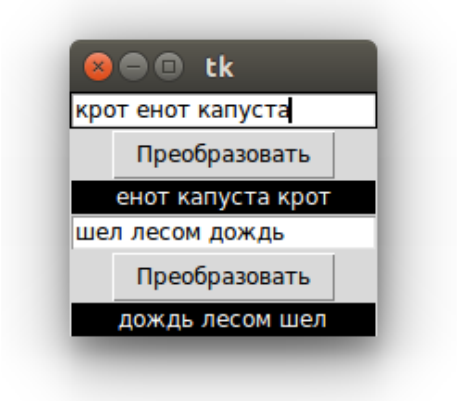
Тогда можно передавать значения для свойства command в конструктор. Значение будет представлять собой привязываемую к кнопке функцию-обработчик события. Полный код программы:

```
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.

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.

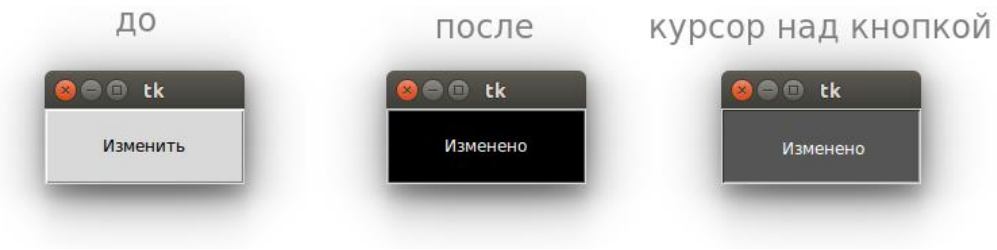
По умолчанию размер кнопки соответствует ширине и высоте текста, однако с помощью свойств **width** и **height** эти параметры можно изменить. Единицами измерения в данном случае являются знакоместа.

Такие свойства как **bg**, **fg**, **activebackground** и **activeforegroun** 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**.

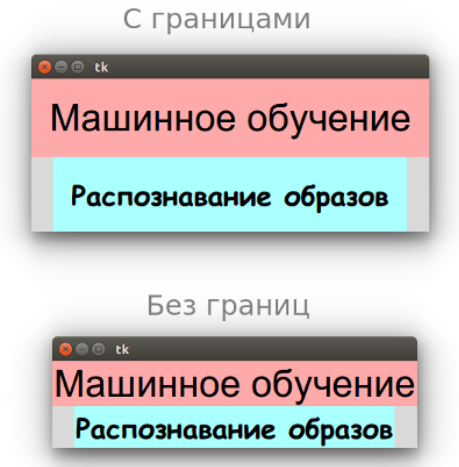
На примере объекта типа **Label** рассмотрим свойство **font** – шрифт.

```
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.

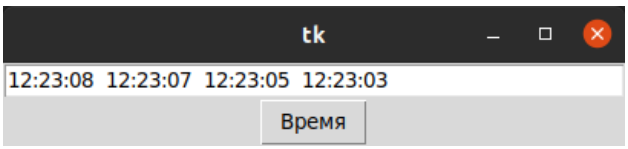
Свойства экземпляров **Entry** во многом схожи с двумя предыдущими виджетами. А вот методы – нет. Из текстового поля можно взять текст. За это действие отвечает метод **get**. В текстовое поле можно вставить текст методом **insert**. Также можно удалить текст методом **delete**.

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**.

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)
```

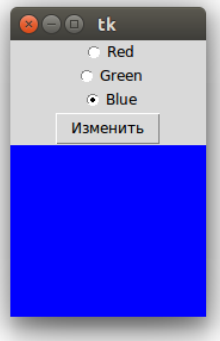
Здесь переменной *r_var* присваивается объект типа **BooleanVar**. С помощью метода **set** он устанавливается в значение 0.

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()
```

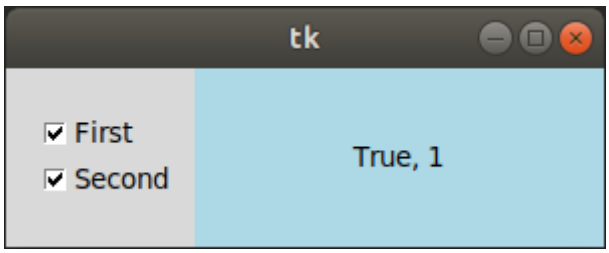
В двух последних вариациях кода мы не используем метод **get**, чтобы получить значение переменной *var*. В данном случае нам это не требуется, потому что цвет метки меняется в момент клика по соответствующей радиокнопке и не находится в зависимости от значения переменной. Несмотря на это использовать 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 **Checkbutto** 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()
```



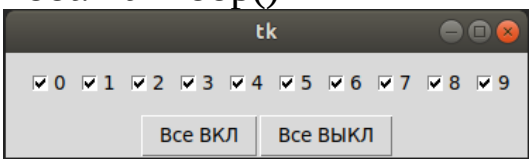
С помощью опции **onvalue** устанавливается значение, которое принимает связанная переменная при включенном флажке. С помощью свойства **offvalue** – при выключенном. В данном случае оба флажка при запуске программы будут выключены, так как методом **set** были установлены отличные от **onvalue** значения.

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()
```

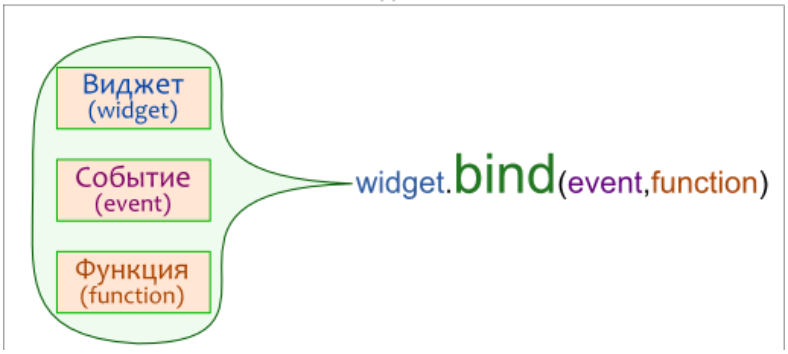
```
f1.pack(padx=10, pady=10)
checks = []
for i in range(10):
    checks.append(CheckButton(f1, i))

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()
```



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.

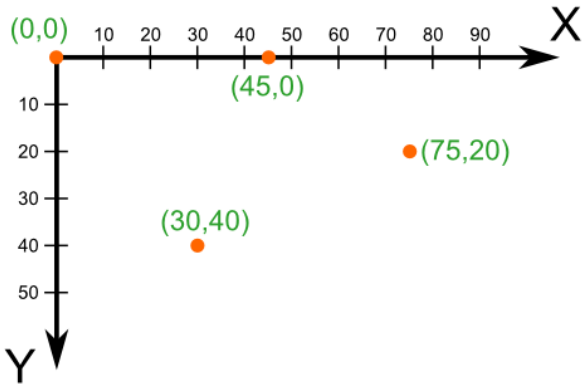
У меток нет **command**, однако это свойство есть у кнопок:

```
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()
```

Canvas

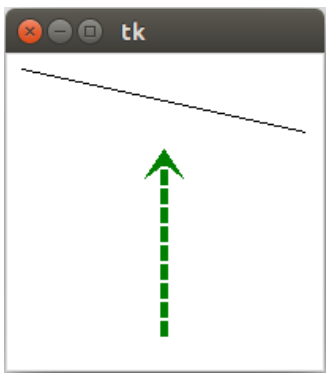
В **tkinter** от класса **Canvas** создаются объекты-холсты, на которых можно "рисовать", размещая различные фигуры и объекты. Делается это с помощью вызовов соответствующих методов.

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).

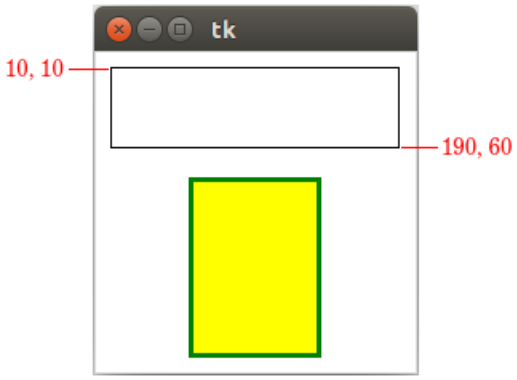
```
from tkinter import *
root = Tk()

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**:

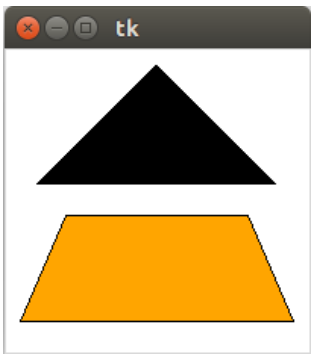
```
…
c.create_rectangle(10, 10, 190, 60)

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')
…
```

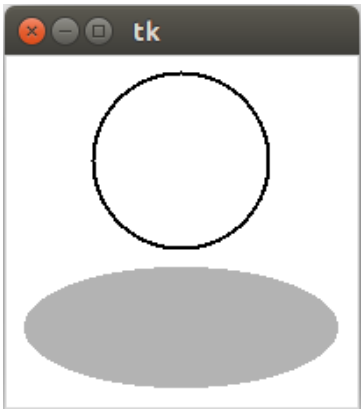


Для удобства координаты точек можно заключать в скобки:

```
…
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.

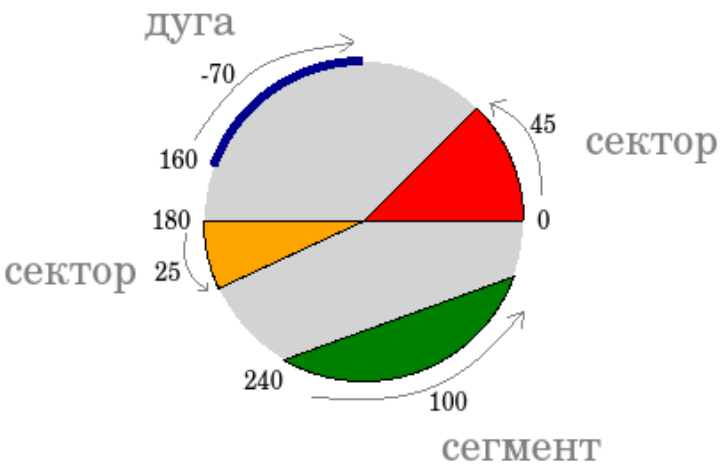
```
…
c.create_oval(50, 10, 150, 110, width=2)
c.create_oval(10, 120, 190, 190,
              fill='grey70', outline='white')
…
```



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)
…
```

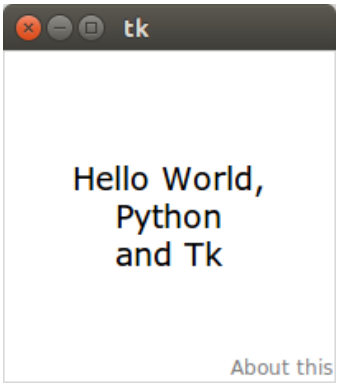


В данном примере светло-серый круг используется исключительно для наглядности.

На холсте можно разместить текст. Делается это с помощью метод **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.

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))
…
```

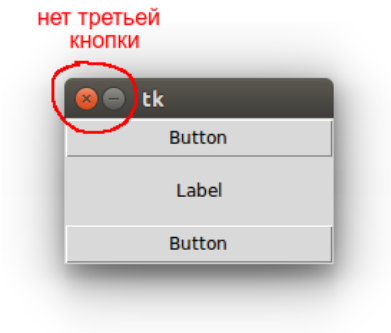
Здесь мы вычитаем половину ширины и высоты окна (по 200 пикселей). Иначе в центре экрана окажется верхний левый угол окна, а не его середина.

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.

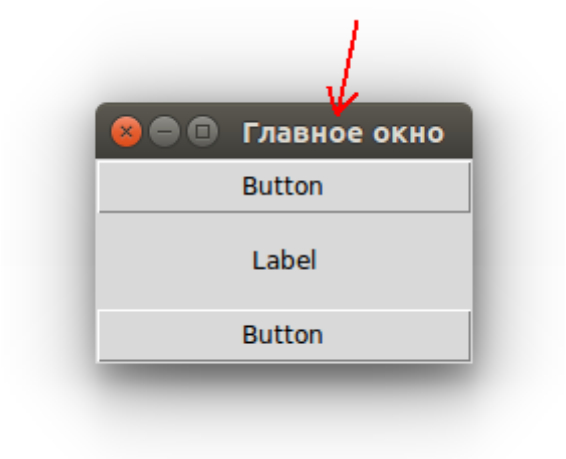
По умолчанию пользователь может разворачивать окно на весь экран, а также изменять его размер, раздвигая границы. Эти возможности можно отключить с помощью метода **resizable**. Так root.resizable(False, False) запретит изменение размеров главного окна как по горизонтали, так и вертикали. Развернуть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

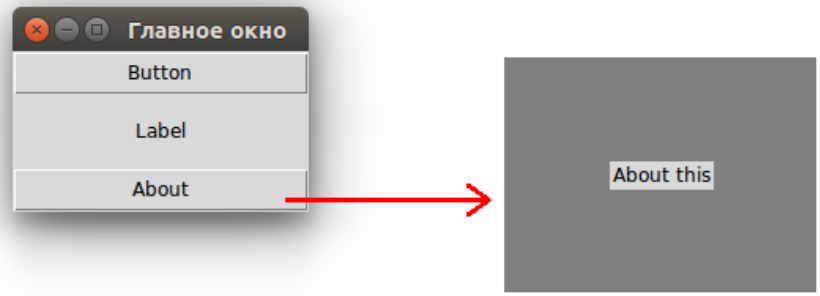
```
…
root.title("Главное окно")
…
```



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()
```



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.

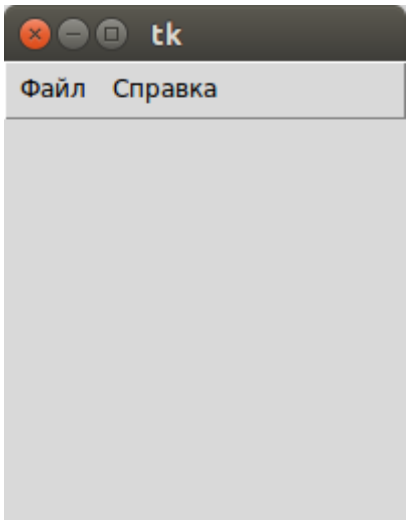
В **tkinter** экземпляр меню создается от класс **Menu**, далее его надо привязать к виджету, на котором оно будет расположено. Обычно таковым выступает главное окно приложения. Его опции **menu** присваивается экземпляр **Menu** через имя связанной с экземпляром переменной.

```
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')
…
```

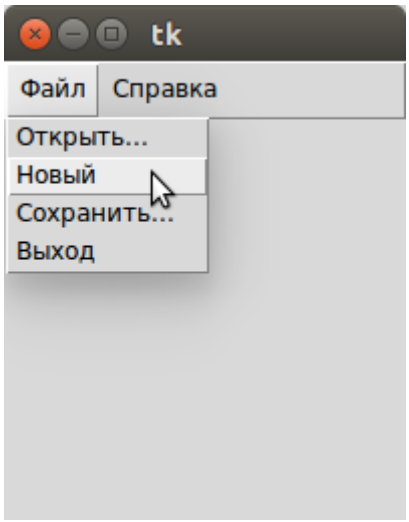


В данном случае "Файл" и "Справка" – это команды. К ним можно добавить опцию **command**, связав тем самым с какой-либо функцией-обработчиком клика. Хотя такой вариант меню имеет право на существование, в большинстве приложений панель меню содержит выпадающие списки команд, а сами пункты на панели командами, 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()
```

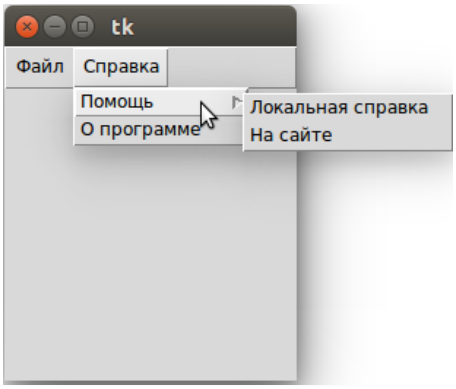


На основное меню *mainmenu*, добавляются не команды, а другие меню. У *filemenu* и *helpmenu* в качестве родительского виджета указывается не *root*, а *mainmenu*. Команды добавляются только к дочерним меню. Значение 0 опции **tearoff** отключает возможность откреплення подменю, иначе его можно было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

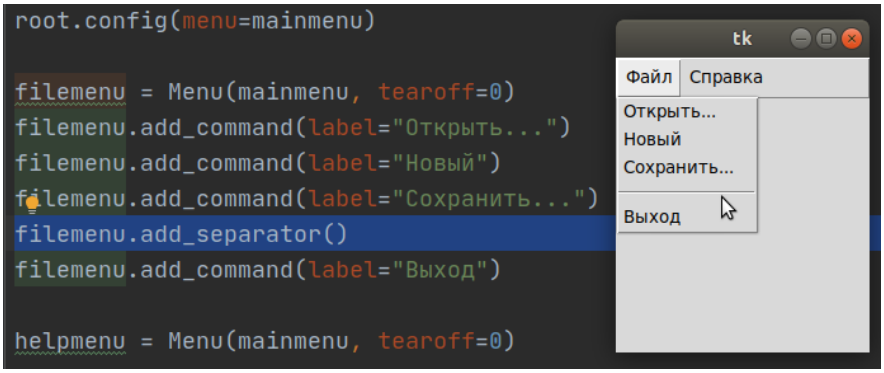
Точно также можно подвязывать дочерние меню к *filemenu* и *helpmenu*, создавая многоуровневые списки пунктов меню.

```
…
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")
…
```



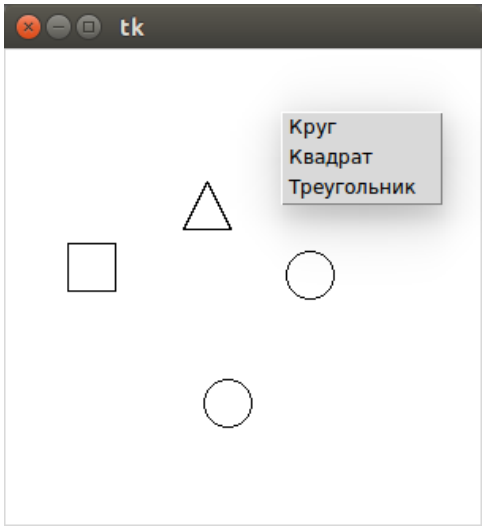
Метод **add_separator** добавляет линию разделитель в меню. Используется для визуального разделения групп команд.



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.

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.

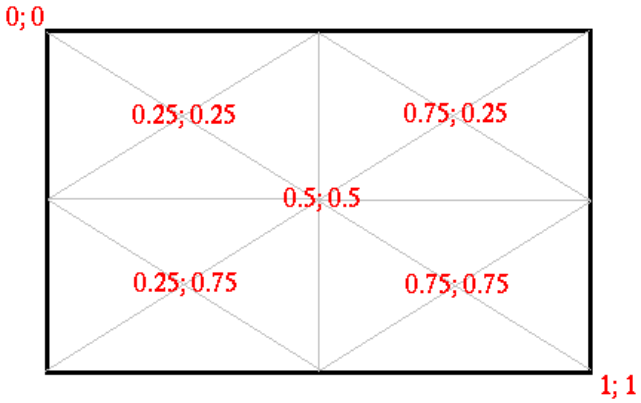
Методом **place** виджету указывается его положение либо в абсолютных значениях (в пикселях), либо в долях родительского окна, то есть относительно. Также абсолютно и относительно можно задавать размер самого виджета.

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.

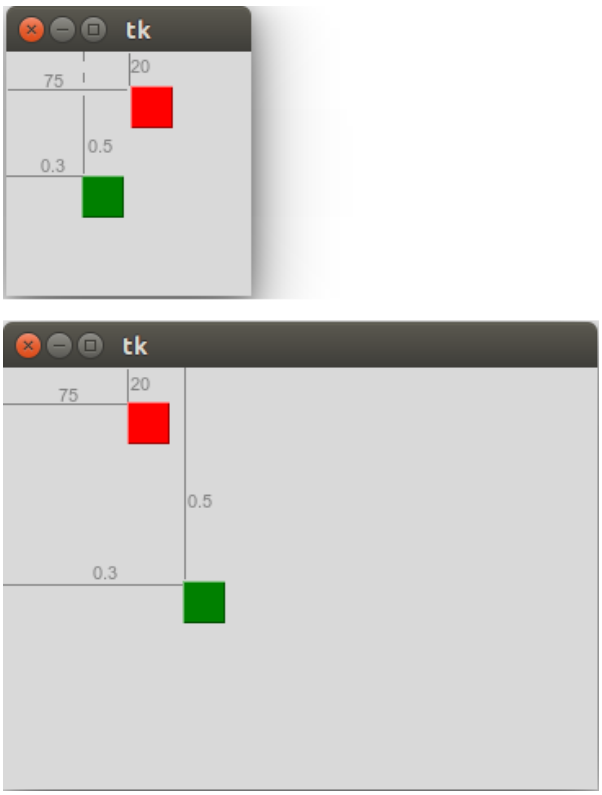
- **width**, **height** – абсолютный размер виджета в пикселях. Значения по умолчанию (когда данные опции опущены) приравниваются к естественному размеру виджета, то есть к тому, который определяется при его создании и конфигурировании.
- **x y** – абсолютная позиция в пикселях. Значения по умолчанию 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.

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**).

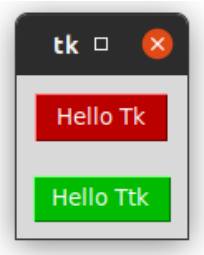
Сложность заключается в том, что в **ttk** свойства виджетов программируются не совсем так как в **tkinter**. Если в приложении сочетаются элементы сразу обоих модулей, код программы становится менее ясным.

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**:

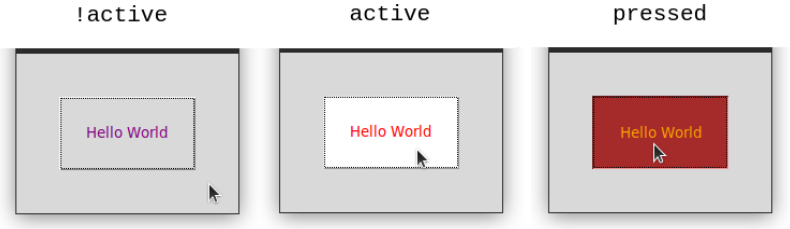
```
…
b_tk.config(background="red",
            foreground="white",
            activebackground="orange",
            activeforeground="white")
```

```
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.