

```
output=subprocess.check_output('python D:/work/aaa.py')
print(output)
```

Далее, 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. Определить 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 (entry), метка (label) и кнопка (button). Данные объекты создаются от соответствующих классов модуля tkinter. Мы сразу сконфигурируем некоторые их свойства с помощью передачи аргументов конструкторам этих классов:

```
e = Entry(root, width=20)
b = Button(root, text="Преобразовать")
l = Label(root, bg='black', fg='white', width=20)
```

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

Обратите внимание, что первым аргументом указывается "хозяин, мастер" – место, где располагается виджет. В данном случае его можно было не указывать. Однако виджеты не обязательно располагаются на root'e. Они могут размещаться в других виджетах.

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

```
def strToSortlist(event):
    s = e.get()
    s = s.split()
    s.sort()
    l['text'] = ' '.join(s)
```

У функций, которые вызываются при наступлении события с помощью метода bind(), должен быть один параметр. Обычно его называют event (событие).

В приведенной 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:

```
b.bind('<Button-1>', strToSortlist)
```

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

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

```
e.pack()
b.pack()
l.pack()
```

Метод `mainloop()` объекта Tk запускает главный цикл обработки событий, что в том числе приводит к отображению главного окна со всеми его причиндалами на экране:

```
root.mainloop()
```

Полный код программы:

```
from tkinter import *

root = Tk()

e = Entry(width=20)
b = Button(text="Преобразовать")
l = Label(bg='black', fg='white', width=20)

def strToSortlist(event):
    s = e.get()
    s = s.split()
    s.sort()
    l['text'] = ' '.join(s)

b.bind('<Button-1>', strToSortlist)

e.pack()
b.pack()
l.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 уместным. Пусть комплект из метки, кнопки и поля представляет собой один объект, порождаемый от некоего класса, скажем, `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.e = Entry(master, width=20)
        self.b = Button(master, text="Преобразовать")
        self.l = Label(master, bg='black', fg='white', width=20)
        self.b['command'] = self.strToSortlist
        self.e.pack()
        self.b.pack()
        self.l.pack()
    def strToSortlist(self):
        s = self.e.get()
        s = s.split()
        s.sort()
        self.l['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 (в данном случае это метод) 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 вынести установку значения для свойства command в отдельный метод, куда передавать привязываемую к кнопке функцию-обработчик события. Полный код программы:

```
from tkinter import *

class Block:
    def __init__(self, master):
        self.e = Entry(master, width=20)
        self.b = Button(master, text="Преобразовать")
        self.l = Label(master, bg='black', fg='white', width=20)
        self.e.pack()
        self.b.pack()
        self.l.pack()
    def setFunc(self, func):
        self.b['command'] = eval('self.' + func)
    def strToSortlist(self):
        s = self.e.get()
        s = s.split()
        s.sort()
        self.l['text'] = ' '.join(s)
    def strReverse(self):
```

```

        s = self.e.get()
        s = s.split()
        s.reverse()
        self.l['text'] = ' '.join(s)

root = Tk()

first_block = Block(root)
first_block.setFunc('strToSortlist')

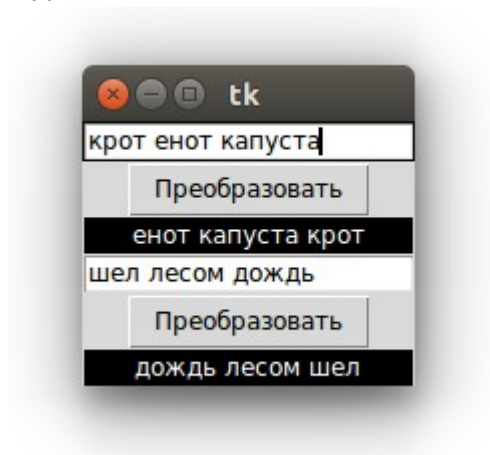
second_block = Block(root)
second_block.setFunc('strReverse')

root.mainloop()

```

Функция `eval()` преобразует строку в исполняемый код. В результате получается `self.b['command'] = self.strToSortlist` или `self.b['command'] = self.strReverse`.

При выполнении 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".

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 *

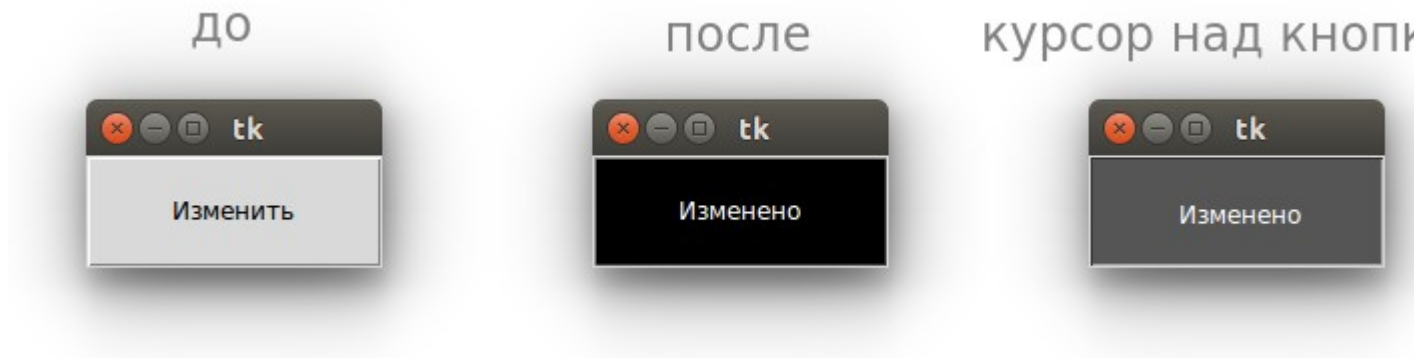
root = Tk()
b1 = Button(text="Изменить", width=15, height=3)

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

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

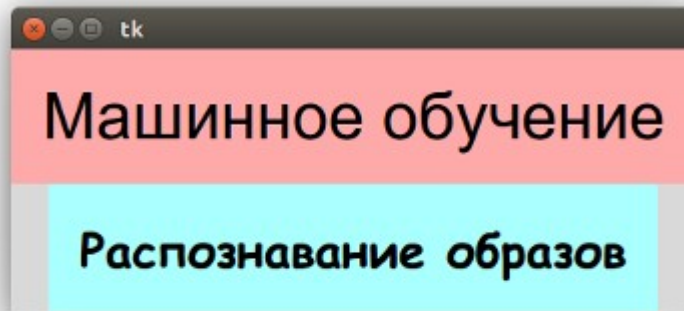
Виджет Label просто отображает текст в окне и служит в основном для информационных целей (вывод сообщений, подпись других элементов интерфейса). Свойства метки во многом схожи с таковыми у кнопки. Однако у меток нет опции `command`. Клик по метке не обрабатывается Tkinter. На примере объекта типа Label 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='#ffa aaa')
l2.config(bd=20, bg='#aaffff')
l1.pack()
l2.pack()
root.mainloop()
```

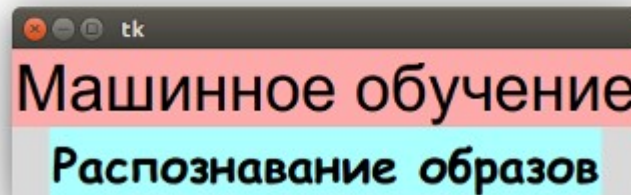
Значение шрифта можно передать как строку или как кортеж. Второй вариант удобен, если имя шрифта состоит из двух и более слов. После названия шрифта можно указать размер и стиль.

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

С границами



Без границ



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():  
    l['text'] = "Выдано"
```

```
root = Tk()  
Label(text="Пункт выдачи").pack()  
Button(text="Взять", command=take).pack()  
l = Label(width=10, height=1)  
l.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.

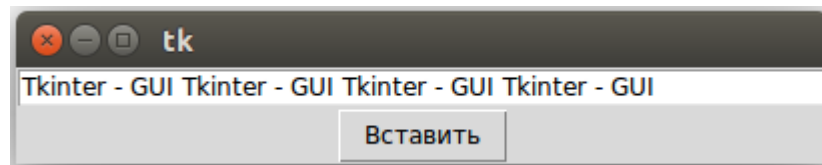
Свойства экземпляров Entry во многом схожи с двумя предыдущими виджетами. А вот методы – нет. Из текстового поля можно взять текст. За это действие отвечает метод `get()`. В текстовое поле можно вставить текст методом `insert()`. Также можно удалить текст методом `delete()`.

Метод insert() принимает позицию, в которую надо вставлять текст, и сам текст.

Такой код

```
from tkinter import *
root = Tk()
e1 = Entry(width=50)
def insert():
    e1.insert(0, "Tkinter - GUI ")
b = Button(text="Вставить", command=insert)
e1.pack()
b.pack()
root.mainloop()
```

приведет к тому, что после каждого нажатия на кнопку будет вставляться новая фраза "Tkinter - GUI " перед 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`.

Метод pack()

Прежде чем продолжить знакомство с виджетами GUI остановимся на вопросе их расположения в окне. Это важный вопрос, так как от интуитивности 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(), то не надо тут же использовать методы grid() (сетка) и place() (место).

Если в упаковщик 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.

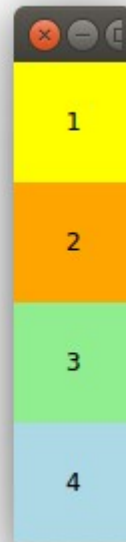
У метода pack() есть параметр side (сторона), который принимает одно из четырех значений-констант tkinter – TOP, BOTTOM, LEFT, RIGHT (верх, низ, лево, право). По умолчанию, когда в pack() не указывается side, его значение равняется TOP. Из-за этого виджеты располагаются вертикально.

Создадим четыре раскрашенные метки

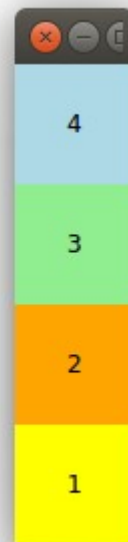
```
...  
l1 = Label(width=7, height=4, bg='yellow', text="1")  
l2 = Label(width=7, height=4, bg='orange', text="2")  
l3 = Label(width=7, height=4, bg='lightgreen', text="3")  
l4 = Label(width=7, height=4, bg='lightblue', text="4")  
...
```

и рассмотрим разные комбинации значений сайда:

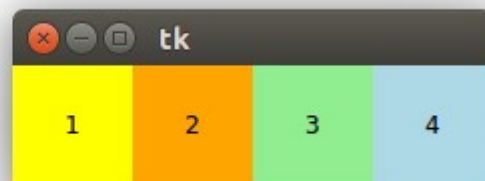
```
l1.pack()  
l2.pack()  
l3.pack()  
l4.pack()
```



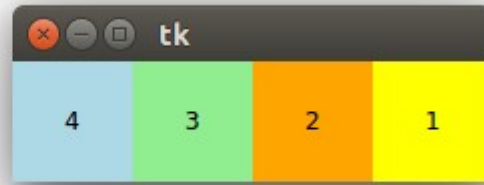
```
l1.pack(side=BOTTOM)  
l2.pack(side=BOTTOM)  
l3.pack(side=BOTTOM)  
l4.pack(side=BOTTOM)
```



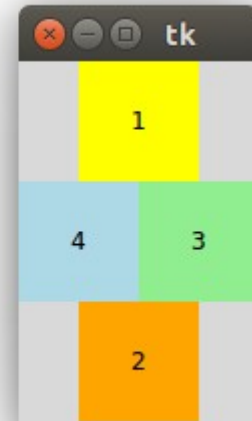
```
l1.pack(side=LEFT)  
l2.pack(side=LEFT)  
l3.pack(side=LEFT)  
l4.pack(side=LEFT)
```



```
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) # root можно не указывать
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)
```

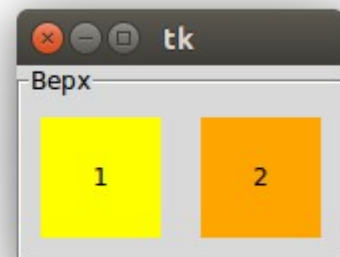
```
root.mainloop()
```



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(bg="lightgreen", width=30, height=10, text="This is a label")  
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 прибавляет к верху.



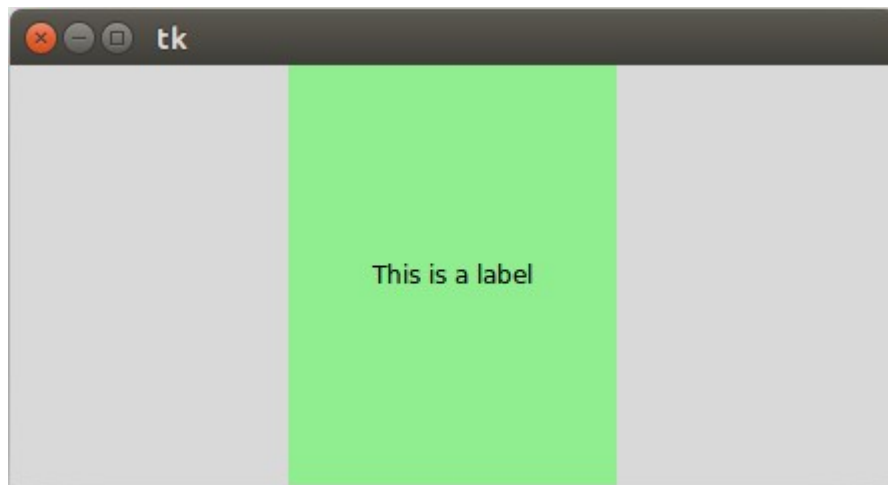
Если установить свойство expand в 1, то при расширении окна метка будет всегда в середине:

```
...  
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)  
...
```

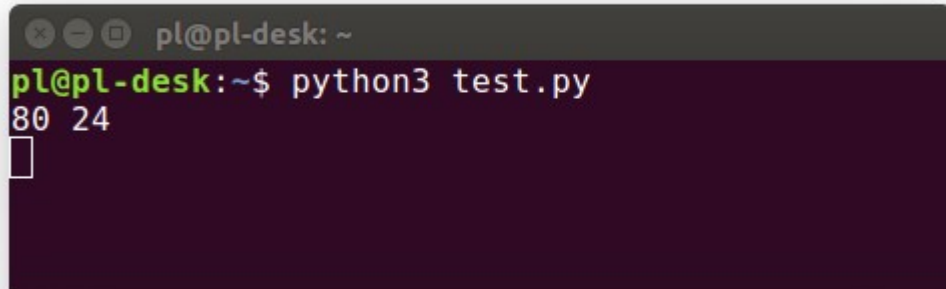


Text – многострочное текстовое поле

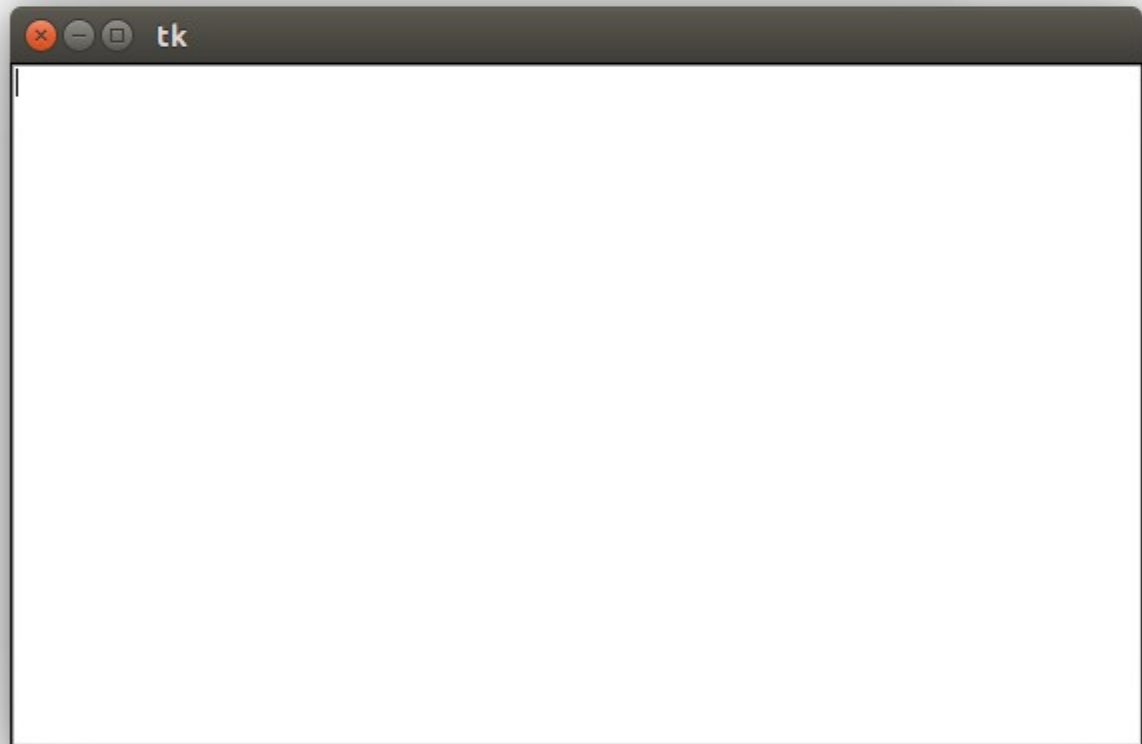
В этом уроке рассмотрим, как с помощью Tkinter запрограммировать такой элемент интерфейса как многострочное текстовое поле. Этот виджет часто встречается при заполнении веб-форм. В приложениях для десктопов он редок, если не считать программы "Терминал", где по-сути вы работаете в большом текстовом поле.

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

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

A terminal window with a dark purple background. The title bar shows 'pl@pl-desk: ~'. The prompt is 'pl@pl-desk:~\$'. The command 'python3 test.py' has been entered and executed, resulting in the output '80 24' on the next line. A cursor is visible on the line following the output.

```
pl@pl-desk:~$ python3 test.py
80 24
```

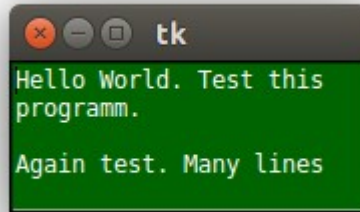


Однако эти свойства можно изменять с помощью опций width и height. Есть возможность конфигурировать шрифт, цвета и другое.

```
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.

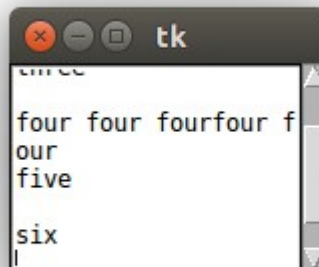
```
from tkinter import *
root = Tk()

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

def getText():
    s = text.get(1.0, END)
    label['text'] = s

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

root = Tk()

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

frame = Frame()
frame.pack()

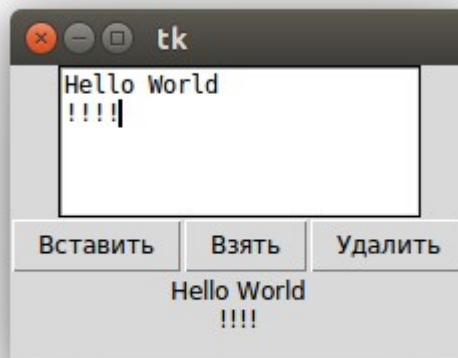
b_insert = Button(frame, text="Вставить", command=insertText)
b_insert.pack(side=LEFT)

b_get = Button(frame, text="Взять", command=getText)
b_get.pack(side=LEFT)

b_delete = Button(frame, text="Удалить", command=deleteText)
b_delete.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', font=("Verdana", 24, 'bold'), justify=CENTER)

root.mainloop()
```



Вставка виджетов в текстовое поле

В Text можно вставлять другие виджеты помощью метода `window_create()`. 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 *

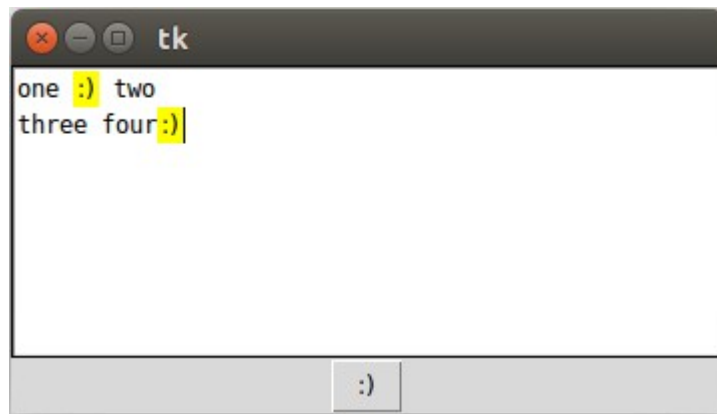
def smile():
    label = Label(text=":)", bg="yellow")
    text.window_create(INSERT, window=label)

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.

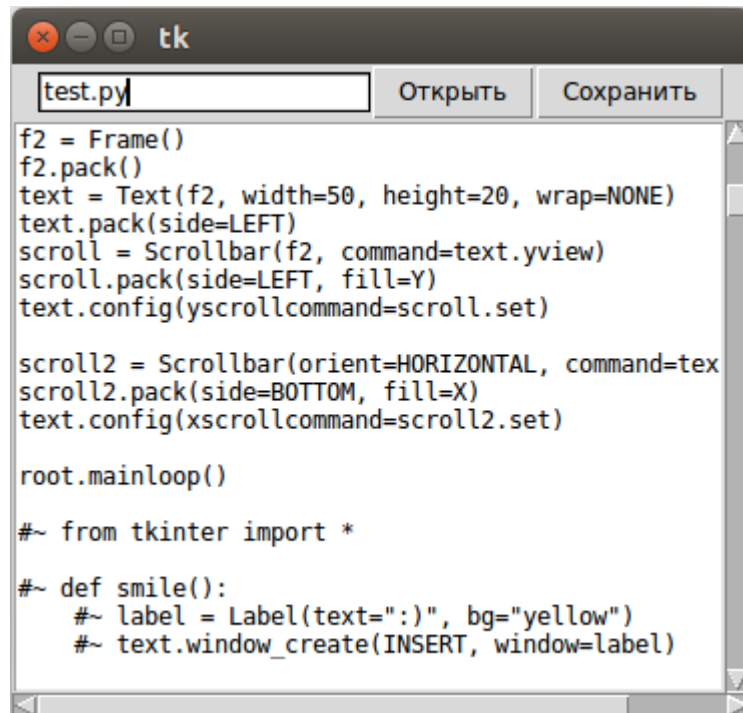
Практическая работа

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](#).



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 визуальнo 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

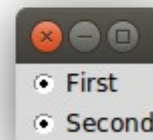
Если мы создадим две радиокнопки без соответствующих настроек, то обе они будут включены и выключить их будет невозможно:

```
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.

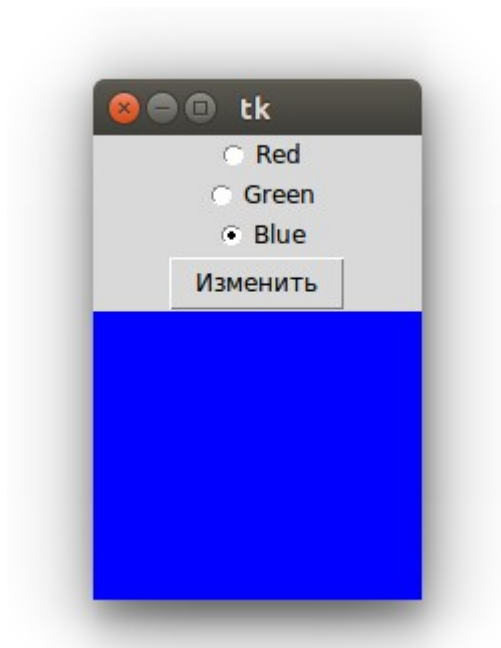
```
from tkinter import *

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.

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 ли флажок или снят, что в свою очередь повлияет на ход выполнения программы.

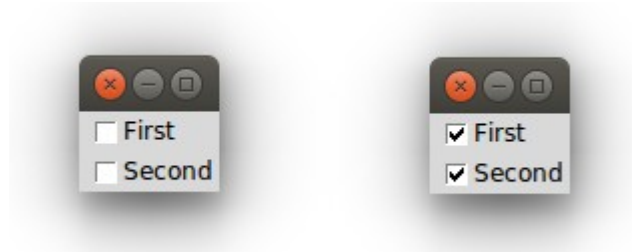
У каждого флажка должна быть своя переменная Tkinter.

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

cvar1 = BooleanVar()
cvar1.set(0)
c1 = Checkbutton(text="First", variable=cvar1, onvalue=1, offvalue=0)
c1.pack(anchor=W)

cvar2 = BooleanVar()
cvar2.set(0)
c2 = Checkbutton(text="Second", variable=cvar2, onvalue=1, offvalue=0)
c2.pack(anchor=W)

root.mainloop()
```



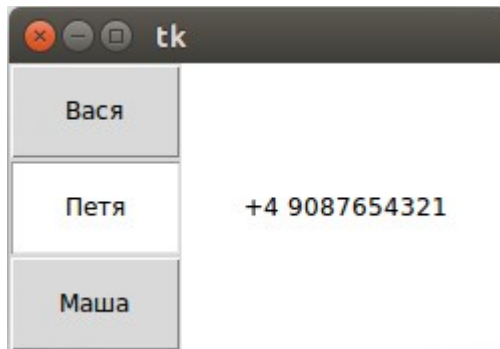
С помощью опции `onvalue` устанавливается значение, которое принимает связанная переменная при включенном флажке. С помощью свойства `offvalue` – при выключенном. В данном случае оба флажка будут выключены, так как методом `set()` были установлены соответствующие этому значения.

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

Практическая работа

Виджеты `Radiobutton` и `Checkbox` 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`.

Виджет Listbox

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

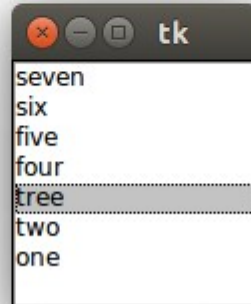
В Tkinter сначала создается экземпляр `Listbox`, после этого он заполняется с помощью метода `insert()`.

```
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.

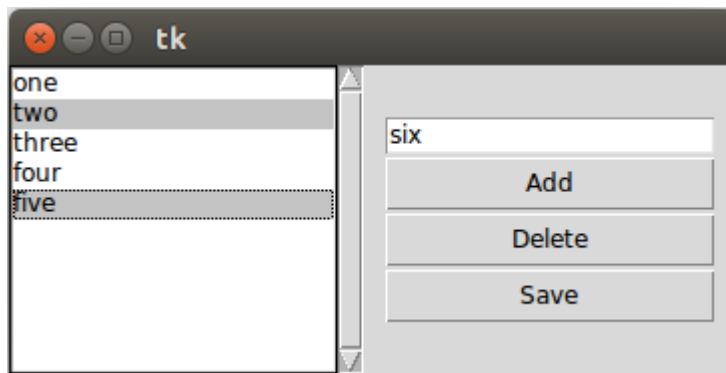
По-умолчанию в `Listbox`, кликая мышкой, можно выбирать только один элемент. Если необходимо обеспечить множественный выбор, то для свойства `selectmode` можно установить значение `EXTENDED`. В этом режиме можно выбрать сколь угодно элементов, зажав `Ctrl` или `Shift`.

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 addItem():
    lbox.insert(END, entry.get())
    entry.delete(0, END)

def delList():
    select = list(lbox.curselection())
    select.reverse()
    for i in select:
        lbox.delete(i)

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

root = Tk()

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

f = Frame()
f.pack(side=LEFT, padx=10)
entry = Entry(f)
entry.pack(anchor=N)
badd = Button(f, text="Add", command=addItem)
badd.pack(fill=X)
bdel = Button(f, text="Delete", command=delList)
bdel.pack(fill=X)
bsave = Button(f, text="Save", command=saveList)
bsave.pack(fill=X)

root.mainloop()
```

В функции `delList()` 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.

В функции `saveList()` 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

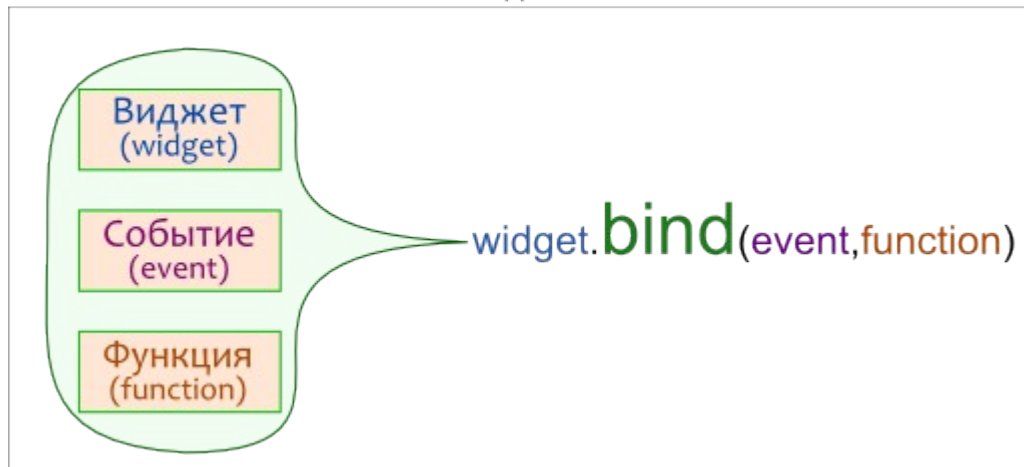
Напишите программу, состоящую из двух списков Listbox. В первом будет, например, перечень товаров, заданный программно. Второй изначально пуст, пусть это будет перечень покупок. При клике на одну кнопку товар должен переходить из одного списка в другой. При клике на вторую кнопку – возвращаться (человек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



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), которая вызывается при наступлении события.

Добавление функциональности графическому элементу с помощью метода bind



<http://younglinux.info>

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, то 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 event, f="Verdana": changeFont(event, f))
l.bind('<Button-3>', lambda event, f="Times": changeFont(event, 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 *
root = Tk()

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

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

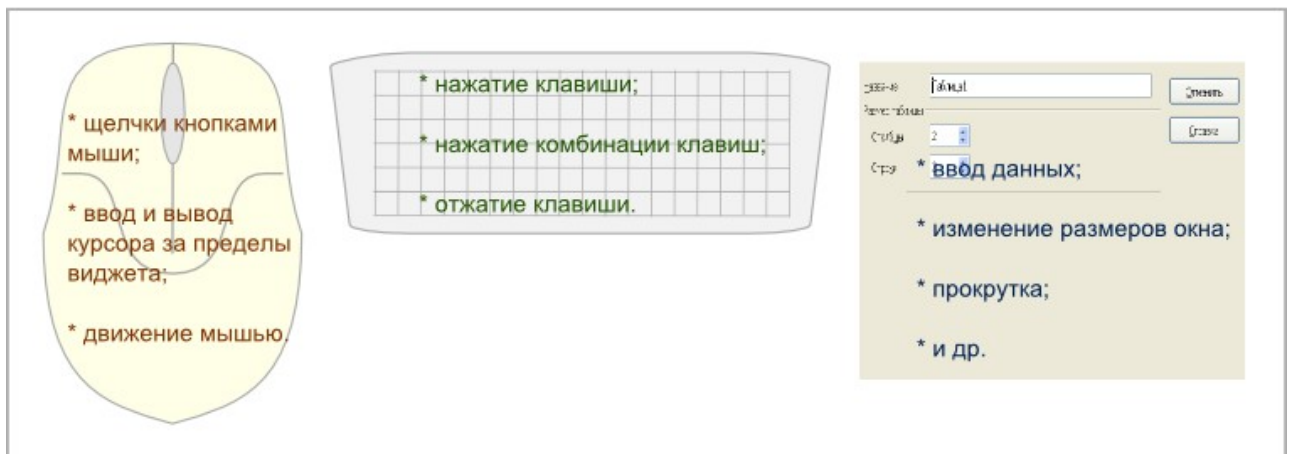
Напишите программу по следующему описанию. Нажатие Enter в однострочном текстовом поле приводит к перемещению текста из него в список (экземпляр Listbox). При двойном клике (<Double-Button-1>) по элементу-строке списка, она должна копироваться в текстовое поле.

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.

Типы событий



При вызове метода bind() событие передается в качестве первого аргумента.



`widget.bind(event,function)`

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

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.)

Сочетания 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 exitWin(event):
    root.destroy()

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

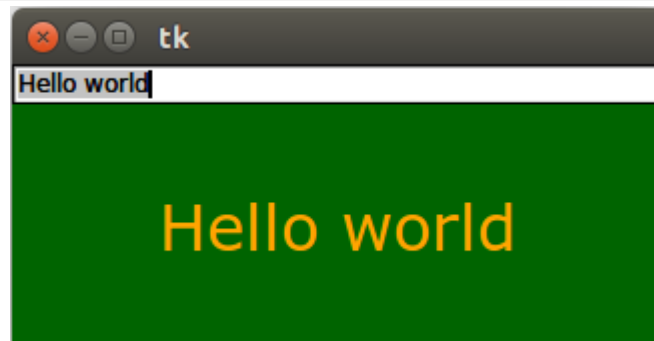
def selectAll(event):
    root.after(10, select_all, event.widget)
def select_all(widget):
    widget.selection_range(0, END)
    widget.icursor(END) # курсор в конец

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>', inLabel)
ent.bind('<Control-a>', selectAll)
root.bind('<Control-q>', exitWin)

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_all() и присвоено параметру widget.

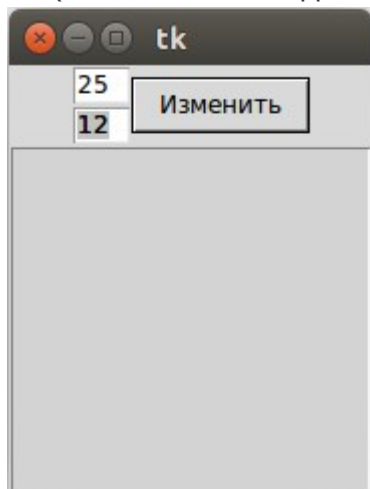
Практическая работа

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>.

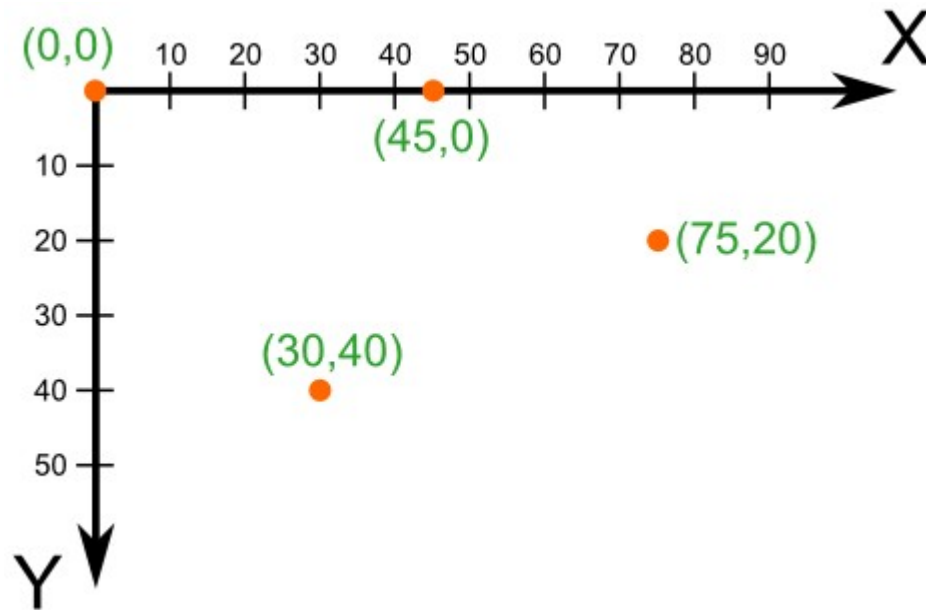
Для справки: фокус перемещается по виджетам при нажатии Tab, Ctrl+Tab, Shift+Tab, а также при клике по ним мышью (к кнопкам последнее не относится).



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).

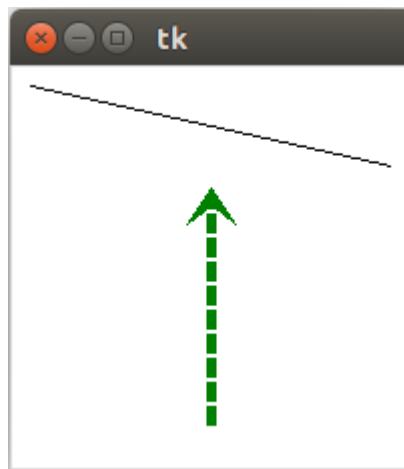
```
from tkinter import *
root = Tk()

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")

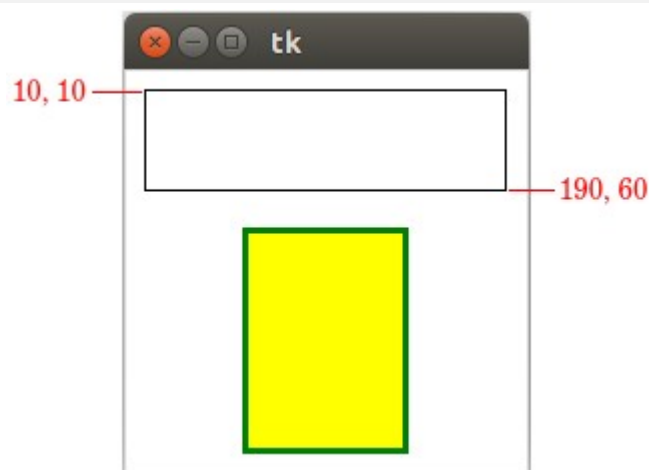
root.mainloop()
```



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

Создание прямоугольников методом `create_rectangle()`:

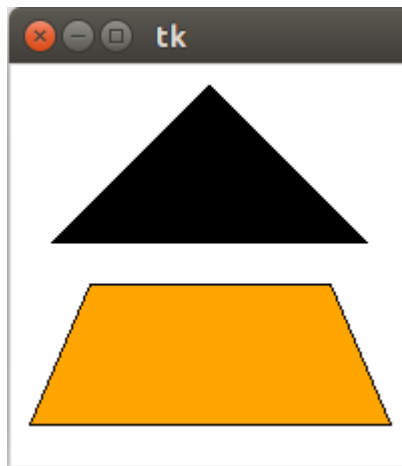
```
...
c.create_rectangle(10, 10, 190, 60)
c.create_rectangle(60, 80, 140, 190, fill='yellow', outline='green',
                  width=3, activedash=(5, 4))
...
```



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')
...
```

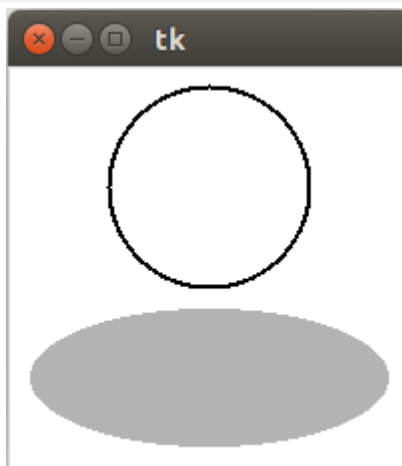


Для удобства координаты точек можно заключать в скобки:

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

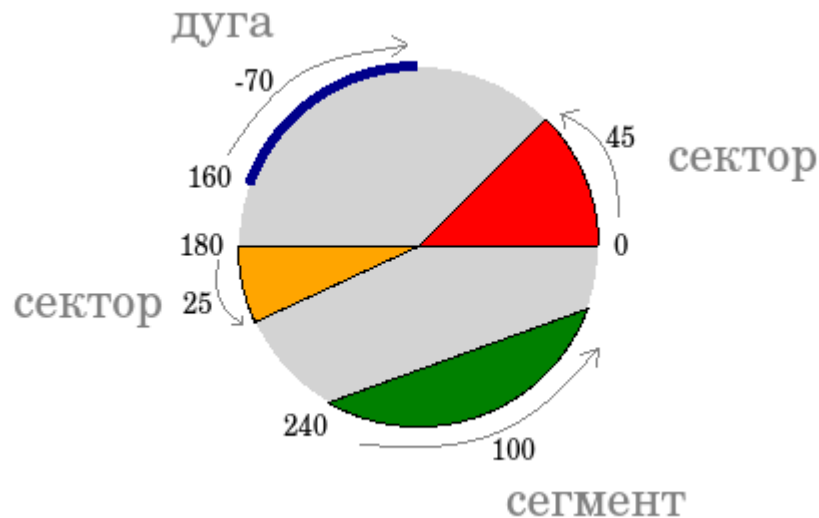
Метод `create_oval()` создает эллипсы. При этом задаются координаты гипотетического прямоугольника, описывающего эллипс. Если нужно получить круг, то соответственно описываемый прямоугольник должен быть квадратом.

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



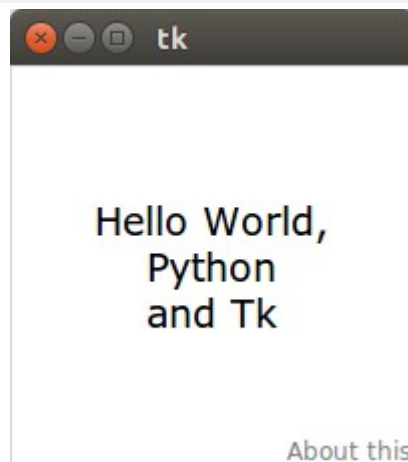
Более сложные для понимания фигуры получаются при использовании метода `create_arc()`. В зависимости от значения опции `style` можно получить сектор (по умолчанию), сегмент (CHORD) или дугу (ARC). Также как в случае `create_oval()` координаты задают прямоугольник, в который вписана окружность (или эллипс), 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()`:

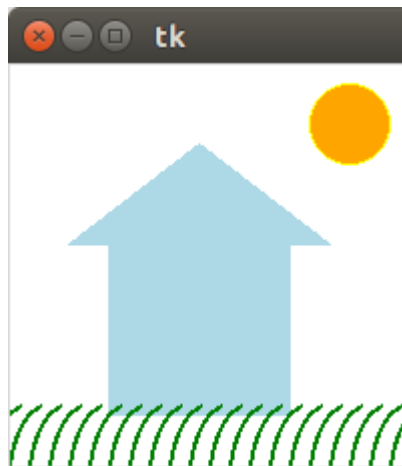
```
...
c.create_text(100, 100, text="Hello World,\nPython\nand Tk",
              justify=CENTER, font="Verdana 14")
c.create_text(200, 200, text="About this",
              anchor=SE, fill="grey")
...
```



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.

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 inFocus(event):
    c.itemconfig(rect, fill='green', width=2)
    c.coords(rect, 70, 70, 130, 130)
c.bind('<FocusIn>', inFocus)

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).

```

...
oval = c.create_oval(30, 10, 130, 80, tag="group1")
c.create_line(10, 100, 450, 100, tag="group1")

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

c.bind('<Button-3>', color)
...

```

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 *

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")

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)

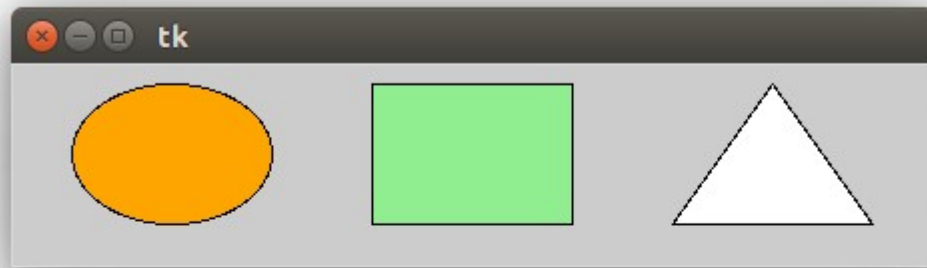
```

```

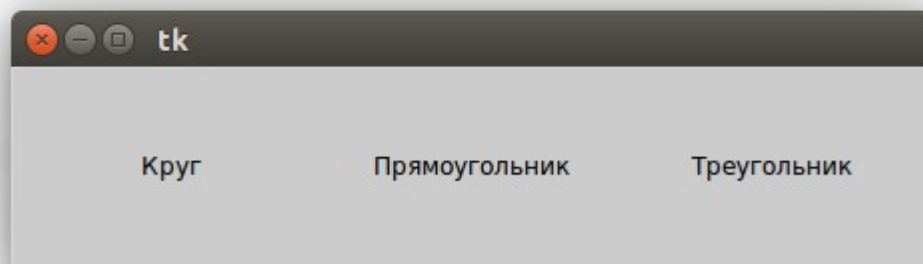
c.create_text(380, 50, text="Треугольник")
c.tag_bind(oval, '<Button-1>', oval_func)
c.tag_bind("rect", '<Button-1>', rect_func)
c.tag_bind(trian, '<Button-1>', triangle)
mainloop()

```

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 *

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():

```

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

motion()

root.mainloop()
```

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

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`).

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() # ширина экрана
h = root.winfo_screenheight() # высота экрана
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))

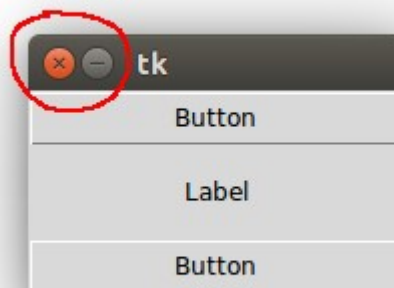
root.mainloop()

```

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.

нет третьей
кнопки



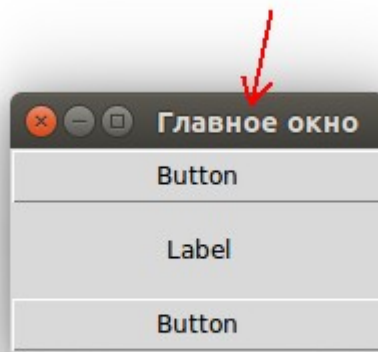
Заголовок окна

По умолчанию в строке заголовка окна находится надпись "tk". Для установки собственного названия используется метод `title()`.

```

...
root.title("Главное окно")
...

```



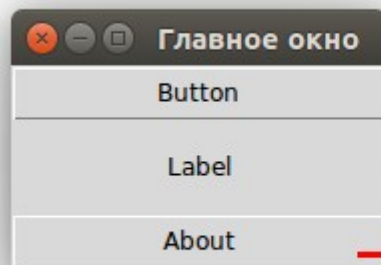
Если необходимо, заголовок окна можно вообще убрать. В программе ниже второе окно (Toplevel) открывается при клике на кнопку, оно не имеет заголовка, так как к нему был применен метод `overrideredirect()` с аргументом `True`. Через пять секунд данное окно закрывается методом `destroy()`.

```
from tkinter import *
root = Tk()
root.title("Главное окно")

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())

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

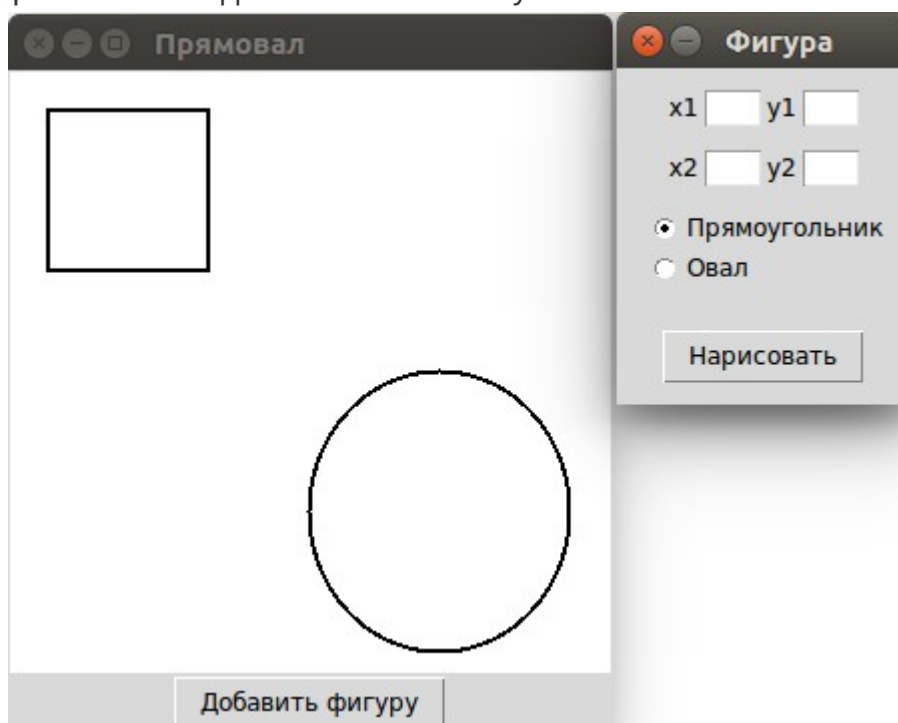
root.mainloop()
```



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.



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.

0, 0		
1, 0	1, 1	1, 2
2, 0		2, 2
3, 0		3, 2
4, 0		

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 (подобную разбивку можно произвести в уме):

Имя: 0, 0	Таблица: 0, 1, colspan=3		
Столбцов: 1, 0	2 1, 1	Строк: 1, 2	2 1, 3
Справка 2, 0	Вставить 2, 2		Отменить 2, 3

Теперь пишем код:

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

```
Label(text="Имя:").grid(row=0, column=0)
table_name = Entry(width=30)
table_name.grid(row=0, column=1, colspan=3)
```

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

```

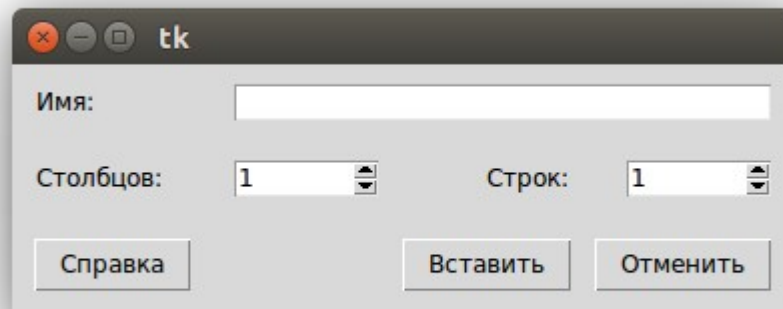
Label(text="Имя:").grid(row=0, column=0, sticky=W, pady=10, padx=10)
table_name = Entry()
table_name.grid(row=0, column=1, colspan=3, sticky=W+E, padx=10)

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

```

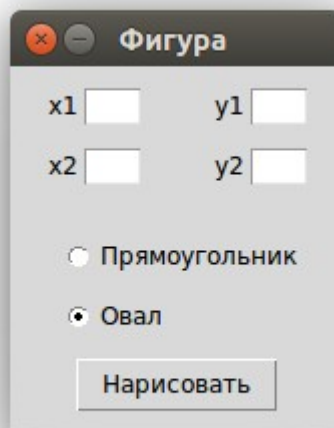


С помощью методов `grid_remove()` и `grid_forget()` можно сделать виджет невидимым. Отличие между этими методами лишь в том, что `grid_remove()` запоминает прежнее положение виджета. Поэтому для его отображения в прежней ячейки достаточно применить `grid()` без аргументов. После `grid_forget()` потребуется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

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()`.



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 == True:
        s = entry.get()
        entry.delete(0, END)
        label['text'] = s

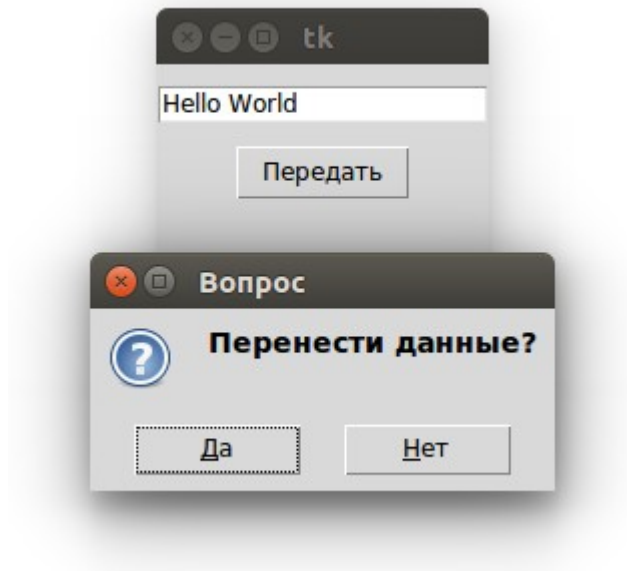
root = Tk()
entry = Entry()
entry.pack(pady=10)
```

```

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?")`.

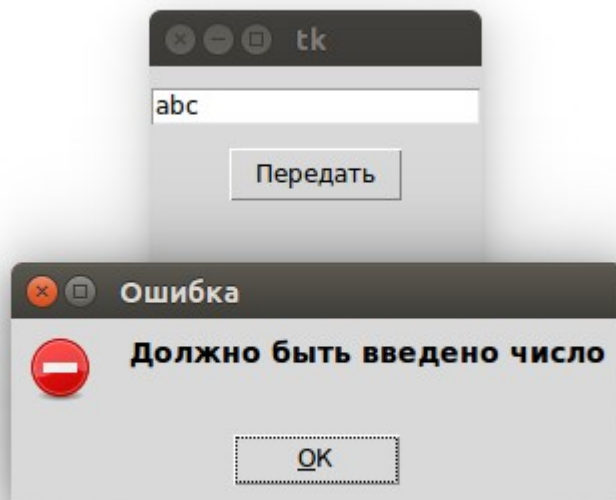
Подобные окна генерируются при использовании функции **askokcancel()** с надписями на кнопках "ОК" и "Отмена", **askquestion()** (возвращает не True или False, а строки 'yes' или 'no'), **askretrycancel()** ("Повторить", "Отмена"), **askyesnocancel()** ("Да", "Нет", "Отмена").

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 s.isdigit() == False:
        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

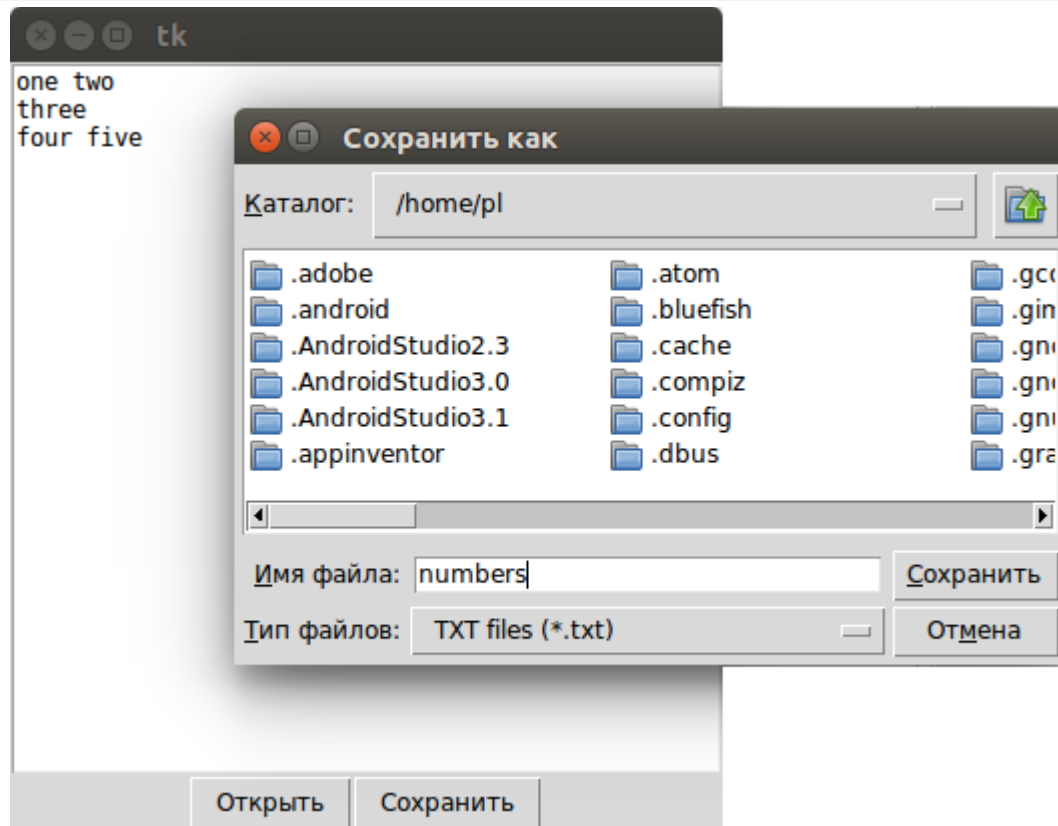
def insertText():
    file_name = fd.askopenfilename()
    f = open(file_name)
    s = f.read()
    text.insert(1.0, s)
    f.close()

def extractText():
    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=insertText)
b1.grid(row=1, sticky=E)
b2 = Button(text="Сохранить", command=extractText)
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.

