



Введение в научный Python.

Часть 2. Дополнительные темы

Оглавление

7. Создание и использование классов.....	1
8. Создание оконных приложений.....	14
8.1 Рисование с помощью «графического пера».....	14
8.2 Знакомство с модулем tkinter.....	18
8.3 Векторная графика в tkinter.....	51
8.4 Графика matplotlib в окнах tkinter.....	58
Заключительные замечания.....	68

7. Создание и использование классов.

В начале нашего пособия мы уже говорили, что переменные Python хранят информацию об объектах. Каждый объект относится к какому-нибудь типу данных. Типы пользователя в Python называются классами. Фактически класс представляет собой коллекцию данных и функций, которые называются атрибутами и методами. Атрибут – это переменная, метод – это функция. В языке Python все является объектами: числа, списки, функции, модули и т.д. Перечисленные понятия относятся к стандартным типам, но пользователь имеет возможность создавать собственные классы/типы.

Объект, созданный на основе некоторого класса, называется экземпляром класса. Экземпляры одного класса отличаются один от другого значениями своих атрибутов. Для доступа к экземплярам класса используются переменные. Переменная создается при присваивании ей значения. Во время присваивания в переменной сохраняется ссылка на экземпляр класса. Для доступа к атрибутам и методам конкретного экземпляра класса используется точка, которая разделяет имя объекта/переменной и имя атрибута или метода.

В этом параграфе мы поговорим о том, как создаются классы пользователя. Создайте файл со следующим кодом и выполните его.

```
# Пример самого простого (пустого) класса
class A:
    pass
```

При выполнении кода в рабочем пространстве исполнительной системы (интерпретаторе) появляется информация о созданном классе/типе. После определения класса можно создавать конкретные экземпляры этого типа. Для этого используется следующий синтаксис:

```
имя = Название_класса ([параметры]).
```

Создадим экземпляры класса A.

```
>>> a=A()
>>> b=A()
```

Как видите, класс создается с помощью ключевого слова `class`. Обычно тело класса содержит описание переменных и функций. Определение класса в тексте программы должно быть выполнено перед тем, как класс будет использоваться. Методы внутри класса создаются, как обычные функции с помощью инструкции `def`.

```
class B:
    def g(self):          # метод с именем g
        return 'Hello '+self.arg
```

Создадим объекты класса B и обратимся к их атрибутам и методам.

```
>>> b1=B()
>>> b2=B()
>>> b1.arg='Peter'
>>> b2.arg='Kate'
>>> b1.g()              # self не указывается при вызове метода
'Hello Peter'
>>> b2.g()
'Hello Kate'
```

Здесь экземпляры класса B имеют имена `b1` и `b2`. Обратите внимание на то, что в теле класса метод `g()` имеет обязательный аргумент `self`, который представляет ссылку на экземпляр класса, т.е. на конкретный объект. Всем методам класса в первом параметре автоматически передается эта ссылка. Этой переменной (ссылке) принято давать имя `self`. В теле методов доступ к атрибутам и методам производится через эту переменную с использованием точки разделителя, например, `self.arg`. Вне класса доступ к атрибутам выполняется через точку, стоящую после имени экземпляра класса, например, `b1.arg`. Обратите также внимание на то, что при вызове метода `g()` мы не передаем аргумент `self`, хотя при определении метода этот аргумент указан. Ссылку `self` на экземпляр класса интерпретатор передает автоматически – фактически инструкция `b1.g()` заменяется инструкцией `g(b1)`.

Тело класса составляет последовательность инструкций, которые выполняются на этапе определения класса, а не при создании экземпляра класса. Здесь мы работаем в интерактивной оболочке, но обычно инструкции создания класса выполняются во время импортирования модуля, содержащего его код. Отметим еще раз, что сама по себе инструкция `class` не создает никаких объектов и определение класса в некотором смысле можно рассматривать как некоторый шаблон.

Количество объектов, которые можно создать на основе класса, не ограничено. Обычно экземпляры одного класса имеют схожий набор атрибутов и методов. В то время, когда значения атрибутов различны, методы обычно одинаковы. Метод должен «знать», данные какого экземпляра он обрабатывает. Для этого ему в качестве первого (или единственного) аргумента передается ссылка `self` на текущий объект.

Наш последний пример с классом `B` не совсем корректен. Если перед инструкцией присваивания `b1.arg='Peter'` попробовать выполнить инструкцию `b1.g()`, то вы получите сообщение об ошибке:

```
...
AttributeError: 'B' object has no attribute 'arg'
```

Это связано с тем, что атрибута `arg`, к которому обращается функция `g()`, у экземпляра класса еще нет. Атрибут экземпляра создается во время первого присваивания `b1.arg='Peter'`. Даже выполнив такое присваивание, второй экземпляр `b2` все равно не будет иметь атрибута `arg`. Чтобы пользователь не забывал создавать нужные атрибуты, в классе можно определить метод `__init__()`, который вызывается автоматически при каждом объявлении нового экземпляра класса. Обычно в этом методе и создаются атрибуты экземпляров. Метод `__init__` называется конструктором и имеет следующий синтаксис:

```
def __init__(self[,имя1,...,имяN]):
    тело метода
```

С его помощью можно присвоить начальные значения атрибутам каждого экземпляра класса.

При создании экземпляра класса начальные значения указываются после имени класса в круглых скобках.

```
class B():
    def __init__(self,name):
        self.arg=name
    def g(self):
        return 'Hello '+self.arg
>>> b1=B('Peter')
>>> b2=B('Vasya')
>>> b1.g()
'Hello Peter'
>>> b2.g()
'Hello Vasya'
```

Изменить значение атрибута экземпляра класса можно с помощью присваивания нового значения составному имени атрибута (имя экземпляра, точка, имя атрибута). Например, для класса `B`, созданного выше, можно изменить значение атрибута `arg`, используя следующий код:

```
>>> b1.arg='Gregor' # к атрибуту можно обратиться непосредственно
>>> b1.g()
'Hello Gregor'
```

При этом изменение атрибута одного экземпляра не затрагивает значение одноименных атрибутов в других экземплярах.

```
>>> b2.g()
'Hello Vasya'
```

Пример. Создание и использование класса «Person».

```
class Person:
    def __init__(self,name,city, date):
        self.name=name
        self.city=city
        self.date=date
    def print(self):
        print('Name:',self.name,'; Adress:',self.city,'; Birthday:',self.date)
# использование класса
Vas=Person('Vasya','Kharkov','11-11-1990')
Kol=Person('Kolya','Kiev','10-02-1993')
Vas.print()
Kol.print()
Vas.name='Vasiliy'
Vas.print()
Name: Vasya ; Adress: Kharkov ; Birthday: 11-11-1990
Name: Kolya ; Adress: Kiev ; Birthday: 10-02-1993
Name: Vasiliy ; Adress: Kharkov ; Birthday: 11-11-1990
```

В этом примере мы создали тип данных (класс) «Person», который содержит поля (имя, город и дата) с информацией о конкретном человеке. Функция `__init__` (конструктор), вызывается при создании экземпляра класса, например, инструкцией `Vas=Person('Vasya','Kharkov','11-11-1990')`. Здесь `Vas` – имя экземпляра класса (имя переменной), а `Person` – имя класса (типа данных), после которого в круглых скобках указываются аргументы функции `__init__` (кроме первого).

Изменить значение атрибута можно путем присваивания его составному имени нового значения. В нашем примере инструкция `Vas.name='Vasiliy'` меняет значение атрибута `name` экземпляра `Vas`.

Метод `print` имеет аргумент `self`, а через него имеет доступ ко всем атрибутам (полям) текущего экземпляра, и печатает их значения в консоли. ■

Определение методов может находиться вне тела класса.

```
def Hello(self):
    return 'Hello '+self.name

class C():
    g=Hello # метод g является синонимом имени функции Hello
    def __init__(self,name):
        self.name=name
```

```
>>> c=C('Peter')
>>> c.g()
'Hello Peter'
```

Некоторые предопределенные имена методов в коде можно заменять обычными (например, арифметическими) операциями. Это называется перегрузкой операций. Например, если для моделируемого понятия определена операция сложения '+', то, создав в классе метод `__add__(y)`, для экземпляров `x` и `y` этого класса в коде программы можно будет использовать инструкцию вида `x+y`, которая, в действительности, заменится инструкцией `x.__add__(y)`. Сказанное поясним примером.

Пример. Создание и использование класса «двумерный вектор».

```
class Vec2d:
    def __init__(self,tpl): # tpl – кортеж из двух чисел (координат вектора)
        self.x=tpl[0]
        self.y=tpl[1]
    def print(self):
        print('Vector= (' ,self.x,',',self.y, ')')
    def __str__(self):      # перегрузка стандартной функции str()
        return '({},{})'.format(self.x,self.y)
    def __add__(self,v):    # перегрузка операции сложения
        return Vec2d((self.x+v.x,self.y+v.y))
    def __sub__(self,v):    # перегрузка операции вычитания
        return Vec2d((self.x-v.x,self.y-v.y))
    def __mul__(self,c):    # операция умножения на скаляр справа
        return Vec2d((self.x*c,self.y*c))
    def __rmul__(self,c):   # операция умножения на скаляр слева
        return Vec2d((self.x*c,self.y*c))

# использование класса
>>>a=Vec2d((1,2))          #a=Vec2d((float(input()),float(input()))))
>>>b=Vec2d((3,-1))         #b=Vec2d((float(input()),float(input()))))
>>>a.print()
Vector=(1,2)
>>>b.print()
Vector=(3,-1)
>>>z1=a+b
>>>print('a+b = ',str(z1))
a+b =(4,1)
>>>z2=a-b
>>>print('a-b = ',str(z2))
a-b =(-2,3)
>>>z3=a*5
>>>z3.print()
Vector=(5,10)
>>>z4=3*a
>>>z4.print()
```

```
Vector=(3,6)
>>>z5=4*(2*a-b*3)
>>>z5.print()
Vector=(-28,28)
```

В нашем примере код методов `__add__` и `__sub__` используется для выполнения операций сложения и вычитания векторов. Для операции умножения на скаляр пришлось написать два метода: `__mul__` и `__rmul__`. Вместо операции `a*5` выполняется метод `a.__mul__(5)`. Перед точкой стоит имя экземпляра класса, а в качестве аргумента используется скаляр. Однако для операции `3*a` такой способ не годится (не существует инструкции `3.__mul__(a)`). Для подобной ситуации предназначен метод `__rmul__`.

Инструкция `z5=4*(2*a-b*3)` показывает, что перегрузка арифметических операций значительно улучшает читабельность программы.

Можно (и нужно для реального класса «вектор») перегрузить ряд других операций, например, деление на скаляр или операцию проверки равенства векторов. Для реального класса нужно также создать множество методов, таких как вычисление длины вектора, скалярного умножения векторов и т.д. Кроме того, наши методы–операции не проверяют тип передаваемых аргументов, что также не соответствует поведению стандартных операций Python. Эти улучшения вы сможете выполнить после более подробного знакомства с особенностями разработки классов в Python.

■

Кроме арифметических, можно перегружать и другие операции. В примере мы показали, как перегрузить функцию `str`, предназначенную для преобразования экземпляров нашего класса в строку. Список операций и стандартных функций, которые разрешается перегружать, можно найти в учебниках по языку Python и в справочной системе.

Операции присваивания внутри инструкции `class` (не вложенные в инструкции `def`) создают атрибуты класса, а не атрибуты экземпляра класса. Числовые атрибуты класса похожи на статические переменные классов в C++: атрибут класса имеет одинаковое значение для каждого экземпляра класса (каждого объекта, созданного на основе класса). Аналогично, функции, созданные в классе, являются одинаковыми для всех экземпляров класса.

Следует учитывать то, что в одном классе могут одновременно существовать атрибуты класса и экземпляра с одинаковым именем.

```
class test:
    arg='Python'
    def __init__(self,name):
        self.arg=name
    def prn(self):
        return 'Hello '+self.arg
```

```
t1=test('Peter')
print(t1.prn())
print(test.arg,t1.arg)
```

```
Hello Peter
Python Peter
```

После выполнения инструкции `class` атрибуты класса (в том числе и функции) становятся доступны по их составным именам: `имя_класса.имя_атрибута`. Например, изменить значение атрибута `arg` класса `test` можно следующим образом.

```
>>> test.arg='Attribute'
>>> print(test.arg,t1.arg)
Attribute Peter
```

Атрибуты, созданные в теле класса, имеются у всех экземпляров этого класса.

В языке Python структура класса не является неизменной, и добавить новые атрибуты можно после его определения. При этом вновь создаваемые поля (атрибуты) могут быть одинаковыми для всех объектов класса или быть специфическими для конкретных экземпляров.

Вот пример создания атрибутов, специфических для конкретных экземпляров класса.

```
class A:
    pass
>>> a1=A()
>>> a2=A()
>>> a1.name='Sasha'
>>> a2.age=32
>>> a1.name
'Sasha'
>>> a2.age
32
>>> a1.age
Ошибка!
>>> a2.name
Ошибка!
```

Как видите, поле `name` есть только у объекта `a1`, а у объекта `a2` есть поле `age`, но поля `name` нет.

Объекты могут содержать произвольное количество собственных атрибутов (данных). Однако если вы хотите, чтобы все объекты класса имели одноименные атрибуты, то лучше определить их в конструкторе класса `__init__`.

Атрибуты объекта класса (а не экземпляра класса) также можно создавать динамически, используя синтаксис `имя_класса.имя_атрибута=новое_значение`.

```
class A:
    pass
>>> A.arg=15          # создает атрибут класса A
```

Теперь атрибут `arg` доступен всем создаваемым экземплярам класса `A` и, если одноименного атрибута экземпляра нет, то обращение к `arg` (чтение значения) можно выполнять также через имя экземпляра. Например,

```
>>> print(A.arg, a1.arg)
```

```
15 15
```

Атрибуты классов могут быть использованы как аналоги статических переменных классов в C++.

```
class Counter:
```

```
    i = 0
```

```
    def __init__(self) :
```

```
        self.__class__.i += 1
```

```
>>> print(Counter.i) # еще экземпляров класса нет
```

```
0
```

```
>>> c = Counter()
```

```
>>> print(c.i, Counter.i)
```

```
1 1
```

```
>>> d = Counter()
```

```
>>> print(c.i, d.i, Counter.i)
```

```
2 2 2
```

При каждом создании экземпляра класса функция `__init__` наращивает значение переменной `i`.

Обратите внимание на то, как мы обратились к атрибуту `i` класса `Counter` внутри метода `__init__` : `self.__class__.i += 1`. Обращение `self.i+=1` (без промежуточного идентификатора `__class__`) создало бы атрибут `i` экземпляра класса. Пока одноименного атрибута `i` экземпляра нет, обращение `c.i` (а также `d.i`) происходит к атрибуту класса (а не к атрибуту экземпляра). Вы должны понять разницу между атрибутами объекта класса и атрибутами экземпляра класса. Атрибут класса доступен всем экземплярам класса, и после его изменения новое значение будет одинаковым во всех экземплярах класса.

```
>>> Counter.i=25
```

```
>>> print(c.i, d.i, Counter.i)
```

```
25 25 25
```

Если конструктор вызывается при создании объекта, то при его уничтожении автоматически вызывается метод, называемый деструктором (если он имеется в классе). В языке Python деструктор реализуется методом `__del__()`. Он не вызывается до тех пор, пока на экземпляр класса существует хотя бы одна ссылка. В языке Python создание деструкторов классов требуется не часто.

В предыдущих примерах все атрибуты классов были открытыми – к ним можно было получить доступ с помощью составного имени (имя объекта, точка, имя атрибута). В объектно – ориентированном программировании (ООП) одной из основных парадигм является инкапсуляция, частью которой является скрытие данных. *Инкапсуляцией* называется механизм языка, позволяющий ограничить доступ одних компонентов программы к другим, в частности, к составляющим объект атрибутам и методам. Фактически инкапсуляция делает некоторые элементы класса (атрибуты и методы) доступными только внутри кодов методов этого класса. Одиночное подчеркивание в начале имени говорит о том, что соответствующий атрибут (или метод) предназначен только для

использования внутри класса. Тем не менее, атрибут доступен по этому имени (его использование нежелательно, но возможно).

```
class A:
    def _private(self):
        print('Одно подчеркивание не рекомендует вызывать метод')
```

```
>>> a=A()
>>> a._private()
```

Одно подчеркивание не рекомендует вызывать метод

Двойное подчеркивание в начале имени атрибута или метода запрещает его использование вне класса. Команда `имя_объекта.__имя_атрибута` приводит к ошибке.

```
class B:
    def __private(self):
        print('Это приватный метод')
```

```
>>> b=B()
>>> b.__private()
```

Ошибка!

Однако полной защиты это не дает. Доступ к методу (или атрибуту) можно получить по команде `имяОбъекта._имяКласса__имяМетода`. Например,

```
>>> b._B__private()
```

Это приватный метод

Повторим еще раз. Атрибуты будут доступны только из методов класса (а не из внешней программы), если их имена содержат не менее двух символов подчеркивания в начале и не более одного символа подчеркивания в конце.

Есть еще один способ ограничить перечень доступных атрибутов экземпляров класса. Для этого разрешенные атрибуты перечисляются внутри класса в атрибуте `_slots_`. Ему присваивается строка или список строк с названиями идентификаторов. При попытка обращения к атрибуту, отсутствующему в атрибуте `_slots_`, возбуждается исключение `AttributeError`.

Внутри класса можно создать идентификатор, через который в дальнейшем будут производиться операции чтения, изменения значения или удаления атрибута. Создается такой идентификатор с помощью функции `property`, имеющей следующий формат:

```
имя_свойства = property(    имя_метода_для_чтения[,
                             имя_метода_для_записи,
                             имя_метода_для_удаления,
                             строка_документирования])
```

В первых трех аргументах указываются ссылки на соответствующие методы класса. При чтении значения вызывается метод, указанный в первом параметре. Для операции присваивания будет вызван второй метод, который должен принимать один аргумент. При удалении атрибута вызывается третий метод. Если в качестве какого-либо аргумента задано значение `None`, то это означает, что соответствующий метод не поддерживается.

```

class test:
    def __init__(self, val):
        self.__arg=val
    def getvalue(self):
        return self.__arg
    def setvalue(self, val):
        if val>0: self.__arg=val
        else: self.__arg=0
    def delvalue(self):
        del self.__arg
        print('Атрибут val удален')
    v=property( getvalue, setvalue, delvalue, 'Строка документации')

```

```
>>> tval=test(123)
```

```
>>> tval.v=456
```

```
>>> print(tval.v)
```

```
456
```

```
>>> tval.v=-111
```

```
>>> tval.v
```

```
0
```

```
>>> del tval.v
```

```
Атрибут val удален
```

В этом классе test вместо защищенного атрибута __arg используется атрибут v, доступ к которому контролируется функциями getvalue и setvalue.

Другой важной парадигмой ООП является наследование. *Наследованием* называется комплекс правил, используемых при создании дочерних (производных) классов, которые содержат все атрибуты и методы родительского класса, а также содержат некоторые новые атрибуты и методы. Кроме того, некоторые методы и атрибуты в дочернем классе могут быть переопределены (заменены).

```

class base:
    def f1(self):
        print("Метод f1() класса base")
    def f2(self):
        print("Метод f2() класса base")

```

```

class derive(base):
    def f3(self):
        print("Метод f3() класса derive")

```

```
>>> d=derive()
```

```
>>> d.f1()
```

```
Метод f1() класса base
```

```
>>> d.f2()
```

```
Метод f2() класса base
```

```
>>> d.f3()
```

Метод f3() класса derive

Как видно, класс base указывается в круглых скобках при определении класса derive. В результате, класс derive наследует все атрибуты и методы класса base. Класс base называется базовым классом, а класс derive – производным классом. В определении производного класса в круглых скобках можно указать сразу несколько базовых классов через запятую. Это называется *множественным наследованием*.

Если имя метода в классе derive совпадает с именем метода в классе base, то используется метод класса derive. Если нужно вызвать одноименный метод из базового класса, то следует указать перед именем метода название базового класса. При этом в его первом аргументе необходимо явно передать ссылку на экземпляр класса.

Для вызова одноименного метода из базового класса, можно также использовать функцию `super()`. Она имеет следующий формат:

```
super([имя_класса, ссылка_self])
```

Для демонстрации сказанного, внесем изменения в два последних класса base и derive.

```
class base:
```

```
    def f1(self):
```

```
        print("Метод f1 () класса base")
```

```
    def f2(self):
```

```
        print("Метод f2 () класса base")
```

```
    def f3(self):
```

```
        print("Метод f3 () класса base")
```

```
class derive(base):
```

```
    def f3(self):
```

```
        print("Метод f3 () класса derive")
```

```
        super().f3()
```

```
    def f4(self):
```

```
        print("Метод f4 () класса derive")
```

```
        base.f3(self)
```

```
        super(derive,self).f3()
```

```
>>> d=derive()
```

```
>>> d.f1()          # метода f1 нет в производном классе
```

Метод f1() класса base

```
>>> d.f2()          # метода f2 нет в производном классе
```

Метод f2() класса base

```
>>> d.f3()          # вызывается метода f3 производного класса
```

Метод f3() класса derive

Метод f3() класса base

```
>>> d.f4()          # вызывается метода f4 производного класса
```

Метод f4() класса derive

Метод f3() класса base

Метод f3() класса base

Обратите внимание на разные способы вызова метода `f3` базового класса из методов `f3` и `f4` производного класса. В частности, при использовании функции `super()` не обязательно явно передавать аргумент `self` в вызываемый метод. Но если аргументы функции `super()` используются, то в ее первом аргументе указывается имя производного (текущего) класса, а не базового. Поиск имени атрибута или метода выполняется во всех базовых классах по цепочке вниз до первого найденного идентификатора.

Продемонстрируем сказанное на примере класса «person», приведенного нами ранее. Создадим из него производный класс «student». Студент является личностью и соответствующий класс должен содержать все атрибуты, которые имеет класс «person». Но в этом классе должна также содержаться дополнительная информация, описывающая студента, например, название факультета и номер группы. В действительности дополнительных полей может быть больше. Будем считать, что класс «person» создан (см. начало этого параграфа). Тогда класс «student» может иметь следующий вид.

```
class Student(Person): # указываем базовый класс
    def __init__(self, name, city, date, departmen, group):
        super().__init__(name, city, date)
        self.departmen = departmen
        self.group = group
    def __str__(self):
        strng = self.departmen + ' ' + self.group + ' ' + \
            self.name + ' ' + self.city + ' ' + self.date
        return strng
```

Вот, например, как можно использовать переменную этого класса.

```
>>> Nik = Student('Николенко', 'Харьков', '11.01.94', 'ФТФ', 'ТЯ51')
>>> print(str(Nik))
ФТФ ТЯ51 Николенко Харьков 11.01.94
```

Обратите внимание на то, что в конструкторе производного класса мы принудительно вызываем конструктор базового класса. Дело в том, что конструктор базового класса автоматически не вызывается, если он переопределен в производном классе.

Вот, например, как можно работать со списком студентов.

```
studList = [ ]
studList.append(Nik)
studList.append(Student('Петренко', 'Киев', '01.08.93', 'ММФ', 'ММ31'))
studList.append(Student('Рожко', 'Сумы', '23.12.93', 'КИС', 'КБ41'))
for stud in studList:
    print(str(stud))
ФТФ ТЯ51 Николенко Харьков 11.01.94
ММФ ММ31 Петренко Киев 01.08.93
КИС КБ41 Рожко Сумы 23.12.93
```

Для некоторых стандартных ситуаций можно создать методы с предопределенными именами. Они называются *специальными методами*. В частности, метод `__call__()` позволяет обработать вызов экземпляра класса

как вызов функции. Методы `__repr__(self)` и `__str__(self)` служат для преобразования объекта в строку. Например, добавьте в класс «Student» следующую функцию:

```
def __call__(self):  
    print(Student.__str__(self))
```

Тогда будет допустима следующая инструкция:

```
>>> Nik() # вызов имени объекта с круглыми скобками  
ФТФ ТЯ51 Николенко Харькоа 11.01.94
```

Мы не будем перечислять другие специальные методы. Познакомьтесь с ними самостоятельно по справочной системе или другим руководствам по Python.

В Python все является объектами и классы тоже. Как объекты классы имеют свои собственные автоматически создаваемые атрибуты. Например, атрибут `__module__` содержит имя модуля, в котором определен класс. А атрибут `__doc__` содержит строку документации класса (или None).

```
class A:  
    'Class documuntation'  
    pass  
>>> A.__doc__  
'Class documuntation'  
>>> A.__module__  
'__main__'
```

В заключении параграфа кратко перечислим некоторые рассмотренные нами концепции.

- Определение класса создает новый тип данных. Чтобы использовать класс, вы должны создать экземпляры класса.
- Доступ к атрибутам класса выполняется оператором `'.'` (точка). Доступом к атрибутам можно управлять, используя соглашения именования: без подчеркивания (открытые), с одинарным подчеркиванием, с двойным подчеркиванием (закрытые). Можно создать функцию – член `property`, контролирующую доступ к значениям атрибутов.
- Конструктор – это специальная функция член, которая автоматически вызывается каждый раз при создании экземпляра класса. Обычно он используется для создания и инициализации атрибутов экземпляра класса или выполнения любых других задач, подготавливающих класс к использованию. Конструктор должен быть определен не менее чем с одним параметром, который является ссылкой на экземпляр класса.
- Деструктор – это специальная функция – член, вызываемая автоматически при уничтожении экземпляра класса.
- Метод (функция – член) класса может быть определен внутри класса или вне него.
- Методы должны иметь как минимум один аргумент `self`, который содержит ссылку на объект, для которого была вызвана функция член.

- Если атрибут определен внутри класса вне методов, то он является атрибутом класса, и будет существовать единственная копия такой переменной – члена, независимо от числа созданных экземпляров класса. К атрибуту класса можно обратиться с использованием имени класса и операции точка без ссылки на экземпляр класса.
- При создании класса можно создать методы с предопределенными именами, которые используются в стандартных ситуациях: `__del__` (деструктор), `__cmp__`, `__getattr__`, `__setattr__`, `__call__` и другие.
- Для классов можно определять перегрузку операций. Она позволяет экземплярам классов участвовать в математических операциях. Чтобы перегрузить операцию необходимо в классе определить метод со специальным названием.
- Можно создавать производные классы. Длина цепочки производных классов неограничена. Допустимо множественное наследование.

Некоторых вопросов ООП мы коснулись только вскользь или даже не упомянули. Для более полного знакомства с механизмом создания и работы с классами в Python вам следует обратиться к другим руководствам.

8. Создание оконных приложений

8.1 Рисование с помощью «графического пера»

Python умеет работать с различными графическими библиотеками и позволяет создавать сложные программы с развитым пользовательским интерфейсом. Самую простую возможность рисования в графическом окне предоставляет модуль `turtle`. Он предназначен для рисования графических примитивов в собственном окне. Выполните следующие команды.

```
>>> import turtle
>>> turtle.reset()          # инициализация (открывает графическое окно)
>>> turtle.mainloop()      # открыть графическое окно
```

В результате откроется графическое окно с изображением стрелки (пера) ➤ в его центре (в зависимости от среды выполнения может быть достаточно одной из этих команд `reset()` или `mainloop()`). Используя команды перемещения пера (иногда его называют «черепашка» по имени модуля) можно создать несложный рисунок. Перо управляется командами относительных перемещений (вперед, назад), поворота, (вправо, влево) и абсолютных перемещений (к заданной точке). Перо оставляет след на плоскости рисования. Его можно поднять, тогда при перемещении след оставаться не будет. Для пера можно установить толщину и цвет.

Текущее направление перемещение пера (соответствующее направлению вперёд) указывается остриём стрелки. Основные команды рисования приведены ниже.

Команда	Описание
<code>forward(n)</code>	Передвижение пера вперёд (в направлении острия стрелки) на n точек.
<code>backward(n)</code>	Передвижение пера назад на n точек
<code>right(α)</code>	Поворот направо (по часовой стрелке) на α единиц (градусов или радиан).
<code>left(α)</code>	Поворот против часовой стрелки на α градусов или радиан.
<code>circle(r)</code>	Рисование окружности радиусом $ r $ точек из текущей позиции пера. Если r положительно, то окружность рисуется против часовой стрелки, если отрицательно, то по часовой стрелке.
<code>circle(r,α)</code>	Рисование дуги радиусом $ r $ точек с углом α градусов или радиан. Например, <code>turtle.circle(50, 90)</code>
<code>goto(x, y)</code>	Перемещение пера в точку с координатами x, y
<code>color('цвет')</code>	Установка цвета пера. Например, <code>turtle.color('blue')</code> <code>turtle.color('#ee77ff')</code>
<code>width(n)</code>	Задание толщины пера. Например, <code>turtle.width(3)</code>
<code>up()</code>	Поднять перо «над рисунком»
<code>down()</code>	Опустить перо на рисунок
<code>radians()</code>	Установка единиц измерения углов в радианы
<code>degrees()</code>	Установка единиц измерения углов в градусы
<code>write('текст')</code>	Печать текста в текущей позиции пера
<code>tracer(flag)</code>	Включение ($flag=1$) и выключение ($flag=0$) отображения следа пера
<code>clear()</code>	Очистка области рисования

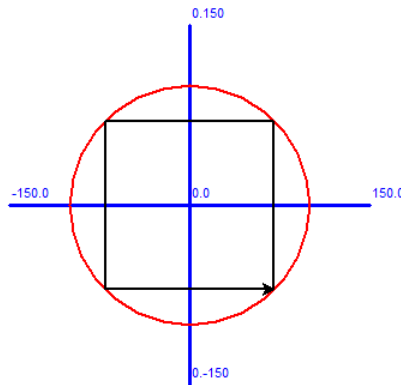
Пример. Выполните команды, которые создают рисунок, приведенный в конце примера. Для их ввода рекомендуем использовать текстовый редактор интерпретатора IDLE.

```
import turtle
turtle.reset()
# Рисование осей
turtle.color('#0000ff')
turtle.width(3)
turtle.up()
turtle.goto(-150,0)
turtle.down()
turtle.goto(150,0)
turtle.up()
turtle.goto(0,150)
turtle.down()
```

```

turtle.goto(0,-150)
# рисование координат точек
turtle.up()
xL=[0,0,0,150,-150]
yL=[0,150,-150,0,0]
for i in range(5):
    x=xL[i]
    y=yL[i]
    turtle.goto(x+3,y+3)
    coords=str(x)+'.'+str(y)
    turtle.write(coords)
# рисование окружности из текущей позиции пера
turtle.goto(0,-100)
turtle.color('#ff0000')
turtle.width(2)
turtle.down()
turtle.circle(100)
# рисование квадрата
turtle.up()
turtle.goto(70,-70)
turtle.down()
turtle.color('black')
for i in range(4):
    turtle.left(90)
    turtle.forward(140)

```



В таблице выше приведены только основные графические команды модуля `turtle`. Имеется много других, о которых вы можете узнать самостоятельно по справочной системе, выполнив инструкцию `help(turtle)`. Некоторые из них использованы в следующем примере.

Пример. *Использование функций модуля `turtle`.*

```

import turtle
turtle.reset()
turtle.goto(100,-30)
turtle.dot(30,'magenta')    # диаметр точки и цвет
turtle.speed(5)
turtle.up()
turtle.goto(0,0)

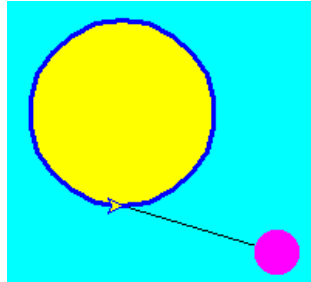
```



```

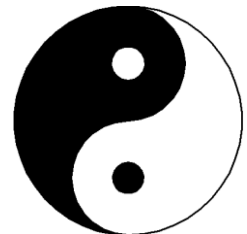
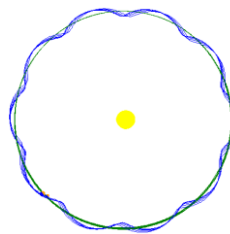
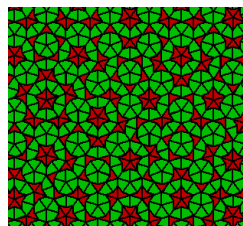
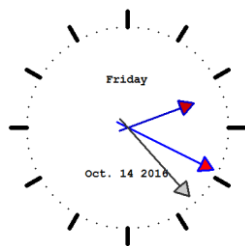
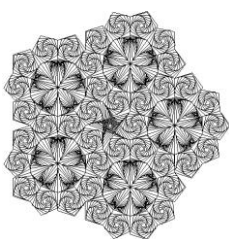
turtle.down()
turtle.color('blue','yellow') # цвет контура и цвет заливки
turtle.pensize(3)             # толщина пера
turtle.begin_fill()
turtle.circle(60)
turtle.end_fill()
ts=turtle.getscreen()
ts.bgcolor('cyan')

```



■

Большое количество примеров рисования с помощью функций модуля **turtle** можно найти в справочной системе IDLE в меню **Help->Turtle Demo**. Эта команда открывает окно, разделенное на две части. Выбор в меню **Examples** одного из примеров откроет его код в левой части. В правой половине будет расположено рабочее окно примера. Нажав появившуюся кнопку «Start», вы запустите программу на выполнение. Ниже мы приводим несколько рисунков, созданных этими примерами. Многие рисунки динамические. Например, стрелки «идущих» часов показывают системное время. А на четвертом рисунке показан кадр с траекторией движения планеты вокруг солнца (анимацию движения в текстовом документе воспроизвести невозможно).



Упражнения.

1. Используя функции модуля **turtle**, нарисуйте солнышко (желтый круг с лучами). Для рисования лучей используйте цикл.
2. Используя функции модуля **turtle**, нарисуйте график функции $\sin(x)$.

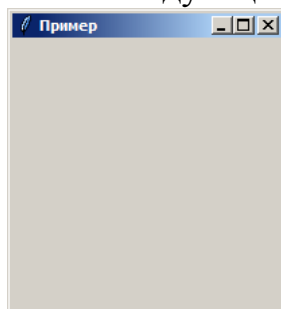
8.2 Знакомство с модулем tkinter

Серьезными графическими возможностями обладает модуль **tkinter**. Его основное назначение – создание графических интерфейсов для программ на Python. Благодаря наличию элемента холста (**Canvas**) его можно использовать для рисования векторной графики. Tkinter входит в стандартный дистрибутив Python. В этой главе будет дано введение в объекты и процедуры этого модуля.

Создайте текстовый файл со следующим кодом.

```
import tkinter
tk=tkinter.Tk()
tk.title("Пример") # следующий рисунок
tk.mainloop()
```

Результат его выполнения показан на следующем рисунке.



Чтобы остановить программу просто закройте окно.

Здесь инструкция `tk=tkinter.Tk()` создает объект «корневого» окна `tk`, которое представляет собой просто окно без содержимого. Инструкция `tk.title("Пример")` задает его заголовок, а инструкция `tk.mainloop()` запускает бесконечный цикл обработки сообщений. В результате создается приведенное выше окно программы.

Пример. Программа «Hello world».

```
import tkinter
top=tkinter.Tk()
label = tkinter.Label(top,text= 'Hello World!')
label.pack()
top.mainloop()
```



Во второй строке программы создается объект `top` корневого окна (окно верхнего уровня). За ним следует создание визуального элемента **Label** (метка), который содержит знаменитую строку. Элемент `Label` в основном предназначен для отображения текста, но он может содержать и изображения. Метод `pack()` этого элемента вызывает менеджер размещения. Он располагает графический элемент в окне родителя по определенным правилам. Затем инструкцией `top.mainloop()` запускается бесконечный цикл обработки сообщений.

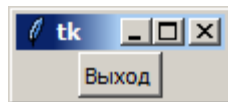
■

В программе можно создать только одно окно верхнего уровня, и оно должно уже существовать при создании любых других визуальных элементов. Его размеры автоматически выбираются такими, чтобы вместить все элементы.

В tkinter визуальные элементы управления называются виджетами (widget, от англ. window gadget). Они создаются вызовом конструктора соответствующего класса. Первый аргумент конструктора – это родительский элемент управления (или окно), в который будет помещён наш элемент. Далее идут необязательные аргументы, настраивающие элемент управления. Это могут быть тип шрифта (font=...), его размер, цвет фона (bg=...), команда, выполняющаяся при активации элемента управления (command=...) и т.д.

Пример. *Программа с одной кнопкой.* Пример программы напоминает предыдущий, но вместо простой текстовой метки создается кнопка.

```
import tkinter
top=tkinter.Tk()
btn = tkinter.Button(top,text='Выход', command=top.destroy)
btn.pack()
top.mainloop()
```



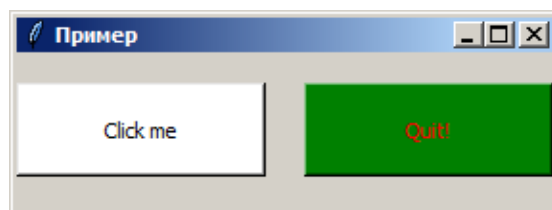
Созданная кнопка имеет один дополнительный параметр `command`, которому сообщается имя процедуры, выполняемой при нажатии на кнопку. В данном случае стандартная процедура `top.destroy` завершает работу всей программы.

Окно приложения не появится до тех пор, пока поток управления не войдет в бесконечный цикл обработки сообщений, запускаемый инструкцией `top.mainloop()`. В этом цикле обрабатываются как сообщения, создаваемые пользователем («нажатие» мышью на виджет **Button**), так и системные сообщения (например, восстановление окна после удаления перекрывающего окна другой программы). Программа остается в цикле обработки сообщений пока не будет закрыто окно. Без инструкции `top.mainloop()` кнопка в окне программы может не реагировать на внешние события.

■
Пример. *Программа с двумя кнопками.* Первая кнопка печатает в окне консоли текст 'Hello World!', вторая – завершает приложение.

```
import tkinter
tk=tkinter.Tk()
tk.title("Пример")
tk.geometry('280x80')      # задание размеров родительского окна
def Hello(event):
    print("Привет мир!")    # печать текста в командном окне
btn = tkinter.Button(tk,   # tk - родительское окно
    text="Click me",       # надпись на кнопке
    width=20,height=3,     # ширина и высота
    bg="white",fg="black") # цвет фона и надписи
```

```
# при щелчке по кнопке вызывается функция Hello
btn.bind("<Button-1>", Hello)
btn.pack(side = 'left')      # поместить кнопку в главном окне слева
# Кнопка завершения
btnQuit = tkinter.Button(tk,
    text='Quit!',
    width=20,height=3,      # ширина и высота
    bg="green",fg="red",    # цвет фона и надписи
    command=tk.destroy)
btnQuit.pack(side = 'right') #поместить кнопку в главном окне справа
tk.mainloop( )
Окно программы показано на следующем рисунке.
```



Кнопки привязаны к серединам левого и правого краев окна. При щелчке по левой кнопке в окне интерпретатора печатается текст «Привет мир!». Правая кнопка завершает приложение.

При сохранении Python программ обратите внимание на расширение имени файла. Если имя программы имеет расширение `.py`, то при ее запуске в Windows (например, двойным щелчком мыши), кроме окна программы, появляется окно консоли. Если имя имеет расширение `.pyw` (а не `.py`), окно консоли не появляется.

Инструкция `tk.geometry('280x80')` задает размеры окна. Вообще метод `geometry` устанавливает положение и размеры окна с помощью строки `"width x height+x+y"`, где `width` и `height` ширина и высота окна, а `x` и `y` – координаты его левого верхнего угла. При этом координаты отсчитываются от левого верхнего угла «родителя» в направлениях вправо и вниз. Например, инструкция `root.geometry("400x300+40+60")` помещает окно в точку с координатам 40,60 и задает его размер 400 x 300 пикселей. Размер или координаты могут быть опущены. Допустимы инструкции `root.geometry("600x400")` или `root.geometry("+40+80")`, которые задают только размер или только положение окна.

В примере мы создаём два экземпляра класса **Button**. Обоим экземплярам указываем родительское окно и задаем дополнительные аргументы (текст на кнопке, ее размеры, цвет и т.д.). Для первой кнопки метод `bind` привязывает функцию `Hello` к событию нажатия на кнопку. В результате, при нажатии левой кнопки мыши в области элемента «Click me» вызывается функция `Hello`. С помощью аргумента `command` привязываем процедуру завершения приложения к событию нажатия мышью на вторую кнопку. Функция `pack()` располагает графические элементы в окне родителя

(положение элементов управляется с помощью аргумента). Функция `mainloop()` запускает цикл обработки событий.

У многих элементов управления имеется «основное» событие, на которое они реагируют. Такое событие можно обрабатывать с помощью функции, имя которой указывается в опции `command` при создании виджета. Этой функции аргументы не передаются. Для виджета **Button** таким событием является «click» (щелчок левой кнопки мыши). В нашем примере эту опцию мы использовали у кнопки «Quit!» для задания процедуры завершения приложения.

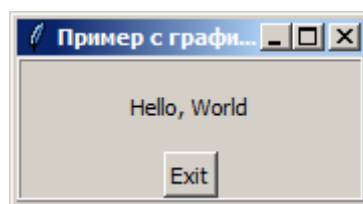
Однако элементы управления могут реагировать на множество других событий. Для сообщения виджету, на какое событие и как он должен реагировать, используется метод `bind(event, имя_процедуры[, "+"])`. Первый аргумент `event` – это строка специального содержания, обозначающая событие. У виджета **Button** эта строка для события нажатия на левую кнопку мыши выглядит так: "<Button-1>". Второй аргумент метода `bind` сообщает, какая функция будет вызываться при обработке события. Третий необязательный аргумент – строка "+", которая означает, что обработчик события добавляется к уже существующим. В примере метод `bind` был использован для первой кнопки: `btn.bind("<Button-1>", Hello)`. Функция `Hello` обработки события "<Button-1>" (это тот же «click») имела следующий вид:

```
def Hello(event):  
    print("Привет мир!")
```

■

Пример. *Размещение элементов управления в виджете Frame (рамка).* В предыдущих примерах мы располагали виджеты в основном окне приложения. Здесь мы покажем, как размещаются одни виджеты в других.

```
import tkinter  
from tkinter.constants import *  
tk=tkinter.Tk()  
tk.title("Пример с графическим интерфейсом")  
frame = tkinter.Frame(tk, relief=RIDGE, borderwidth=2)  
frame.pack(fill=BOTH, expand=1)  
label = tkinter.Label(frame, text="Hello, World")  
label.pack(fill=X, expand=1)  
button = tkinter.Button(frame, text="Exit", command=tk.destroy)  
button.pack(side=BOTTOM)  
tk.mainloop()
```



Виджет **Frame** (рамка) предназначен для группировки других виджетов. В программе функция-упаковщик `pack()` элемента **Frame** использует опцию

`fill=BOTH`, которая означает, что виджет будет заполнять всю область своего родителя (в данном случае все окно приложения) при изменении размеров родительского элемента. Возможны также значения `fill=NONE`, `X` или `Y`. Значение `fill=X`, используемое при размещении метки (**Label**), означает, что она должна «подстраиваться» к размеру родителя по ширине (по `X` координате). Если внутри главного окна расположить несколько виджетов и изменить размер окна приложения, то подстраивать свои размеры будут только те элементы, для которых установлена опция `expand=1` (или `expand=True`). Менеджер расположения будет выделять дополнительное пространство только для таких элементов. Тем самым опция `fill` выражает только пожелание поведения элемента, а опция `expand` дает разрешение на изменение его размеров.

Опция `relief`, используемая при создании элемента **Frame**, задает стиль контура виджета и имитирует трехмерные эффекты. Возможны следующие значения этой опции: `FLAT` (плоский), `RAISED` (приподнятый), `SUNKEN` (утопленный), `GROOVE` (контур в форме паза), `RIDGE` (контур в форме ребра). Опция `side=BOTTOM` упаковщика `pack()`, использованная при размещении кнопки, привязывает ее к нижней части родительского виджета **Frame**.

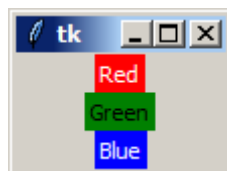
■

Метод `pack()` – это один из трех упаковщиков, имеющихся в `tkinter`. Упаковщики отвечают за размещение визуальных элементов в главном окне или внутри родительского виджета. Для каждого элемента нужно вызывать упаковщик. До сих пор мы использовали упаковщик `pack()`, который автоматически размещает виджеты в родительском окне. Рассмотрим его и другие упаковщики подробнее.

Пример. *Размещения нескольких виджетов с помощью упаковщика `pack()`.*

Для размещения нескольких виджетов один над другим можно использовать метод `pack()` без аргументов.

```
from tkinter import *
root = Tk()
Label(root, text="Red", bg="red", fg="white").pack()
Label(root, text="Green", bg="green", fg="black").pack()
Label(root, text="Blue", bg="blue", fg="white").pack()
mainloop()
```



С помощью опции `fill=X` можно растянуть виджеты на ширину родительского окна.

```
from tkinter import *
root = Tk()
Label(root, text="Red", bg="red", fg="white").pack(fill=X)
Label(root, text="Green", bg="green", fg="black").pack(fill=X)
```



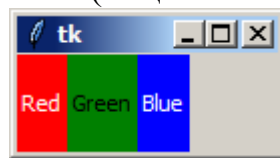
```
Label(root, text="Blue", bg="blue", fg="white").pack(fill=X)
mainloop()
```



Для размещения виджетов один рядом с другим, можно использовать опцию `side`. Чтобы высота виджета совпадала с высотой своего родителя, можно использовать опцию `fill=Y`.

```
from tkinter import *
root = Tk()
Label(root, text="Red", bg="red", fg="white").pack(side=LEFT,fill=Y)
Label(root, text="Green", bg="green", fg="black").pack(side=LEFT,fill=Y)
Label(root, text="Blue", bg="blue", fg="white").pack(side=LEFT,fill=Y)
mainloop()
```

На рисунке показан вид программы после изменения размеров родительского окна. Элементы пристыкованы к своему левому соседу (опция `side=LEFT`), и их высота совпадает с высотой окна (опция `fill=Y`).



Пример. Использование различных значений опции `side` метода `pack()`.

```
from tkinter import *
root = Tk()
Label(root, text = 'FLAT', relief=FLAT).pack(side = 'left')
Label(root, text = 'RAISED',relief=RAISED).pack(side = 'top')
Label(root, text = 'SUNKEN',relief=SUNKEN).pack(side = 'right')
Label(root, text = 'GROOVE',relief=GROOVE).pack(side = 'bottom')
Label(root, text = 'RIDGE',relief=RIDGE).pack(fill = 'both', expand=1)
root.mainloop()
```



Окно программы показано на предыдущем рисунке слева. Обратите внимание на то, что если для метки `RIDGE` не использовать опцию `expand=1`, то она не будет «растягиваться» в обоих направлениях при изменении размера окна программы.

Полезными опциями метода `pack()` являются `ipadx`, `ipady`, `padx`, `pady`. Они позволяют задать минимальные размеры пространства вокруг надписей внутри элементов и непосредственно вокруг элементов. Значением по умолчанию этих опций является ноль. Измените в последнем примере инструкцию, создающую элемент с надписью `'SUNKEN'` следующим образом:

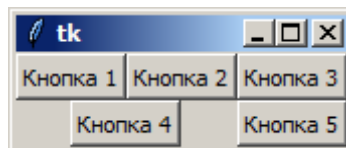
```
Label(root, text='SUNKEN', relief=SUNKEN).pack(
    side='right', ipady=12, padx=20)
```

Внутренняя высота кнопки 'SUNKEN' увеличится (ipady=12), а также появится зарезервированное пространство слева и справа (padx=20). Стартовое окно измененной программы показано на предыдущем рисунке справа.

Вместо pack() можно использовать метод grid(). Соответствующий упаковщик работает с таблицей ячеек, в которые помещаются визуальные элементы. Аргументы row и column метода grid (...) определяют строку и столбец в таблице, а аргументы rowspan и colspan задают количество строк и столбцов, занимаемых элементом.

Пример. Использование упаковщика grid.

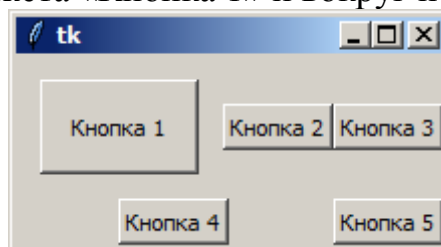
```
from tkinter import *
root = Tk()
Button(root, text = 'Кнопка 1').grid(row = 1, column = 1)
Button(root, text = 'Кнопка 2').grid(row = 1, column = 2)
Button(root, text = 'Кнопка 3').grid(row = 1, column = 3)
Button(root, text = 'Кнопка 4').grid(row = 2, column = 1, colspan = 2)
Button(root, text = 'Кнопка 5').grid(row = 2, column = 3)
root.mainloop()
```



У метода grid() имеются опции padx, ipady, pady. Измените в предыдущем коде инструкцию, создающую первую кнопку, следующим образом:

```
Button(root, text = 'Кнопка 1').grid(row = 1, column = 1, \
                                     padx=12, pady=12, ipadx=12, ipady=12)
```

Пространство внутри виджета «Кнопка 1» и вокруг него увеличится.



Третий упаковщик place() позволяет размещать виджеты в указанных координатах с заданными размерами. Опциям x и y присваиваются значения координат левого верхнего угла виджета, а опциям width и height – размеры (ширина и высота). При этом координаты отсчитываются от левого верхнего угла родительского виджета в направлениях вправо и вниз.

Пример. Примитивный калькулятор, виджеты которого размещаются с использованием метода place().

```
from tkinter import *
def is_a_b_numbers(): # проверка, что введены числа
    global a,b
```



```

astr=txtA.get().strip()
bstr=txtB.get().strip()
try:
    a=float(astr)
    b=float(bstr)
    return True
except ValueError:
    pass
return False

def sums(ev):
    if is_a_b_numbers():
        lblOp['text']='+'
        lblRez['text']=str(a+b)
    else:
        print("Input error!")
        return
    return

def muls(ev):
    if is_a_b_numbers():
        lblOp['text']='x'
        lblRez['text']=str(a*b)
    else:
        print("Input error!")
        return
    return

tk = Tk()
tk.title("Калькулятор")
tk.geometry('400x100')
frmMain = Frame(tk,bg = 'lightgray',relief=RIDGE,borderwidth=4)
frmMain.pack(fill='both',expand=True)

txtA = Entry(frmMain, font='Arial 10')
txtA.insert(0,"0.0")
txtA.place(x = 20, y = 20, width = 60,height=25)

lblOp=Label(frmMain, font='Arial 10', text=' ')
lblOp.place(x = 90, y = 20, width = 20,height=25)

txtB = Entry(frmMain, font='Arial 10')
txtB.insert(0,"0.0")
txtB.place(x = 120, y = 20, width = 60,height=25)

lblEq=Label(frmMain, font='Arial 10', text='=')
lblEq.place(x = 190, y = 20, width = 20,height=25)

```

```

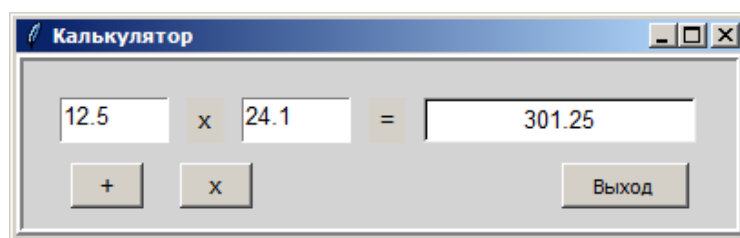
lblRez=Label(frmMain,font='Arial 10',bg='white',
              text=" ",relief=SUNKEN)
lblRez.place(x = 220, y = 20, width = 150,height=25)

btnPlus=Button(tk, text = '+',font='Arial 10')
btnPlus.bind("<Button-1>", sums)
btnPlus.place(x = 30, y = 60, width = 40,height=25)

btnMult=Button(tk, text = 'x',font='Arial 10')
btnMult.bind("<Button-1>", muls)
btnMult.place(x = 90, y = 60, width = 40,height=25)

btnQuit=Button(tk, text = 'Выход', command=tk.destroy)
btnQuit.place(x = 300, y = 60, width = 70,height=25)
tk.mainloop()

```



В программе для ввода операндов использовались однострочные текстовые элементы **Entry**. Они предназначены для ввода и редактирования текста.

Кроме элементов интерфейса, создано три функции. Функция `is_a_b_numbers()` проверяет, что в текстовых элементах находятся строки, которые можно интерпретировать как числа. Если в поля операндов введены не числовые значения, то сообщение об ошибке помещается в консоль инструкцией `print(...)`.

Функции `sums(ev)` и `muls(ev)` выполняют арифметические операции сложения и умножения, и вызываются при нажатии левой кнопки мыши на соответствующие виджеты. Они также меняют надпись у метки, обозначающей арифметическую операцию, и у метки, содержащей результат. Чтобы не было возможности редактировать ответ, он отображается в элементе **Label**, а не в текстовом поле. Изменить надпись метки можно, изменив значение ее свойства `'text'`. Одним из способов для этого является следующая инструкция: `widget['option']=new_value`. Для меток `lblOp` и `lblRez` соответствующие инструкции имели вид: `lblOp['text']='+'` и `lblRez['text']=str(a+b)`.

Обозначения с квадратными скобками можно также использовать для получения значения свойств. Например, допустима инструкция `var=lblRez['text']`. Но для чтения значения, вводимого пользователем в текстовое поле, лучше применять метод `get()`, который мы и использовали в нашей программе.

Надеемся, что добавление в программу других арифметических операций для вас не составит труда.

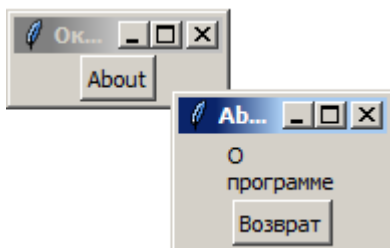
■

Обычно сообщения программ с оконным интерфейсом помещаются не в окно консоли, а отображаются во вспомогательном окне сообщений. Посмотрим, как это можно сделать.

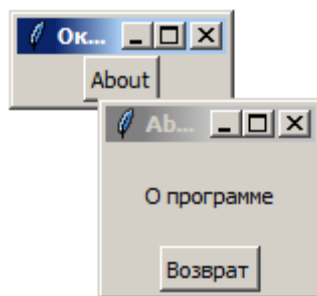
Пример. *Создание окна диалога.*

```
from tkinter import *
def about(ev):
    aux = Toplevel()
    aux.title("About this application...")
    msg = Message(aux, text="О программе")
    msg.pack()
    button = Button(aux, text="Возврат", command= aux.destroy)
    button.pack()
    return
# =====
tk = Tk()
tk.title("Окно сообщений")
btnAbout=Button(tk, text = 'About')
btnAbout.bind("<Button-1>", about)
btnAbout.pack()
tk.mainloop()
```

Клик в главном окне по кнопке «About» вызывает функцию `about(ev)`, которая создает окно сообщений. Кнопка «Возврат» закрывает вспомогательное окно и активирует основное. При этом метод `aux.destroy()` уничтожает только вспомогательное окно `aux`.



Метод `Toplevel()`, используемый в программе, создает диалоговые окна и окна для многооконных приложений. Первым его аргументом указывается родительское окно или виджет. Если этот аргумент опущен, то «родителем» является главное окно приложения. На дочерних окнах, создаваемых с помощью класса **Toplevel**, могут располагаться виджеты. Виджет **Message**, который мы поместили во вспомогательном окне `aux`, является вариантом элемента **Label** и спроектирован для отображения многострочных сообщений. Если не указано иное, то он пытается разместить текст так, чтобы его размеры соблюдали заданные по умолчанию пропорции. Однако вы можете управлять его шириной с помощью опции `width`. Виджет **Message** имеет и другие опции, характерные для элемента **Label**, например, опцию `padx`, задающую минимальный размер пространства по горизонтали вокруг элемента. Измените в предыдущей программе инструкцию `msg=Message(aux, ...)` на следующую: `msg = Message(aux, text="О программе", width=150, padx=18)` и выполните программу. Результат будет следующим.



Как видите, текстовое сообщение отображено по – другому.

При закрытии главного окна дочерние окна также закрываются. Однако закрытие дочернего окна не приводит к закрытию главного.

Окна, создаваемые с помощью класса **Toplevel**, имеют множество методов. Здесь укажем только некоторые из них. Метод `geometry` определяет геометрию окна, и описан нами ранее. Метод `transient(master)` делает вспомогательное окно зависимым от окна `master` в том смысле, что оно (зависимое окно) будет сворачиваться вместе с окном `master`. Например, в последнюю программу после инструкции `aux = Toplevel()` добавьте инструкцию `aux.transient(tk)`. Когда оба окна открыты, сворачивание и разворачивание окна `tk` будет приводить к сворачиванию и разворачиванию окна `aux`. Методы `minsize` и `maxsize` задают минимальный и максимальный размеры окна. Если аргументы этих методов одинаковы, то пользователь не сможет менять размеры окна. Вот, например, как можно ограничить размеры информационного окна из предыдущего примера.

```
def about(ev):
    aux = Toplevel(relief=SUNKEN,bd=8,bg="lightblue")
    aux.minsize(width=200,height=150)
    aux.maxsize(width=400,height=300)
    aux.title("About this application...")
    . . .
```

Запретить изменение размеров окна можно с помощью инструкции `aux.resizable(False,False)`. Два аргумента определяют возможность изменения размеров по горизонтали и вертикали.

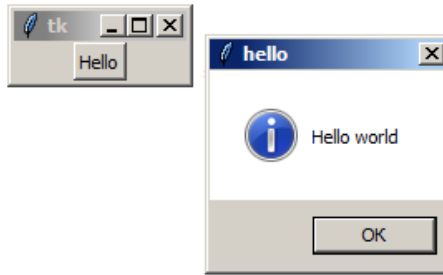
■

Если нужно создать только окно с сообщением, то это можно сделать проще. Рассмотрим еще один вариант программы «Hello world!»

Пример. *Создание окон сообщений.*

```
from tkinter import *
def hello():
    messagebox.showinfo('hello','Hello world')

tk = Tk()
btnHello = Button(tk, text = 'Hello',command=hello)
btnHello.pack()
tk.mainloop()
```



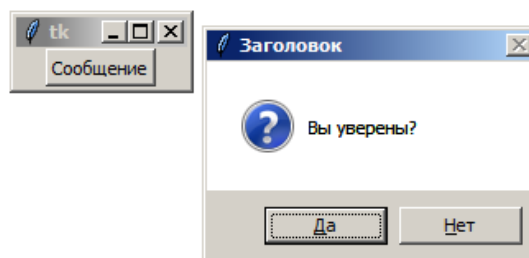
Здесь в теле функции `hello` вызывается конструктор `messagebox`, который создает объект специального окна сообщений, предназначенного для «диалога» программы с пользователем. Этот объект использует следующий синтаксис:

`messagebox.FunctionName(title, message [, options])`,

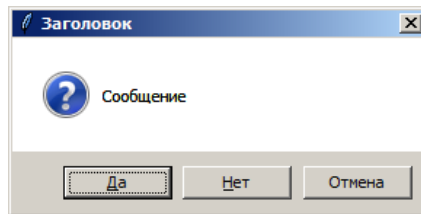
где `FunctionName` одно из допустимых имен: `showinfo`, `askyesno`, `askyesnocancel`, `askokcancel`, `askquestion`, `askretrycancel`, `showwarning`, `showerror`. В зависимости от метода в окне сообщений отображаются различные иконки и кнопки. Все методы возвращают значение, которое можно впоследствии проанализировать. Например, в следующей программе, инструкция `var=messagebox.askyesno('Заголовок', 'Вы уверены?')` открывает окно с вопросом и, в зависимости от ответа пользователя (нажатия кнопки «Да» или «Нет»), переменная `var` принимает значение `True` или `False` (ответ печатается функцией `print(var)` в консоли исполнительной системы).

```
from tkinter import *
```

```
def hello():
    var=messagebox.askyesno('Заголовок','Вы уверены?')
    print(var)
    return
tk = Tk()
btnHello =Button(tk, text = 'Сообщение',command=hello)
btnHello.pack()
tk.mainloop()
```



Метод `messagebox.askokcancel(...)` открывает окно с такой же иконкой в виде вопроса, но на кнопках будут другие надписи («ОК» и «Отмена»). Если вместо метода `askokcancel` использовать метод `askyesno`, то внешний вид окна не меняется, но в переменную `var` будет возвращаться одно из двух значений: `'yes'` и `'no'`. Инструкция `var=messagebox.askyesnocancel('Заголовок', 'Сообщение')` отобразит окно сообщений с тремя кнопками



В зависимости от нажатой кнопки, переменной `var` будет присваиваться одно из трех значений: `True`, `False` или `None`. Проанализируйте самостоятельно значение переменной `var`, возвращаемой оставшимися методами.

Иногда нужно, чтобы пользователь в отдельном окне выполнил текстовый ввод или вводил число. Для этого можно использовать окно простого диалога **`simpdialog`**.

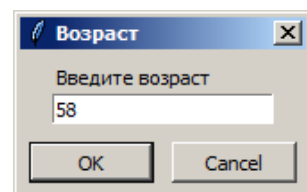
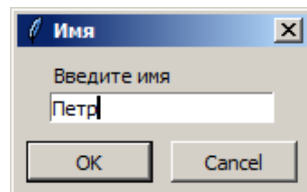
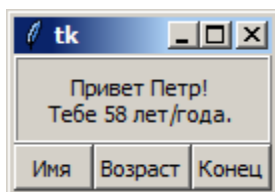
Пример. Окно ввода.

```
from tkinter import *
def cmdname():
    name = simpdialog.askstring("Имя", "Введите имя")
    if name!=None: lblHello["text"]="Привет "+name+'!'
    return

def cmdage():
    age=simpdialog.askinteger("Возраст", "Введите возраст")
    if age!=None:
        lblHello["text"]=lblHello["text"]+'\nТебе %d лет/года.' % age
    return

tk = Tk()
lblHello = Label(tk,relief=RIDGE, borderwidth=2)
lblHello.pack(fill=X, ipady=6)
btnName=Button(tk, text = ' Имя ',command=cmdname)
btnName.pack(side = 'left')
btnAge=Button(tk, text = 'Возраст',command=cmdage)
btnAge.pack(side = 'left')
btnQuit=Button(tk, text = 'Конец',command=tk.destroy)
btnQuit.pack(side = 'left')
tk.mainloop()
```

Основное окно приложения после ввода пользователем имени и возраста показано на следующем рисунке слева. Кнопка с надписью «Имя» открывает окно ввода текстового сообщения, показанное на следующем рисунке в середине, а кнопка «Возраст» открывает окно ввода целого числа, показанное справа.



Для ввода имени использовалась инструкция следующего вида:

```
name=simpledialog.askstring("Заголовок", "Текст")
```

А для ввода целого числа (возраста) использовалась инструкция

```
age=simpledialog.askinteger("Заголовок", "Текст")
```

Если в окне простого диалога нажать кнопку «ОК», то в соответствующую переменную возвращается текст (или целое значение). Если нажимается кнопка «Cancel», то метод возвращает None.

Заметим, что у объекта `simpledialog` есть еще метод `askfloat(...)`, который предназначен для ввода вещественных чисел.

■

Часто необходимо изменить конфигурацию виджета во время выполнения программы. Для этого предназначен метод `configure`, аргументы которого совпадают с опциями конструктора элемента. Если метод `configure` использовать без аргументов, то он возвращает словарь всех свойств (пары `option:value`).

Фактически, сам элемент (виджет) можно трактовать как словарь и обращаться к значениям его свойств с помощью ключей: `widget["свойство"]`. Это относится как к заданию значения свойства, например, `button['text']= value`, так и к его чтению: `value=button['text']`.

Получить значение свойств можно также с помощью метода `widget.cget("option")`. Обратите внимание на то, что значения свойств возвращаются в виде строк, даже если они имеют числовые значения. В таком случае вам придется использовать функции преобразования типа.

Пример. *Изменение цвета и надписи кнопки, после щелчка мыши.*

```
from tkinter import *
from random import randrange
def btn_clicked():
    bgr='#'+'{0:02x}'.format(randrange(0,256,1))+\
        '{0:02x}'.format(randrange(0,256,1))+\
        '{0:02x}'.format(randrange(0,256,1))
    # btn['bg'] = bgr
    # btn['text'] = bgr
    btn.configure(text=bgr, bg=bgr)
root=Tk()
btn = Button(root, command=btn_clicked,fg='white')
btn.pack(fill=X)
btn_clicked()
root.mainloop()
```



Щелчок мыши по кнопке вызывает функцию `btn_clicked()`, в которой случайно сгенерированные целые числа из диапазона 0 – 255 используются в качестве красной, зеленой и синей составляющей цвета. Строка цвета `bgr`

получается конкатенацией символа “#” с тремя форматированными строками вида `'{0:02x}'.format(random_color)`. В них целое десятичное число `random_color` преобразуется к 16-тиричному виду (на это указывает символ “x”). Под него выделяется две позиции (двойка перед “x”), и символом заполнения свободных позиций является 0. Таким образом, строка “02x” означает вывод двухсимвольного шестнадцатеричного числа с ведущим нулем, если он потребуется. Начальный ноль, стоящий перед двоеточием `{0:...}`, является номером используемого аргумента метода `format(...)` (у нас других аргументов нет). Строка цвета `bgr` используется при задании цвета фона кнопки, а также выводится белым цветом на самой кнопке. Инструкция `btn.configure(text=bgr, bg=bgr)` выполняет непосредственные изменения свойств. Ее можно заменить закомментированными двумя инструкциями `btn['bg'] = bgr` и `btn['text'] = bgr`.

■

Иногда требуется выполнить какую – либо функцию с задержкой по времени. Для этого можно использовать метод `after(...)`. Он есть у многих виджетов, и имеет следующий синтаксис:

```
widget.after(time_ms, function[, arg1,...])
```

Первый аргумент `time_ms` задает время задержки в миллисекундах, второй – указывает имя выполняемой функции.

Метод `after` часто оказывается полезным для создания простой анимации. Например, следующая функция `loopproc`, будучи запущенной, вызывает функцию `function()`, выполняющую какие-либо действия (к примеру, что – то рисует), а затем с задержкой запускает себя снова.

```
def loopproc():
    function()
    root.after(delay, loopproc)
```

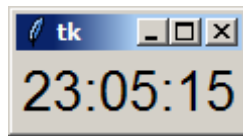
В результате функция `function()` будет выполняться каждые `delay` миллисекунд до тех пор, пока работает программа.

Метод `after_idle(function)` выполняет функцию, указанную в аргументе, после завершения обработки всех событий.

Пример. Часы.

```
from tkinter import *
import time
def tick():
    label['text'] = time.strftime('%H:%M:%S')
    label.after(200, tick)

root=Tk()
label = Label(font='sans 20')
label.pack()
label.after_idle(tick)
# tick() это также работает
root.mainloop()
```

Инструкция `label.after_idle(tick)` запускает на выполнение функцию `tick()`. Здесь вместо этого можно просто использовать вызов функции `tick()`. ■

Вы уже знаете, что элементы управления могут реагировать на множество различных событий. Для сообщения виджету на какое событие, и как он должен реагировать, используется метод `bind(event, имя_процедуры[, "+"])`. Первый аргумент `event` – это строка специального содержания, обозначающая событие. Второй аргумент метода `bind` связывает событие с обработчиком, т.е. сообщает, какая функция будет вызываться при обработке события. Третий необязательный аргумент – строка "+", которая означает, что обработчик события добавляется к уже существующим. Если третий аргумент опущен или равен пустой строке, то указанный обработчик события замещает другие обработчики данного события.

В первом аргументе метода `bind` события записываются с помощью зарезервированных строк. Названия событий заключаются в кавычки и в угловые скобки (знаки `<` и `>`). Вот обозначения для некоторых событий, производимых мышью:

- `"<Button-1>"` или `<1>` – щелчок левой кнопкой мыши;
- `"<Button-2>"` или `<2>` – щелчок средней кнопкой мыши (или колесом);
- `"<Button-3>"` – щелчок правой кнопкой мыши;
- `"<Double-Button-1>"` – двойной щелчок левой кнопкой мыши;
- `"<Motion>"` – движение мыши и т.д.

События, производимые клавиатурой, обозначаются следующим образом:

- нажатие буквенных клавиш можно записывать без угловых скобок, например, `'P'`;
- неалфавитные клавиши обозначаются специальными зарезервированными строками: `"<Return>"` – нажатие клавиши Enter, `"<space>"` – пробел, `"<Key>"` – нажатие любой клавиши на клавиатуре и т.д.
- сочетания клавиш записываются через тире: `"<Control-Shift>"` (одновременное нажатие клавиш Ctrl и Shift) и т.д.

Например, если переменная `root` обозначает корневое окно, то участок кода, описывающий реакцию программы на комбинацию клавиш Ctrl + z может иметь следующий вид:

```
def prog_exit(event):  
    root.destroy()  
  
...  
root.bind('<Control-z>', prog_exit)
```

Нажатие комбинации клавиш Ctrl + z приведет к закрытию приложения.

Вот пример подключения процедуры `procedure(ev)` к событию нажатия комбинации клавиш `Ctrl+f`.

```
widget.bind("<Control-KeyPress-f>", procedure)
```

Использование слова `KeyPress` необязательно.

```
widget.bind("<Control-f>", procedure)
```

Метод `bind` возвращает идентификатор привязки, который можно использовать в функции `unbind()`. Она отсоединяет виджет от события и в качестве аргумента принимает идентификатор, полученный от метода `bind`. Функция `bind()` может возвращать строки `"continue"` и `"break"`, которые используются для продолжения или прекращения обработки других привязок этого события. Если функция ничего не возвращает (т.е. возвращает `None`), то обработка событий продолжается.

Обратите внимание также на то, что если использован метод `bind` окна верхнего уровня, то соответствующее событие будет обрабатываться всеми виджетами этого окна.

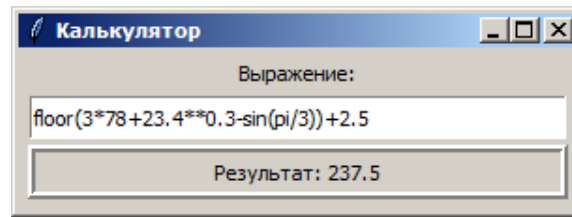
Функция обработчик события, подключаемая с использованием метода `bind`, должна принимать один аргумент – объект класса **Event**. Он имеет много различных атрибутов, которые здесь не рассматриваются.

Пример. Интерактивный калькулятор. Пользователь вводит математическое выражение в однострочном текстовом поле, нажимает клавишу «**Enter**» и получает результат вычисления.

```
from tkinter import *
from math import *
def evaluate(event):
    try:
        res.configure(text = "Результат: " + str(eval(entry.get())))
    except ZeroDivisionError:
        print("Деление на ноль!")
    except NameError:
        print("Отсутствует переменная")
    except SyntaxError:
        print("Синтаксическая ошибка")
    except Exception:
        print("Ошибка ввода")
    return

top = Tk()
top.title("Калькулятор")
Label(top, text="Выражение:").pack(pady=3)
entry = Entry(top)
entry.bind("<Return>", evaluate)
entry.pack(fill=X, ipady=3, padx=6)
res = Label(top, relief=RIDGE, borderwidth=4)
res.pack(fill=X, padx=5, ipady=3, pady=3)
top.bind('<Control-z>', lambda event: top.destroy() )
```

top.mainloop()



Особенностью данной программы является использование встроенной функции `eval(string)`, которая вычисляет строку с кодом и возвращает результат.

```
>>>eval("2*3.5 + 4 * len('Петр')")
23.0
```

Чтобы в калькуляторе можно было использовать элементарные математические функции, в коде мы импортируем модуль **math**.

Признаком завершения ввода является нажатие пользователем на клавиатуре клавиши «**Enter**». Инstrukция `entry.bind("<Return>", evaluate)` подключает функцию `evaluate()` к обработке события нажатия на эту клавишу. При анализе возможных ошибок ввода в функции `evaluate()` выполняется обработка нескольких возможных исключительных ситуаций.

Инструкция `top.bind('<Control-z>', lambda ev: top.destroy())` определяет реакцию программы на комбинацию клавиш **Ctrl – z**. Ввод этой комбинации приводит к завершению работы калькулятора. Чтобы не писать отдельно код для обработки этого события мы использовали лямбда – функцию.

■

Для работы с изображениями в **tkinter** имеется два класса **BitmapImage** и **PhotoImage**. **BitmapImage** представляет собой простое двухцветное изображение, **PhotoImage** - полноцветное изображение. Объекты этих классов прикрепляются к другим виджетам путем установки атрибута `image`. Кнопки, метки, текстовые виджеты и меню имеют этот атрибут, и могут выводить изображения. Размеры кнопок и меток автоматически изменяются в соответствии с размерами изображений. Виджеты **Canvas** (холст) – универсальные поверхности для вывода графики, обсуждаемые в следующем параграфе, тоже могут отображать эти объекты.

Класс **BitmapImage** предоставляет объект для хранения монохромного (двухцветного) изображения. Он состоит из фрагмента кода на языке C, который определяет ширину и высоту рисунка, а также содержит массив данных изображения. Чтобы поместить битовое изображение в Python программу такой фрагмент следует окружить тройными кавычками. Например,

```
BITMAP = """
#define image_width 32
#define image_height 32
static char image_bits[ ] = {
0x00, 0x00, 0xe0, 0x00, 0x10, 0x0e, 0xd2, 0x0d,
. . . . .

```

```
0xe0, 0xff, 0xa0, 0xaa, 0x80, 0x91, 0x00, 0x01  
}; ""
```

Здесь массив `image_bits[]` содержит в шестнадцатеричном формате информацию об изображении. Каждое шестнадцатеричное число, будучи представлено в двоичном формате, кодирует двоичными нулями и единицами пиксели изображения. Два цвета, которые в объекте **BitmapImage** кодируются нулем и единицами, можно задавать с помощью опций `background` и `foreground` (по умолчанию пиксели, кодируемые единицами, отображаются черным цветом, а нулевые – прозрачные). Например,

```
pict=BitmapImage(data=BITMAP,background='gray',foreground='red')
```

Объект класса **BitmapImage** может быть загружен из текстового файла с расширением XBM, имеющего такую же структуру, как и строка C, приведенная выше.

```
pict = BitmapImage(file="bitmap.xbm")
```

Для добавления объекта **BitmapImage** на кнопку (или другой виджет) используется опция `image`.

```
button=Button(top, image=pict[,options])
```

Объекты **BitmapImage** можно использовать в любом виджете, имеющем опцию `image`. Значение опции `image` можно менять, используя метод `config` или оператор `[]` (квадратные скобки). Для чтения текущего значения опции используйте только оператор `[]`.

У многих визуальных элементов имеется опция `bitmap`, аналогичная `image`, но используемая для отображения небольших предопределенных монохромных изображений. Например, для кнопки соответствующая инструкция может иметь следующий вид:

```
button=Button(top, bitmap="hourglass"),
```

где `"hourglass"` имя, определяющее картинку  «песочные часы».

Вы можете поместить на виджет следующие изображения.



Вот имена показанных выше картинок (слева направо): `'error'`, `'gray75'`, `'gray50'`, `'gray25'`, `'gray12'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, и `'warning'`. Можно использовать собственные изображения. Подходит также любой файл в XBM формате. Для этого вместо стандартной строки следует использовать строку `'@'`, за которой следует путь к `xbm` файлу. Если задана опция `image`, то опция `bitmap` игнорируется.

Класс **PhotoImage** используется для хранения данных о цветных растровых изображениях, обычно читаемых из GIF файлов.

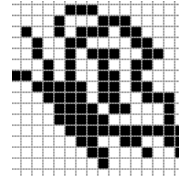
```
photo = PhotoImage(file="image.gif")
```

Объект **PhotoImage** помещается в виджет также, как и **BitmapImage** – с помощью опции `image`. Например,

```
lbl=Label(top, image= photo)
```

Опция `image` имеет приоритет перед опциями `text` и `bitmap`.

Пример. Простая анимация. Два изображения бабочки поочередно перекрывают друг друга, создавая имитацию полета. Изображения имеют размер 77 x 77 пикселей, хранятся в GIF файлах, и показаны на следующем рисунке слева и в середине. Монохромное 16 x 16 пиксельное изображение бабочки, показанное справа, помещается на кнопку.



```
from tkinter import *
import os
def vis():
    global vs,a;
    if vs:
        lbl1.lift(lbl2)
    else:
        lbl2.lift(lbl1)
    vs=not vs
    a+=3
    if a>280: a=-70
    lbl1.place(x = a, y = 40, width = 77,height=77)
    lbl2.place(x = a, y = 40, width = 77,height=77)
    return

def loopproc():
    vis()
    if anim==False: return
    top.after(200,loopproc)

def tick():
    global anim
    anim= not anim
    if anim: top.after(200,loopproc)

BITMAP="""
define image_width 16
define image_height 16
static unsigned char image_bits[] = {
0x00, 0x00, 0xe0, 0x00, 0x10, 0x0e, 0xd2, 0x0d,
0x64, 0x18, 0xa4, 0x33, 0x24, 0x19, 0x2b, 0x16,
0x7a, 0x1b, 0x74, 0x73, 0xf0, 0x46, 0xf0, 0x41,
0xe0, 0xff, 0xa0, 0xaa, 0x80, 0x91, 0x00, 0x01};"""

top = Tk()
top.geometry('280x130')          # размеры окна
top['bg']='white'                # цвет фона окна
```

```

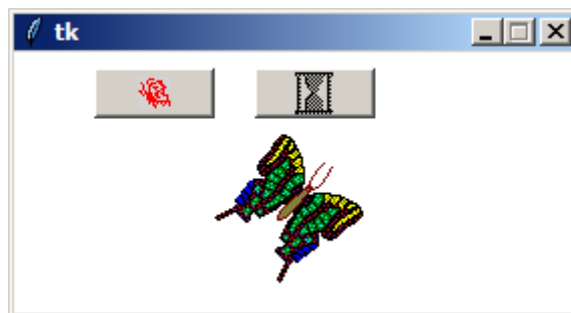
top.resizable(False,False)      # запретить изменение размеров окна
vs=False                        # признак видимости 1-й картинки
a=50                            # начальный сдвиг картинки
anim=False                      # признак работы анимации

pict=BitmapImage(data=BITMAP,background='lightgray',
                  foreground='red')

but1=Button(top, image=pict, command=vis)
but1.place(x = 40, y = 8, width = 60,height=25)
but2=Button(top, bitmap="hourglass", command=tick)
but2.place(x = 120, y = 8, width = 60,height=25)

dr = os.path.dirname(__file__)  # формирование имени каталога файла
filename1=dr+'/Graphics/BFLY1.gif' # полное имя файла картинки 1
filename2=dr+'/Graphics/BFLY2.gif' # полное имя файла картинки 2
img1 = PhotoImage(file=filename1)
img2 = PhotoImage(file=filename2)
lbl1=Label(top, image=img1)
lbl2=Label(top, image=img2)
vis()                            # начать показ со второй картинки
top.mainloop()

```



Правая кнопка с песочными часами включает и выключает анимацию. Левая кнопка меняет порядок картинок (какая сверху, какая снизу), тем самым, выполняет один взмах крыльев. При каждой смене картинок, метки их содержащие, смещаются вправо. В результате, бабочка движется вправо. Когда горизонтальная координата становится достаточно большой, бабочка появляется в окне слева.

Функция `vis()` чередует изображения. Для этого она использует метод `lift(...)` меток, содержащих изображение бабочек. Метод `widget.lift()` (без аргументов) перемещает виджет на вершину оконного стека. Если в аргументе задано имя элемента, то виджет перемещается так, чтобы расположиться поверх переданного в качестве аргумента элемента. Функция `vis()`, используя метод `place(...)`, также сдвигает метки вправо на 3 пикселя.

Функция `loopproc()` вызывает функцию `vis()`, меняющую кадры, и рекурсивно вызывает себя с задержкой в 200 миллисекунд до тех пор, пока переменная `anim` не станет равной `False`. Для вызова какой – либо функции с задержкой используется метод `widget.after(задержка_в_мс, имя_функции)`.


Смена значения переменной `anim` выполняется функцией `tick()`, которая, когда `anim=True`, также выполняет метод `top.after(200, loopproc)`, запускающий процедуру `loopproc()`.

Графические файлы находятся в подкаталоге `\Graphics` каталога с файлами примеров. Переменная (атрибут) `__file__` любого файла содержит его полное имя. Для выделения из него имени каталога мы использовали функцию `os.path.dirname(__file__)`, которая возвращает строку с именем каталога. Добавление к ней строки `'/Graphics/BFLY1.gif'` создает полное имя графического файла. Используемый здесь модуль `os` представляет интерфейс к часто используемым службам операционной системы.

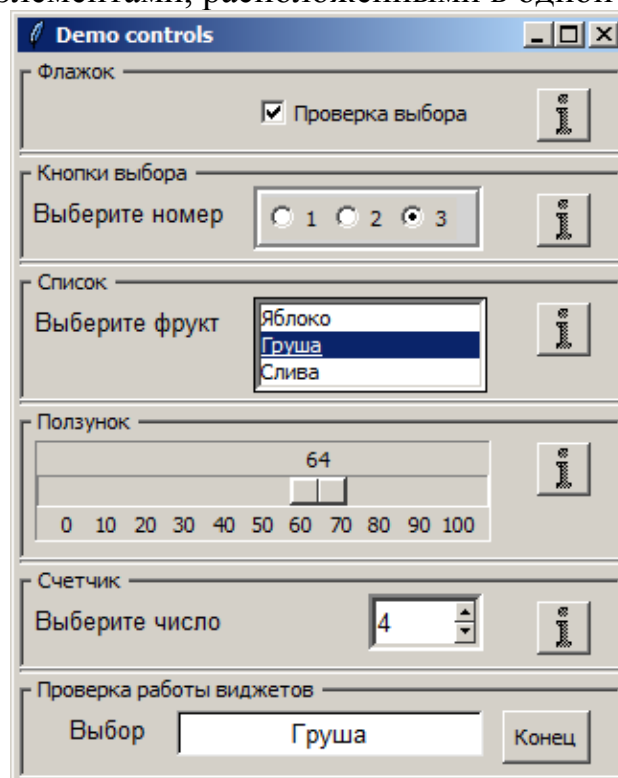
■

Краткое описание некоторых других визуальных элементов библиотеки `tkinter` приведем вместе с описанием программ, демонстрирующих их работу.

Пример. *Пример работы с визуальными элементами `PanedWindow`, `LabelFrame`, `Checkbutton`, `Radiobutton`, `Listbox`, `Scale` и `Spinbox`.*

Окно программы с этими элементами показано на следующем рисунке. Клик по кнопке с пиктограммой  переносит данные из визуального элемента, расположенного рядом с кнопкой, в метку, расположенную в нижней части окна в панели «Проверка работы виджетов».

Код программы будем приводить и пояснять по частям. Каждая часть описывает работу с элементами, расположенными в одной полосе окна.



Программа начинается с определения трех функций обработчиков события нажатия на информационные кнопки в полосах «Флажок», «Кнопки выбора» и «Список». Затем конструируется объект `root` окна приложения, в котором создается визуальный элемент `PanedWindow`. Этот элемент спроектирован как

контейнер, который может содержать произвольное количество панелей, расположенных горизонтально или вертикально. Выравнивание панелей выполняется с помощью опции `orient`. На панелях помещаются другие виджеты или другие такие же элементы **PanedWindow**. Между панелями имеются разграничительные полосы, которые можно перемещать с помощью мыши. Форма этих полосок управляется опциями `sashwidth` и `sashrelief`. Каждая панель может содержать только один виджет. Добавление панелей выполняется вместе с добавлением нового виджета методом `PanedWindow.add(widget)`.

```
from tkinter import *
def fromCheckToLabel():          # реакция на выбор флажка
    if var1.get()==1:
        lblRez['text']='Флажок выбран'
    else:
        lblRez['text']='Флажок не выбран'
def fromRadioToLabel():          # реакция на выбор элемента из группы
    lblRez['text']='Выбрана кнопка '+str(vr.get())
def fromListToLabel():           # реакция на выбор из списка
    lblRez['text']=lstBox.get(lstBox.curselection())
# =====
root=Tk()
root.title('Demo controls')
pw = PanedWindow(root, sashwidth=3,sashrelief=SUNKEN,
                  orient=VERTICAL, width = 316)
pw.pack(fill="both", expand="yes")
```

Для демонстрации работы с виджетами окно программы разделено на полосы, расположенные одна под другой. Для этого панели элемента **PanedWindow** в нашей программе созданы с использованием опции `orient=VERTICAL`. Каждая панель может содержать только один элемент, а нам нужно размещать в них по 2 или 3 виджета. Поэтому на панели мы помещаем контейнерные элементы **LabelFrame**, в которые затем помещаем другие виджеты. **LabelFrame** – простой контейнерный виджет, подобный элементу **Frame**, сочетающий в себе возможности метки (**Label**) и рамки (**Frame**). Он отличается от **Frame** только тем, что на своем контуре может иметь подписи, которые, как обычно, задаются опцией `text`.

В верхней панели (строка виджетов 1) тестируется работа с визуальным элементом **Checkbutton** (Флажок). Группа команд начинается с создания элемента **LabelFrame**, который добавляется на элемент **PanedWindow** инструкцией `pw.add(lf1)`. Затем создаются элементы **Checkbutton** и информационная кнопка, которые помещаются на элемент **LabelFrame**.

```
# ===== строка виджетов 1 (Флажок) =====
lf1=LabelFrame(pw,text=' Флажок ',borderwidth=2,
               relief=SUNKEN, height=50)
pw.add(lf1)
```



```

var1=IntVar()
check1=Checkbutton(lf1,text='Проверка выбора',variable=var1,
                    onvalue=1,offvalue=0, command=fromCheckToLabel)
var1.set(1)
check1.place(x = 108, y = 4, width = 140, height=20)
btnCopyChk1=Button(lf1, command=fromCheckToLabel, bitmap="info")
btnCopyChk1.place(x = 268, y = 2, width = 28, height=28)

```

Checkbutton (флажок) – это виджет, который позволяет отметить «галочкой» определенный пункт в окне. Для хранения его состояния мы используем «переменную слежения» `var1`. Переменные слежения – это объекты специальных классов, которые предназначены для хранения значений состояния различных виджетов. Изменение значения такой переменной ведет к изменению свойства виджета, и наоборот, изменение состояния виджета изменяет значение ассоциированной переменной. Существует несколько классов **tkinter** для «переменных слежения». Для строк используется класс **StringVar**, для целых чисел используется **IntVar**, для дробных чисел – **DoubleVar**, для обработки булевых значений **BooleanVar**. В приведенном выше коде, инструкция `var1=IntVar()` создает переменную `var1`, которая принимает целые значения, и прикрепляется к элементу **Checkbutton** опцией `variable=var1`. Опции `onvalue=1` и `offvalue=0` этого виджета определяют значения, которые принимает переменная `var1`, когда флажок выбран или не выбран.

Функция реакции флажка на изменение его состояния задается опцией `command=fromCheckToLabel`. В теле функции `fromCheckToLabel()` инструкция `var1.get()` получает значение переменной `var1` и сравнивает его с единицей. В зависимости от значения в метке результата `lblTxtRez` отображается соответствующий текст. Эта же функция используется в качестве обработчика события нажатия на информационную кнопку, расположенную рядом с флажком.

У переменных слежения имеется метод `set()`, который позволяет задать им значение. Инструкцией `var1.set(1)` устанавливается значение переменной `var1` в единицу, и вместе с этим устанавливается флажок.

Во второй полосе программы (вторая панель сверху) тестируется работа с радиокнопками (виджетами **Radiobutton**). Для этого в начале соответствующей группы команд создается второй элемент **LabelFrame**, который добавляется на элемент **PanedWindow**. Затем создаются другие элементы интерфейса, помещаемые на элемент **LabelFrame**.

```

# ===== строка виджетов 2 (Радиокнопки) =====
lf2=LabelFrame(pw,text=' Кнопки выбора ',borderwidth=2,
               relief=SUNKEN,height=54)

pw.add(lf2)
lblTxt2=Label(lf2,font='Arial 10',width=12,text='Выберите номер')
lblTxt2.place(x = 4, y = 4, width = 100, height=20)
frmRadio = Frame(lf2,bg = 'lightgray',relief=GROOVE,borderwidth=4)

```

```

frmRadio.place(x = 120, y = 0, width = 120, height=34)
vr=IntVar()
rb1=Radiobutton(frmRadio,text='1',variable=vr,value=1,
                 command=fromRadioToLabel)
rb2=Radiobutton(frmRadio,text='2',variable=vr,value=2,
                 command=fromRadioToLabel)
rb3=Radiobutton(frmRadio,text='3',variable=vr,value=3,
                 command=fromRadioToLabel)

rb1.pack(side = 'left')
rb2.pack(side = 'left')
rb3.pack(side = 'left')
vr.set(1)
btnCopyRadio=Button(lf2,bitmap="info",command=fromRadioToLabel,)
btnCopyRadio.place(x = 268, y = 4, width = 28, height=28)

```

Отличие в создании радиокнопок от флажка состоит в том, что виджеты **Radiobutton** работают группой (выбрать можно только один элемент из группы). Для визуального выделения группы мы использовали рамку (элемент **Frame**). Но фактически группа определяется общим именем переменной слежения `vr`, которая задается опцией `variable=vr` при конструировании элементов. Повторим еще раз – всем элементам группы назначается одна переменная слежения. Она будет содержать одно из значений, которые устанавливаются опциями `value` при создании элементов **Radiobutton**.

После создания радиокнопок и размещения их внутри элемента **Frame** мы инструкцией `vr.set(1)` задаем выбор первой кнопки группы. Функция `fromRadioToLabel()` опциями `command=fromRadioToLabel` задается как обработчик событий выбора радиокнопок и события щелчка по информационной кнопке. Эта функция, используя метод `vr.get()`, выводит в метку результата соответствующий текст.

В третьей панели программы (строка виджетов 3) тестируется работа с элементом **Listbox** (Список). Для этого, как и для других полос, в начале соответствующей группы команд создается элемент **LabelFrame**, помещаемый на новую панель элемента **PanedWindow**. Затем создаются другие элементы интерфейса, помещаемые на элемент **LabelFrame**.

```

# ===== строка виджетов 3 (Список) =====
lfList=LabelFrame(pw,text=' Список ',borderwidth=2,
                  relief=SUNKEN, height=70)

pw.add(lfList)
lblTxtList=Label(lfList,font='Arial 10',width=12,text='Выберите фрукт')
lblTxtList.place(x = 4, y = 4, width = 100, height=20)
lstBox=Listbox(lfList,height=3,width=19,selectmode=SINGLE,
               borderwidth=3)

lst=["Яблоко","Груша","Слива"]
for elem in lst:
    lstBox.insert(END,elem)
lstBox.selection_set(first=0)    # выделяем нулевой элемент списка

```

```
lstBox.place(x = 120, y = 1)
```

```
fromListToLabel2=lambda ev:
```

```
    lblRez.configure(text=lstBox.get(lstBox.curselection()))
```

```
lstBox.bind("<<ListboxSelect>>", fromListToLabel2)
```

```
btnCopyList=Button(lfList, bitmap="info",command=fromListToLabel)
```

```
btnCopyList.place(x = 268, y = 4, width = 28, height=28)
```

Listbox – это виджет, представляющий собой список, из которого пользователь может выбрать один или несколько элементов. Он имеет свойство `selectmode`, которое, при значении `SINGLE`, разрешает выбрать только один элемент списка. Добавление строк списка в нашей программе выполняется методом `lstBox.insert(END, elem)`. Первый аргумент этого метода указывает позицию для добавления, второй – добавляемую строку.

Инструкция `Listbox.selection_set(first=0[,last=None])` выделяет группу элементов в списке. Выделить можно несколько элементов. В аргументе `first` указывается номер первого из выделенных элементов, в аргументе `last` – номер последнего. В примере инструкция `lstBox.selection_set(first=0)` выделяет один элемент списка с номером 0 (не путайте выделенный элемент с активным, который внешне отличим от других подчеркиванием).

Конструктор элемента **Listbox** не имеет опции `command`. Поэтому для подключения обработчика события выделения нового элемента списка кликом мыши мы используем метод `bind()`. Инструкция `fromListToLabel2= lambda ev :`

```
    lblRez.configure(text=lstBox.get(lstBox.curselection()))
```

создает функцию – обработчик с аргументом, а инструкция

```
lstBox.bind("<<ListboxSelect>>", fromListToLabel2)
```

подключает этот обработчик.

Метод `lstBox.curselection()` возвращает индекс выделенного элемента списка, а метод `lstBox.get(index)` возвращает соответствующую индексу строку списка.

К информационной кнопке мы подключаем другую функцию–обработчик `fromListToLabel`. Ее код ничем не отличается от функции `fromListToLabel2`, однако эта функция не имеет аргументов. Напомним, что функции–обработчики, подключаемые методом `bind()` имеют обязательный аргумент класса **Event**, а подключаемые с помощью опции `command`, не должны иметь аргументов.

В четвертой панели программы (строка виджетов 4) тестируется работа с элементом **Scale** (шкала). Как и для других полос, в начале соответствующей группы команд мы создаем элемент **LabelFrame**, добавляем его в элемент **PanedWindow**, а затем создаем другие элементы интерфейса, помещаемые на созданный виджет **LabelFrame**.

```
# ===== строка виджетов 4 (Шкала) =====
```

```
lfScale=LabelFrame(pw,text=' Ползунок ',borderwidth=2,  
                    relief=SUNKEN, height=80)
```

```

pw.add(IfScale)
scl = Scale(IfScale,orient=HORIZONTAL,length=236,from_=0,to=100,
            tickinterval=10,resolution=1, relief=SUNKEN)
scl.set(50)
scl.place(x = 4, y = 1)
getScaleMoveValue=lambda pos : lblRez.configure(text=pos)
scl['command']=getScaleMoveValue
getScaleValue=lambda : lblRez.configure(text=scl.get())
btnCopyScale=Button(IfScale, bitmap="info",command=getScaleValue)
btnCopyScale.place(x = 268, y = 4, width = 28, height=28)

```

Scale (шкала) – это виджет, позволяющий выбрать какое-либо значение из заданного интервала. Внешне элемент представляет собой горизонтальную или вертикальную полосу с разметкой, по которой пользователь может перемещать ползунок. Виджет **Scale** имеет следующие свойства: `orient` – задает ориентацию (горизонтально или вертикально), `length` – задает длину элемента, `from_` и `to` – начальное и конечное значение шкалы, `tickinterval` – интервал, через который отображаются метки шкалы, `resolution` – шаг ползунка. Метод `set(value)` задает начальное положение/значение ползунка на шкале, а метод `get()` возвращает его значение. Процедура, вызываемая каждый раз, когда ползунок перемещается, подключается опцией `command` и имеет один аргумент – новое значение/положение ползунка. В блоке кода, приведенном выше, такая процедура называется `getScaleMoveValue`, имеет один аргумент и подключается к элементу **Scale** инструкцией `scl['command']=getScaleMoveValue`. Функция `getScaleValue` выполняет практически то же действие, но не имеет аргументов. Опцией `command=getScaleValue` она подключается к информационной кнопке во время ее конструирования.

В пятой панели программы (строка виджетов 5) тестируется работа с элементом **Spinbox** (счетчик). В начале группы команд мы создаем элемент **LabelFrame**, добавляем его в элемент **PanedWindow**, а затем создаем другие элементы интерфейса.

```

# ===== полоса виджетов 5 (счетчик)=====
IfSpin=LabelFrame(pw,text=' Счетчик ',borderwidth=2,
                  relief=SUNKEN, height=54)

pw.add(IfSpin)
lblTxtSpin=Label(IfSpin,font='Arial 10',width=12,
                 text='Выберите число')
lblTxtSpin.place(x = 4, y = 4, width = 100, height=20)
spb = Spinbox(IfSpin, from_=0, to=10, width=16,
              borderwidth=4, font='Arial 10')
spb.place(x = 180, y = 0, width = 60, height=30)
getSpbValue=lambda:lblRez.configure(text=spb.get())
spb['command']=getSpbValue
btnCopySpb=Button(IfSpin, bitmap="info",command=getSpbValue )
btnCopySpb.place(x = 268, y = 4, width = 28, height=28)

```

Элемент **Spinbox** (счетчик) представляет сочетание поля ввода с кнопками выбора из фиксированного множества значений. В конструкторе этого элемента можно задать начальное и конечное значение опциями `from_` и `to`, а также шаг изменения опцией `increment`. Можно также задать набор допустимых значений. Например, инструкция

```
spb = Spinbox(lfSpin, values=(1,3,8,5,14,11),  
              width=16, borderwidth=4, font='Arial 10')
```

задаст массив шести допустимых значений.

В коде, приведенном выше, диапазон счетчика равен `[0,10]` с единичным шагом, задаваемым по умолчанию. Процедура элемента **Spinbox**, привязываемая опцией `command`, вызывается каждый раз, когда значение счетчика меняется. В нашем примере ею является функция `getSpbValue`. Для ее привязки мы используем инструкцию с квадратными скобками: `spb['command']=getSpbValue`. Эта же функция используется для обработки клика по информационной кнопке `btnCopySpb`.

Последняя панель окна программы содержит метку, в которую помещаются результаты чтения значений тестируемых элементов, а также кнопку завершения приложения. Соответствующий код приведен ниже.

```
# ===== полоса результирующих виджетов =====
```

```
lfRez=LabelFrame(pw,text=' Проверка работы виджетов ',  
                 borderwidth=2,relief=SUNKEN, height=54)
```

```
pw.add(lfRez)
```

```
lblTxtRez=Label(lfRez,text=' Выбор ',font='Arial 10')
```

```
lblTxtRez.place(x = 4, y = 4, width = 80, height=20)
```

```
lblRez=Label(lfRez,font='Arial 10',bg='white',relief=SUNKEN)
```

```
lblRez.place(x = 80, y = 4, width = 160, height=24)
```

```
btnQuit = Button(lfRez, text=" Конец ", command=root.destroy)
```

```
btnQuit.place(x = 250, y = 4, width = 46, height=28)
```

Завершается код примера командами, подключающими процедуры тестирования виджетов к комбинациям клавиш клавиатуры. Последняя инструкция программы запускает цикл обработки сообщений

```
# ===== комбинации клавиш =====
```

```
root.bind('<Control-c>', lambda ev: fromCheckToLabel())
```

```
root.bind('<Control-r>', lambda ev: fromRadioToLabel())
```

```
root.bind('<Control-l>', lambda ev: fromListToLabel())
```

```
root.bind('<Control-s>', lambda ev: getScaleValue())
```

```
root.bind('<Alt-c>', lambda ev: getSpbValue())
```

```
root.bind('<Control-z>', lambda ev: root.destroy())
```

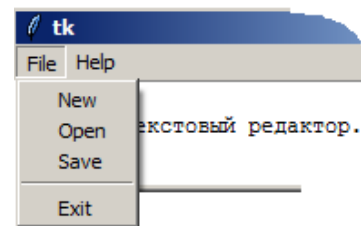
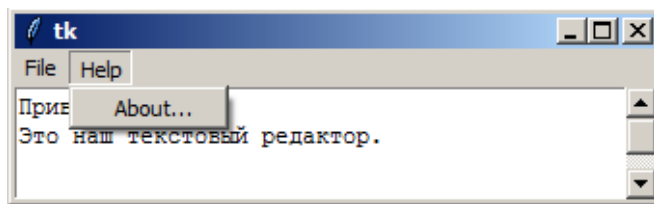
```
root.mainloop()
```

■

Пример. Текстовый редактор. Работа с визуальными элементами **Text, **Scrollbar**, **Menu** и окнами диалога открытия и сохранения файлов.**

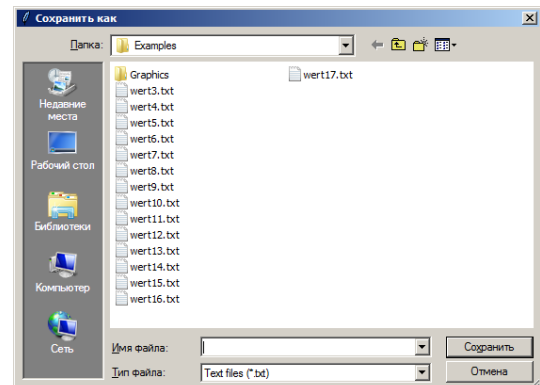
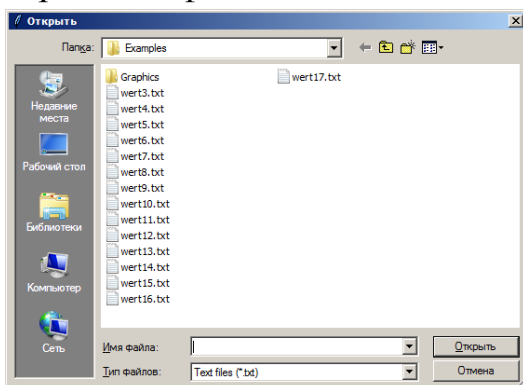
Окно программы показано на следующем рисунке слева. В рабочей части вводится текст, который с помощью меню **File** можно сохранить в файл или загрузить из файла. Меню **File** показано на следующем рисунке справа. Пункт

меню **File->Save** сохраняет текст в файл, а команда **File->Open** загружает текстовый файл в окно программы. Пункт меню **File->New** очищает окно редактора, а команда **Exit** завершает работу программы. Меню **Help->About** выводит окно сообщения.



Код программы будем приводить и пояснять частями, которые выделяются строками комментариев. Начинается код импортированием всех функций модуля **tkinter** и трех дополнительных функций **askopenfile**, **asksaveasfile** и **showinfo** из его подмодулей.

Затем идет код функций, вызываемых командами меню. Функция **about()** вызывается из пункта **Help->About** и создает окно сообщений **messagebox.showinfo(...)**, описанное нами ранее. Функции **openfile()** и **savefile()** начинаются с вызова методов **filedialog.askopenfile()** и **filedialog.asksaveasfile()**, которые вызывают диалоговые окна открытия и сохранения файлов.



Настройка окон выполняется опциями конструктора. Опция **filetypes=(("Text files", "*.txt"), ("All files", "*.*"))** определяет содержимое выпадающего списка «Тип файлов», а опция **defaulttextension=".txt"** определяет, какое расширение имени файла будет использоваться по умолчанию. Методы **askopenfile()** и **asksaveasfile()** возвращают ссылки на файловые объекты **fopen** и **fsave**. Они имеют методы чтения **str=fopen.read()** и записи **fsave.write(str)** файлов. При использовании в окнах диалога кнопки «Отмена» методы **askopenfile()** и **asksaveasfile()** возвращают **None**, что и должно быть проанализировано до того, как в коде будет происходить чтение или запись файла. Функция **fopen.read()** возвращает строку, которая содержит прочитанный из файла текст, а функция **fsave.write(str)** в качестве аргумента принимает строку, которая будет записана в текстовый файл. Эта строка вставляется в виджет **Text** в коде функции **openfile()** и читается из него в коде функции

savefile(). Другие инструкции, используемые в функциях меню, мы опишем ниже.

```
from tkinter import *
from tkinter.filedialog import askopenfile, asksaveasfile
from tkinter.messagebox import showinfo
def about():
    msg="""
        Учебная программа.
        Демонстрация работы визуальных
        элементов Text, Scrollbar и Menu.

        Copyright \u00A9 2016.
        ХНУ им. В.Н. Каразина, г.Харьков."""
    showinfo('О программе',msg)

def newfile():    # очистка окна без запроса о сохранении
    text.delete(0.0, END)

def openfile():
    fopen=askopenfile( mode='r', defaultextension=".txt",
                       filetypes =(("Text files", "*.txt"),("All files", "*.*")) )
    if fopen==None: return
    str=fopen.read()
    text.delete(0.0, END)
    text.insert(END,str)

def savefile():
    fsave=asksaveasfile( mode='w',defaultextension=".txt",
                          filetypes =(("Text files", "*.txt"),("All files", "*.*")) )
    if fsave==None: return
    str=text.get(0.0,END)
    fsave.write(str)
    fsave.close()
```

В окне программы расположены два виджета: **Text** и **Scrollbar**. Элемент **Text** позволяет пользователю ввести любое количество строк текста. Опишем некоторые его возможности. Свойство `wrap` управляет отображением длинных строк. Например, при использовании опции `wrap=WORD` текст внутри виджета будет переноситься по словам. При значении опции по умолчанию `wrap=CHAR` строка, достигая правого края, будет переноситься (разрываться) по любому символу. Для изменения шрифта предназначена опция `font`, которую можно применять к отрывкам текста. Такие отрывки называются тегами и конструируются методом `tag_add()` объекта **Text**. Например, инструкция `text.tag_add('first','3.0','4.11')` создает тег с именем 'first', который начинается в третьей строке первым символом ('3.0') и заканчивается в четвертой строке двенадцатым символом ('4.11'). Первый

аргумент 'first' метода `tag_add` – имя тега. Далее идут индексы начального и конечного символов текста, которые указывают область, к которой тег прикрепляется. Место вставки записывается в виде строки 'line.col', где line – это строка, а col – столбец. При этом нумерация строк начинается с единицы, а столбцов с нуля. Например, первый символ второй строки имеет индекс '2.0', а последний символ пятой строки – '5.end'. Разные области текста могут быть помечены одинаковым тегом (общим именем).

Изменение свойств помеченного тега выполняется методом `tag_config(имя_тега[, опции])`. Например, инструкция `text.tag_config('first', foreground='red', font=('times', 14))` устанавливает в элементе **Text** в области, отмеченной тегом 'first', цвет фона и шрифт текста.

Для чтения текста из элемента **Text** обычно используется метод `get(startindex [,endindex])`. Его аргументы являются строками вида 'line.col', представляющими индексы положения начала и конца читаемого текста. Значение 'end' второго аргумента указывает на то, что текст читается до конца. Метод возвращает текстовую строку.

Метод `insert('line.col' ,string)` вставляет строку `string` в указанную позицию. Метод `delete(startindex[,endindex])` удаляет участок текста. Например, инструкция `text.delete(0.0,END)`, использованная нами в функции меню `newfile()`, удаляет из виджета `text` весь текст.

Чтобы в элементе **Text** прокручивать текст, нужно использовать другой виджет – **Scrollbar** (полоса прокрутки). Он даёт возможность пользователю "прокрутить" другой виджет (текстовое поле, список или холст (**Canvas**)). Для привязки вертикальной полосы прокрутки к текстовому полю выполняются два действия:

- у элемента **Text** устанавливается опция `yscrollcommand=scrollbar.set`;
- у элемента **Scrollbar** устанавливается опция `command=text.yview`.

Когда визуальное представление элемента **Text** меняется, он информирует **Scrollbar** (полосу прокрутки), вызывая его метод `set()`. Наоборот, когда пользователь управляет элементом **Scrollbar**, вызывается метод `yview()` виджета **Text**.

Добавление горизонтальной полосы прокрутки выполняется аналогично, только используются опция `xscrollcommand` и метод `xview`.

```
# ===== создание окна программы =====  
root = Tk()  
text = Text(root, height=3, width=40)  
text.pack(side='left', fill='both', expand='yes')  
scrollbar = Scrollbar(root, command=text.yview)  
text.configure(yscrollcommand=scrollbar.set)  
scrollbar.pack(side='right', fill=Y)
```


После проектирования основного окна мы создаем строку меню. Для этого предназначен визуальный элемент **Menu**. Он используется для создания меню главного окна, выпадающих и всплывающих меню.

Меню главного окна находится под строкой заголовка. Оно состоит из выпадающих меню, которые представляют списки пунктов меню, а также другие вложенные выпадающие меню. Каждый пункт меню представляет собой команду, выполняющую какое-либо действие.

Для создания объекта меню верхнего уровня мы используем инструкцию `menubar=Menu(root)`. Аргументом конструктора `Menu` является идентификатор `root` родительского окна. Конструктор создает объект `menubar`, который инструкцией `root.config(menu=menubar)` добавляется в главное окно `root`. Затем, используя тот же конструктор `Menu`, мы создаем объект выпадающего меню `filemenu`. Теперь функции `Menu` в качестве первого аргумента передается идентификатор меню главного окна, а опция `tearoff=0` отключает отображение пунктирной линии в начале списка пунктов меню.

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

Эта инструкция создает объект меню, который превращается в выпадающее меню после его добавления в строку меню `menubar` методом `menubar.add_cascade(...)`. Добавляемое каскадное меню указывается в опции `menu=filemenu`. В нашем примере инструкция создания выпадающего меню **File** выглядит следующим образом:

```
menubar.add_cascade(label="File", underline=0, menu=filemenu)
```

Опция `label` задает название меню, а опция `underline` указывает в нем номер символа, который будет подчеркнут при нажатии на клавишу ALT (клавишу доступа). Последующие инструкции `filemenu.add_command(...)` добавляют команды в выпадающее меню **File**. Опция `command` метода `add_command` связывает команды меню с функциями-обработчиками. Опция `accelerator` добавляет в пункт меню строку, в которой обычно указывается быстрая комбинация клавиш. Добавляется только строка, а саму функцию, вызываемую этой комбинацией клавиш, вам следует подключить отдельной инструкцией, используя метод `bind()` объекта `root` окна верхнего уровня.

Инструкция `filemenu.add_separator()` добавляет разграничительную линию в выпадающем меню **File**.

Второе выпадающее меню **Help** создается аналогично.

Таким образом, для создания строки меню нужно создать экземпляр класса `Menu`, а затем использовать его в качестве значения атрибута `menu` главного окна. После этого можно создавать другие меню и подключать их в строку меню, используя метод `add_cascade()`.

```
# ===== Строка меню =====
```

```
menubar = Menu(root)
```

```
root.config(menu=menubar)
```

```
# ===== Меню File =====
```

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

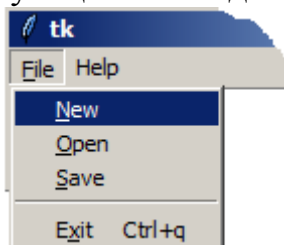
```
menubar.add_cascade(label="File", underline=0, menu=filemenu)
```

```

filemenu.add_command(label="New",underline=0,command=newfile)
filemenu.add_command(label="Open", underline=0,command=openfile)
filemenu.add_command(label="Save", underline=0,command=savefile)
filemenu.add_separator()
filemenu.add_command(label="Exit",underline=1,
                      accelerator="Ctrl+q", command=root.destroy )
# ===== Меню Help =====
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="About...", command=about)
menubar.add_cascade(label="Help", menu=helpmenu)
# =====
root.bind("<Control-q>", lambda event: root.destroy())
root.mainloop()

```

Скажем несколько слов о клавишах доступа, задаваемых опцией `underline`. В Windows 7 подчеркивание символов в названиях меню появляется только после нажатия клавиши ALT. Меню **File** выделяется, и вы можете, нажимая быстрые клавиши, выполнять соответствующие команды.



■

У многих виджетов, в частности у элемента **Text**, имеется еще одна полезная опция `textvariable`. Она определяет переменную слежения, содержащую текст или надпись виджета. Хотя «текстовое» свойство может быть установлено и прочитано в процессе выполнения кода без использования ассоциированных переменных, иногда второй способ связи оказывается более удобным. Например,

```

strvar = StringVar()
ent1 = Entry(root,textvariable = strvar)
ent2 = Entry(root,textvariable = strvar)

```

Здесь содержимое одного текстового поля немедленно отображается в другом, поскольку оба поля привязаны к общей переменной `strvar`.

В завершении этого параграфа скажем несколько слов о других визуальных элементах, не упомянутых нами выше. В модуле **tkinter** имеется подмодуль **ttk**, содержащий усовершенствованные версии описанных выше виджетов: **Button**, **Checkbutton**, **Entry**, **Frame**, **Label**, **LabelFrame**, **Menubutton**, **PanedWindow**, **Radiobutton**, **Scale** и **Scrollbar**. Их можно использовать вместо соответствующих виджетов **tk**. Кроме того, в **ttk** имеется несколько новых виджетов: **Combobox**, **Notebook**, **Progressbar**, **Separator**, **Sizegrip** и **Treeview**. Импорт функций этого подмодуля осуществляется инструкцией `from tkinter.ttk import *`.

С точки зрения программиста главное отличие новых виджетов от старых заключается в том, что у виджетов **ttk** отсутствуют опции для конфигурирования их внешнего вида, который теперь настраивается через темы и стили. В остальном использование виджетов **ttk** аналогично соответствующим виджетам **tk**. В **ttk** имеет четыре встроенных темы: **default**, **classic**, **alt**, **clam**. Кроме того, дополнительно под Windows есть темы **winnative**, **xpnative** и **vista**.

В **tkinter** есть еще подмодуль **tix**, который содержит множество дополнительных реже используемых визуальных элементов: **Balloon**, **ButtonBox**, **CheckList**, **DirList**, **DirTree**, **Grid**, **Meter** и другие (всего около 40 виджетов). Имея представление о программировании виджетов модуля **tk**, знакомство с возможностями новых виджетов не составит труда.

8.3 Векторная графика в tkinter

В этом параграфе мы рассмотрим работу с графическим элементом **Canvas** (холст). **Canvas** позволяет располагать на нем другие виджеты, но в основном предназначен для создания векторной графики.

Для того чтобы создать объект холста необходимо вызвать его конструктор и установить значения его свойств. Например,

```
canvas=Canvas(root,width=480,height=360,bg='#faffff',  
cursor="pencil")
```

Затем, используя один из менеджеров геометрии, «холст» следует разместить в главном окне. После этого можно приступить к рисованию геометрических фигур, форма и положение которых обычно управляется координатами точек. Начало координат (0,0) объекта **Canvas** располагается в верхнем левом углу; направление осей X и Y – вправо и вниз.

Метод **Canvas.create_line()** рисует линию, ломанную или гладкую (в зависимости от значения опции **smooth=0** или **1**). Он принимает последовательность координат точек ($x_1, y_1, x_2, y_2, \dots$). Если **smooth=0** (значение по умолчанию), то будет построена ломаная, проходящая через точки (x_i, y_i). Если **smooth=1**, то будет построена гладкая кривая. Ее форма только управляется положением этих точек, и обычно кривая через них не проходит.

```
canvas.create_line(200,50,300,50,250,100,fill="blue",smooth=1)  
canvas.create_line(0,0,100,100, width=3,arrow=LAST)
```

Каждая пара чисел представляет x и y координаты узла. Опция **fill** задает цвет, а **width** – толщину линии. Опция **arrow** устанавливает стрелку в конце, начале или по обоим концам линии.

Метод **Canvas.create_rectangle()** рисует прямоугольник.

```
canvas.create_rectangle(20,50,100,100,fill="cyan",outline="blue")
```

Первые два числа обозначают координаты верхнего левого угла прямоугольника, вторые – правого нижнего. Опция **outline** задает цвет контура прямоугольника, а опция **fill** – цвет заливки.

Многоугольник создается методом `Canvas.create_polygon()`. Вначале задаются пары (x_i, y_i) координат его вершин, а потом другие опции.

```
canvas.create_polygon([20,120],[200,150],[80,200],fill="yellow")
```

Квадратные скобки при задании координат используются для удобства и могут быть опущены. Свойство `smooth` управляет сглаживанием.

При создании эллипса методом `canvas.create_oval()` используются координаты прямоугольника, в который этот эллипс вписан.

```
canvas.create_oval([360,10],[460,60],fill="gray80")
```

Сектор, сегмент и дуга создаются методом `canvas.create_arc()`. Управляет формой опция `style`. Если она не задана, то рисуется сектор. При `style=CHORD` будет нарисован сегмент, а при `style=ARC` – дуга.

```
canvas.create_arc([200,230],[270,330],start=0,extent=135,  
                 fill="lightgreen")
```

```
canvas.create_arc([210,120],[280,220],start=0,extent=135,  
                 style=CHORD,fill="green")
```

```
canvas.create_arc([340,230],[410,330],start=45,extent=270,  
                 style=ARC,outline="darkgreen",width=2)
```

Если опция `fill` не задана, то будет нарисован контур соответствующей фигуры.

Текст рисуется методом `create_text(x, y, text="строка" [, опции])`. По умолчанию в точке (x, y) располагается центр текстовой надписи. Для смещения точки привязки используется опция `anchor` (якорь). Ее значение "w" (west – запад) разместит по указанной координате левую границу текста. Другие значения: n, ne, e, se, s, sw, w, nw. Если букв, задающих привязки две, то первая определяет горизонтальную привязку, а вторая – вертикальную.

Пример. *Рисование графических фигур на холсте Canvas.*

```
from tkinter import *  
import math  
root = Tk()  
root.title("Векторная графика")  
cnvs=Canvas(root, width=480,height=360,bg= '#faffff')  
cnvs.pack(expand=1, fill='both')  
# ===== использование графических методов =====  
cnvs.create_text(20,10,text="20,10", font=("Helvetica", "10") )  
cnvs.create_text(460,350 ,text=" 460 ,350 " )  
cnvs.create_line([[20,10],[340,50],[460,350]],fill="blue",smooth=1,  
                width=3)  
cnvs.create_line( [[20,10],[340,50],[460,350]] ,fill="#ff7777" ,  
                smooth=0, width=2)  
cnvs.create_line( [[475,250],[460,350]] ,fill="black" ,  
                width=4, arrow=LAST)  
# залитый прямоугольник  
cnvs.create_rectangle(20,50,100,100,fill="cyan",outline="blue")  
# контур прямоугольника  
cnvs.create_rectangle(30,60,90,90,outline="magenta")
```

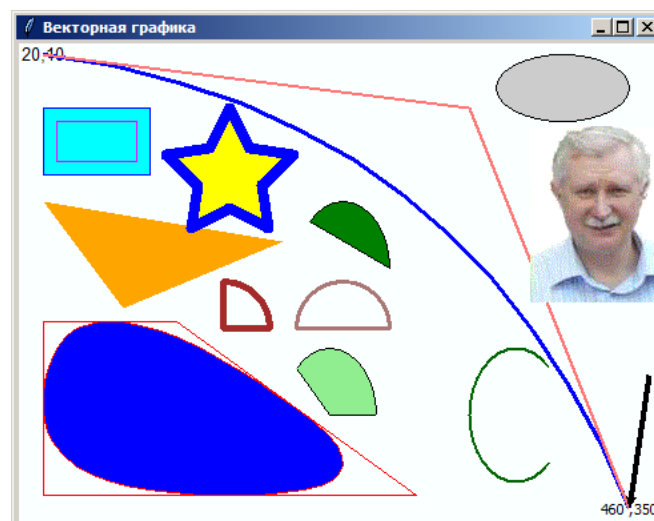
```

# многоугольники
cnvs.create_polygon([20,120],[200,150],[80,200],fill="orange")
cnvs.create_polygon([20,210],[120,210],[300,340],[20,340],
                    outline="red",smooth=0, fill='#faffff')
cnvs.create_polygon([20,210],[120,210],[300,340],[20,340],
                    outline="red",smooth=1, fill='blue')

# рисование звезды; x0, y0 – координаты центра, clr – цвет,
# rout, rin – радиусы описанной и вписанной окружности
def star(x0, y0, rout, rin, clr):
    a=-0.1*math.pi
    krd=[]
    for i in range(5):
        krd.append([x0+rout*math.cos(a),y0+rout*math.sin(a)])
        krd.append([x0+rin*math.cos(a-0.2*math.pi),
                    y0+rin*math.sin(a-0.2*math.pi)])
        a-=0.4*math.pi
    cnvs.create_polygon(krd,fill=clr,outline="blue",width=8)
star(160, 100, 50, 25, 'yellow')
cnvs.create_oval([360,10],[460,60],fill="gray80") # эллипс
cnvs.create_arc([200,230],[270,330],start=0,extent=135,
                fill="lightgreen")
cnvs.create_arc([210,120],[280,220],start=0,extent=135,
                style=CHORD,fill="green")
cnvs.create_arc([340,230],[410,330],start=45,extent=270,
                style=ARC,outline="darkgreen",width=2)
cnvs.create_arc([120,180],[190,250],start=0,extent=90,
                outline="brown", width=5)
cnvs.create_arc([210,180],[280,250],start=0,extent=180,style=CHORD,
                outline="#aa7777", width=3)

# отображение фотографии
photoauthor = ".\\Graphics\\author.gif"
imgauthor = PhotoImage(file=photoauthor)
cnvs.create_image(440,130,image= imgauthor)
root.mainloop( )

```



На виджете **Canvas** можно размещать точечную графику. Для этого используется метод `create_image(x0,y0, image=PhotoImage,...)`. В примере выше мы поместили один такой GIF файл в окно программы.

Обратите внимание на две фигуры в левом нижнем углу окна. Обе построены методом `create_polygon()` по одному и тому же множеству точек. Но при построении многоугольника использована опция `smooth=0` (0 – значение по умолчанию), а синяя «клякса» построена с использованием опции сглаживания `smooth=1`.

■

Все графические методы возвращают идентификатор графической фигуры, который впоследствии можно использовать. Например, если прямоугольник создан инструкцией

```
rect=cnvs.create_rectangle(20,50,100,100,fill="cyan"),
```

то удалить его можно командой `Canvas.delete(rect)`. Удалить все нарисованные фигуры можно командой `Canvas.delete(ALL)`.

Использовать идентификаторы графических фигур можно также для сдвига, изменения свойств или даже для изменения формы. Для этого предназначены методы – модификаторы.

Для сдвига используется метод `Canvas.move(figure, deltaX, deltaY)`, где `figure` – идентификатор фигуры, `deltaX`, `deltaY` – смещения вдоль осей X и Y. Для изменения свойств фигуры (цвета, толщины линии и т.д.) можно использовать метод `Canvas.itemconfig(figure[,опции])`. Метод `Canvas.coords(...)` изменяет координаты «характеристических» точек фигуры.

Пример. *Смещение, изменение формы, цвета и толщины линии геометрической фигуры.* В примере мы рисуем овал. Клавиши управления курсором (→, ←, ↑, ↓) двигают его по холсту. Комбинация клавиш CTRL-f случайным образом меняет цвет и толщину, а комбинация CTRL-s задает случайный размер овала. Для последних двух команд в программе предусмотрены также кнопки.

```
from tkinter import *
import random as rnd
def rndcolor():          # случайный цвет
    bgr='#'+'{0:02x}'.format(rnd.randrange(0,256,1))+\
        '{0:02x}'.format(rnd.randrange(0,256,1))+\
        '{0:02x}'.format(rnd.randrange(0,256,1))
    return bgr

def moveoval(event):     # смещение овала
    if event.keysym=='Right': cnvs.move(ovl,10,0)
    elif event.keysym=='Left': cnvs.move(ovl,-10,0)
    elif event.keysym=='Up': cnvs.move(ovl,0,-10)
    elif event.keysym=='Down': cnvs.move(ovl,0,10)
```

```

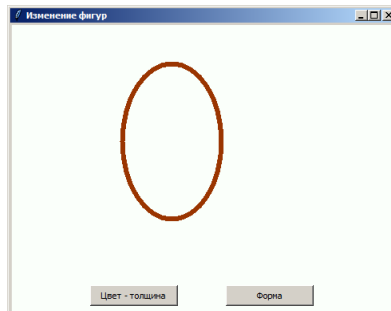
def cnfgeoval(event):      # изменение цвета и толщины линии
    cnvs.itemconfig(ovl,outline=rndcolor(), width=rnd.randint(1,5))

def sizeoval(event):      # изменение ширины и высоты овала
    crds=cnvs.coords(ovl)
    crds[2]=crds[0]+rnd.randint(50, 200)
    crds[3]=crds[1]+rnd.randint(50, 200)
    cnvs.coords(ovl,tuple(crds))

root = Tk()
root.title("Изменение фигур")
cnvs=Canvas(root, width=480,height=360,bg= '#fafffa')
cnvs.pack(expand=1, fill='both')
ovl=cnvs.create_oval([20,20],[120,80],outline="blue", width=3)

butCnf=Button(cnvs, text="Цвет - толщина",
              command=lambda : cnfgeoval(0))
cnvs.create_window(100, 340, window=butCnf, width = 110,
                  height=26, anchor='w')
butSize=Button(cnvs, text="Форма", command=lambda : sizeoval(0))
cnvs.create_window(270, 340, window=butSize, width = 110,
                  height=26, anchor='w')
root.bind('<Key>', moveoval)
root.bind("<Control-KeyPress-f>", cnfgeoval)
root.bind("<Control-s>", sizeoval)
root.bind('<Control-z>',(lambda event:root.destroy()))
root.mainloop( )

```



Функция `rndcolor()` возвращает строку – значение случайного цвета. Функция `moveoval(event)` анализирует аргумент `event` на предмет определения, какая клавиша клавиатуры нажата. Если это одна из клавиш управления курсором (\rightarrow , \leftarrow , \uparrow , \downarrow), то вызывается метод холста `move()`, который смещает овал в нужном направлении. Функция `cnfgeoval(event)` с помощью метода холста `itemconfig()` меняет цвет и толщину эллипса, используя для них случайные значения. Функция `sizeoval()` меняет ширину и высоту прямоугольника, в который вписан эллипс. Вначале инструкция `crds=cnvs.coords(ovl)` определяет список `crds` четырех чисел – координат описанного прямоугольника. Затем два последних числа из этого списка корректируются случайным образом. В конце инструкция `cnvs.coords(ovl,tuple(crds))` присваивает новые координаты овалу.

Обратите внимание на то, что для списка `crds` потребовалось преобразование его в кортеж.

Функции `moveoval()`, `cnfgeoval()`, `sizeoval()` используются в качестве обработчиков событий нажатия на клавиши клавиатуры. Для двух последних функций также созданы лямбда – оболочки без аргумента, которые используются в опциях кнопок, меняющих случайным образом цвет и толщину, а также размер эллипса.

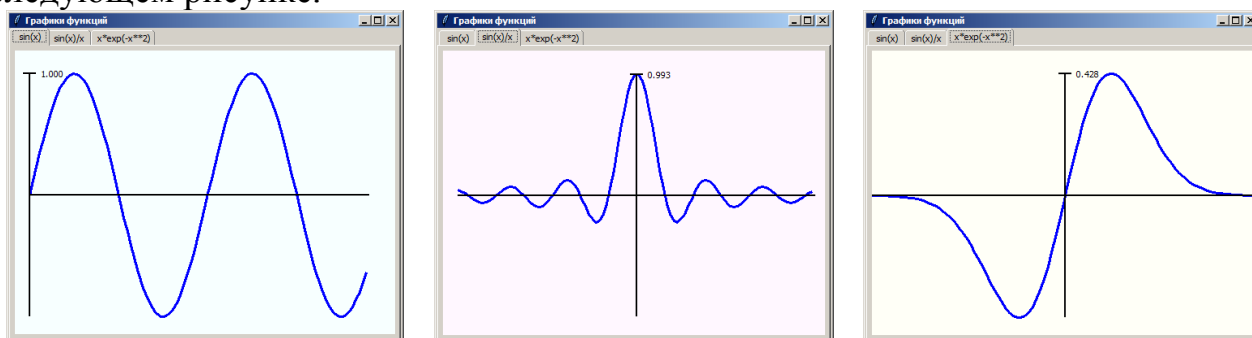
Чтобы установить на холсте кнопки «Цвет – толщина» и «Форма» мы использовали метод `create_window(position, ...)`. Его можно использовать вместо упаковщиков для размещения виджетов. Этот метод немного похож на упаковщик `place(position, ...)`, поскольку имеет сходные с ним аргументы `position`.

■

Рассмотрим пример построения графиков функций с использованием виджета **Canvas**.

Пример. Графики функций. Создадим окно заданного размера и установим на нем виджет **Notebook**. Он представляет собой набор окон, которые имеют закладки. Щелчок мыши по одной из закладок открывает соответствующее окно. Для работы с виджетом **Notebook** следует импортировать модуль `tkinter.ttk`.

На закладках виджета **Notebook** разместим холсты **Canvas**. На первом холсте нарисует график функции $\sin x$. На втором – график функции $\frac{\sin x}{x}$. На третьем – график функции $x \cdot \exp(-x^2)$. Вид трех закладок показан на следующем рисунке.



Ниже приведен код примера, который чередуется поясняющим текстом.

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

def makecanvas(txt, bgclr): # добавление панели Notebook с холстом
    cnvs=Canvas(nb, background=bgclr)
    nb.add(cnvs, text=txt, padding=3)
    return cnvs

root = Tk()
root.title('Графики функций')
```



```

nb = Notebook(width=480, height=360)
nb.pack(expand=1, fill='both')
# создание трех холстов на закладках виджета Notebook
cnvs1=makecanvas(' sin(x) ', '#f7fff')
cnvs2=makecanvas(' sin(x)/x ', '#fff7ff')
cnvs3=makecanvas(' x*exp(-x**2)', '#ffff7')

```

Обычно каждая панель **Notebook** состоит из контейнерного элемента, содержащего другие виджеты. В нашем примере таким контейнером является холст. Добавление новой панели с холстом выполняется методом `Notebook.add(canvas[, заголовок, опции_панели])`. В примере мы используем опцию `padding`, которая добавляет в панели пустое пространство вокруг холста. У нас метод `Notebook.add(...)` используется в функции `makecanvas(...)`, каждый вызов которой создает новую панель с холстом и возвращает объект холста.

Для управления панелями элемента **Notebook** имеется множество свойств и методов. В частности, к панели, открытой в данный момент, можно обратиться с помощью инструкции `Notebook.selected()`. С атрибутами и методами этого виджета вы можете познакомиться самостоятельно по справочной системе.

Основной код построения графиков собран в функции `drawFunc(...)`.

```

# =====
# функция построения на объекте холста canvas графика непрерывной
# функции fun. График строится на отрезке [a,b],
# kx,ky - масштабные множители по осям
# положение начала координат (0,0) в точке холста (xo,yo)
# аргумент exclude - список X координат исключаемых точек
def drawFunc(canvas,fun,a,b,kx,ky, exclude=None):
    points=[ ]
    num=500    # количество точек
    for n in range(num):
        x=a+(b-a)/num*n
        if exclude!=None:
            if x in exclude: continue
        y=fun(x)
        pp=(xo+kx*x,yo-ky*y)    # положительное направление оси Y вверх
        points.append(pp)       # список пар (xi,yi) точек кривой
    canvas.create_line(points,fill="blue",smooth=0,width=3)    # график
    xAxe=[(xo+a*kx,yo),(xo+b*kx,yo)]
    canvas.create_line(xAxe,fill="black",width=2)    # ось X
    my=max([abs(e[1]-yo) for e in points])    # максимальная амплитуда
    yAxe=[(xo,yo-my*1.05),(xo,yo+my*1.05)]
    canvas.create_line(yAxe,fill="black",width=2)    # ось Y
    maxy=min([e[1]-yo for e in points])
    gorMark=[(xo-5,yo+maxy),(xo+5,yo+maxy)]
    canvas.create_line(gorMark,fill="black",width=1) # засечка на оси Y

```

```

# текстовая отметка на оси Y
canvas.create_text(xo+28,yo+maxy,text="{0:.3f}".format(-maxy/ky))
Завершает программу код, в котором функция drawFunc(...) вызывается три
раза. Каждый раз функции передается идентификатор холста соответствующей
панели и имя функции, график которой будет строиться. Перед вызовом
задается точка холста (xo,yo), в которой будет располагаться начало
координат нового графика.
# =====
xo,yo=20,180      # экранное положение начала координат
drawFunc(cnvs1,math.sin,0,12,35,150)    # график функции sin(x)

def g(x):
    return math.sin(x)/x
def f(x):
    return x*math.exp(-x**2)
# график функции sin(x)/x (исключить из рассмотрения точку x=0)
xo,yo=240,180     # экранное положение начала координат
drawFunc(cnvs2,g,-20,20,11,150,[0])
# график функции x*exp(-x**2)
xo,yo=240,180     # экранное положение начала координат
drawFunc(cnvs3,f,-3,3,80,350)
root.mainloop()

```

8.4 Графика matplotlib в окнах tkinter

Как показано в предыдущем параграфе, модуль tkinter позволяет создавать приложения с оконным интерфейсом, содержащими векторную графику. В частности, использование графических функций виджета **Canvas** позволяет строить графики функций. Однако возможности модуля matplotlib значительно совершеннее и желательно совмещать оконные приложения tkinter с графическими функциями matplotlib. Эта возможность реализована с помощью «составного» холста **FigureCanvasTkAgg** — специального класса, который наследует многие методы виджета **Canvas**, и добавляет возможности использования графических функций модуля matplotlib. Класс **FigureCanvasTkAgg** импортируется из модуля `matplotlib.backends.backend_tkagg`. Вместе с ним часто импортируется класс **NavigationToolbar2TkAgg**, который представляет собой панель управления графикой matplotlib. Она показана на следующем рисунке и обычно всегда присутствует в графических окнах matplotlib.



Ее наличие в ваших окнах tkinter необязательно.

В следующем примере приведен простой код, демонстрирующий встраивание графики matplotlib в программу с оконным интерфейсом.

Пример 1. График синусоиды, построенной функциями модуля *matplotlib.pyplot* в приложении *tkinter*.

Ниже приведен код примера, который чередуется поясняющим текстом.

```
from tkinter import *
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
root = Tk()
frm=Frame(root)
# строим график синусоиды функциями matplotlib.pyplot
x = np.linspace(0,2*np.pi)
fig = plt.figure(facecolor='white')
ax = fig.add_subplot(111)
ax.plot(x, np.sin(x),'-rh',linewidth=3, markersize=5,
        markerfacecolor='b', label=r'$\sin x$')
ax.grid(color='b',linewidth=1.0)
plt.legend(fontsize=12)
```

В первых строках кода программы выполняется импортирование необходимых модулей и функций. Следом за этим создается объект окна **root** и контейнерный элемент **Frame**, внутри которого мы позже разместим «составной» холст **FigureCanvasTkAgg** с графикой *matplotlib*.

Затем идет код *matplotlib* построения графика. Инструкцией `fig=plt.figure(facecolor='white')` создается объект рисунка и в нем инструкцией `ax = fig.add_subplot(111)` создается графическая область **Axes**. В графической области функциями модуля *matplotlib.pyplot* рисуется график (или несколько графиков), но метод `show()` не вызывается. Инструкция

```
canvasAgg=FigureCanvasTkAgg(fig, master=frm)
```

создает объект класса «составного» холста **FigureCanvasTkAgg**. Его конструктор в качестве первого аргумента принимает ссылку на объект рисунка **Figure**, использованного при создании графики *matplotlib*. С помощью опции `master=widget` задается родительски виджет (в нашем случае рамка **Frame**), в котором будет находиться «составной» холст. В результате устанавливается «связь» между графическим окном `fig` (размещенным в памяти) и холстом, размещенным внутри родительского виджета `frm`. После этого метод `show()` «составного» холста включает отображение графики *matplotlib*.

```
canvasAgg.show()
```

Нам остается корректно разместить холст внутри окна приложения.

```
canvas=canvasAgg.get_tk_widget() # получить объект стандартного холста
canvas.pack(fill=BOTH, expand=1) # используем упаковщик
```

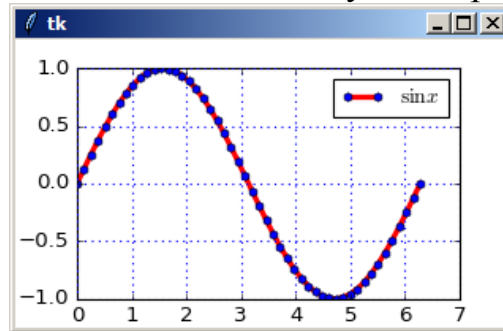
Метод `canvasAgg.get_tk_widget()` возвращает ссылку на объект «стандартного» холста **Canvas**, лежащего «в основании» класса составного

холста. Полученная ссылка `canvas` используется при размещении холста упаковщиком `pack()`.

Оставшиеся две инструкции программы размещают контейнер – рамку внутри главного окна и запускают цикл обработки сообщений.

```
frm.pack(fill=BOTH, expand=1)
root.mainloop()
```

Результат работы программы показан на следующем рисунке.



Как видите, окно нашей `tkinter` программы содержит всю `matplotlib` графику, которую мы захотели нарисовать: кривую, маркеры, легенду и сетку. Для простоты мы не стали размещать другие `tkinter` виджеты в нашем приложении.

■

Замечание 1. Функции низкого уровня библиотеки `matplotlib` образуют ее исполнительную систему (`backend`). Имеется несколько различных систем, которые, в зависимости от потребностей пользователя, можно менять. Вот названия некоторых: `'GTK'`, `'GTKAgg'`, `'QtAgg'`, `'Qt4Agg'`, `'TkAgg'`, `'WX'`, `'WXAgg'`, `'GTK3Agg'`, `'agg'`. Стандартной системой является `'QtAgg'` и, обычно, нет надобности в ее изменении. Но иногда полезно переключиться на другую систему. Такая потребность возникает, в частности, когда мы пытаемся выводить графику `matplotlib` в окнах `tkinter`. Если примеры этого параграфа у вас не будут работать, добавьте в начале их кода следующие инструкции:

```
import matplotlib
matplotlib.use('TkAgg') # выбор исполнительной системы matplotlib
```

Они должны быть выполнены до того, как будет импортирован модуль `matplotlib.pyplot`. Вызов `use()` после импортирования `pyplot` не даст результата.

Замечание 2. Если вы создаете Python программы в Windows XP, то, вероятно, примеры этого параграфа у вас работать не будут.

Таким образом, для создания графиков `matplotlib` на виджете `tkinter` используется следующий набор инструкций:

```
fig = plt.figure(...)
ax = fig.add_subplot(111)
ax.plot(...) # рисование графиков
# ... другие графические функции ...
```

```
canvasAgg = FigureCanvasTkAgg(fig, master=виджет)
canvasAgg.show()
```

Вы рисуете график на объекте `figure`. Потом передаете `figure` первым аргументом конструктору `FigureCanvasTkAgg(figure, master=frm)`, а он, используя объект `figure`, создает холст на родительском виджете `master`.

В следующем примере мы повторяем эту последовательность шагов, создавая трехмерное изображение сферы.

Пример 2. Сфера.

Ниже приведен код рисования сферы, который чередуется поясняющим текстом.

```
from tkinter import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
root = Tk()
frm=Frame(root, relief=RIDGE,borderwidth=4)
frm.pack(fill=BOTH, expand=1)
```

Программа начинается с импортирования необходимых модулей и создания главного окна программы. Затем мы рисуем трехмерную сферу, используя трехмерную графическую область `Axes3D`.

```
fig = plt.figure(figsize=(4,4))
ax = Axes3D(fig)
# рисуем сферу по ее параметрическим уравнениям
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
X = np.outer(np.cos(u), np.sin(v))
Y = np.outer(np.sin(u), np.sin(v))
Z = np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(X,Y,Z,rstride=4,cstride=4,
                color='lightgreen',linewidth=1)
```

После создания графического окна `fig` с графической областью `ax` в памяти, мы создаем объект класса «составного» холста **FigureCanvasTkAgg**, который связывается с объектом `fig` и родительским виджетом – рамкой `frm`.

```
canvasAgg=FigureCanvasTkAgg(fig, master=frm)
```

Затем «составной» холст отображается методом `show()` и размещается менеджером `pack()`.

```
canvasAgg.show() # необязательно
canvasAgg.get_tk_widget().pack(side='top', fill='both',
                                expand=1, padx=4, pady=4)
```

Чтобы трехмерное изображение можно было вращать мышкой, используем инструкцию `ax.mouse_init()`, которая подключает события мыши к графической области рисунка `ax`. Эта инструкция должна вызываться после создания составного холста `canvasAgg`.

```
ax.mouse_init() # подключение событий мыши
```

В конце примера мы создаем кнопку завершения приложения и вызываем цикл обработки сообщений `mainloop()`.

добавление кнопки «Выход»

```
def progquit():
```

```
    root.quit()          # stops mainloop
```

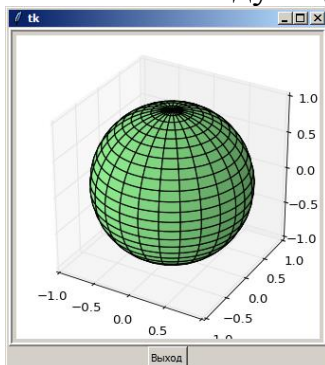
```
    root.destroy()       # используется в Windows
```

```
button = Button(master=root, text='Выход', command=progquit)
```

```
button.pack(side=BOTTOM)
```

```
root.mainloop()
```

Результат работы программы показан на следующем рисунке.



Как мы говорили ранее, в программу можно добавить кнопки управления рисунком. Для этого используются инструкции

```
toolbar = NavigationToolbar2TkAgg(canvasAgg, виджет)
```

```
toolbar.update()
```

Первый аргумент конструктора `NavigationToolbar2TkAgg` указывает на составной холст с `matplotlib` графикой, которой будут управлять кнопки панели. Второй аргумент обычно указывает имя виджета (или главного окна), в котором находится этот холст.

Холст, создаваемый конструктором `FigureCanvasTkAgg()`, можно использовать для рисования `matplotlib` графики и векторной графики одновременно. Однако нужно учитывать одну особенность, которая состоит в том, что векторную графику следует создавать в процедуре обработки сообщения `<Configure>`. Соответствующее событие происходит, когда изменяется размер, положение или внешний вид виджета. Например, оно происходит тогда, когда окно приложения открывается после удаления перекрывающего окна другого приложения. В результате, инструкция `canvas.bind('<Configure>', myredraw, '+')`

должна добавлять к уже существующим процедурам перерисовки `matplotlib` графики (они создаются вместе с классом **`FigureCanvasTkAgg`**) вашу процедуру перерисовки векторной графики.

Пример 3. Совместное использование *matplotlib* и векторной графики.

Ниже приведен код примера, который чередуется поясняющим текстом.

```
from tkinter import *
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg,
NavigationToolbar2TkAgg
```

Программа начинается с импортирования необходимых модулей и функций. Далее мы создаем две функции. В первой из них (mydraw) вызываются графические функции стандартного холста canvas. Вторая (myplotcode) – демонстрирует использование различных двумерных графических функций модуля matplotlib.pyplot.

```
def mydraw(event): # векторная графика
    canvas.create_line([[0,0],[800,800]],fill="blue",width=3)
    canvas.create_line([[0,0],[800,800]],fill="blue",width=3)
    canvas.create_polygon([300,20],[340,150],[500,200],fill="orange")
    canvas.create_rectangle(150, 250, 220, 320, fill="cyan")
    canvas.create_image(30,100,image= imgauthor) # отображение фото
```

```
def myplotcode(): # графика matplotlib
    x = np.linspace(0,2*np.pi)
    fig = plt.figure(facecolor='white')
    ax = fig.add_subplot(111)
    ax.plot(x, np.sin(x),'-rh',linewidth=3, markersize=12,
            markerfacecolor='b', label=r'$\sin x$')
    ax.plot(x, -x*np.exp(-x**2/4),'-go',linewidth=3, markersize=12,
            markerfacecolor='y', label=r'$-x e^{\{-x^2\}}$')
    ax.grid(color='b',alpha=0.5,linestyle='dashed',linewidth=0.5)
    plt.legend(fontsize=12)
    plt.title( 'Functions graphs')
    plt.xlabel('X-ось абсцисс',{'fontname':'Times New Roman'})
    plt.ylabel('Ордината',{'fontname':'Times New Roman'})
    return fig
```

```
root = Tk()
root.title('Графики функций')
frm=Frame(root)
fgr=myplotcode()
canvasAgg=FigureCanvasTkAgg(fgr, master=frm) # составной холст
canvasAgg.show()
```

Затем мы создаем объект окна root и контейнерный элемент **Frame**. Вызов функции myplotcode() создает matplotlib графику и возвращает объект графического окна fgr. Потом fgr передается первым аргументом конструктору FigureCanvasTkAgg(). Он создает объект составного холста на родительском виджете frm. Инstrukция canvasAgg.show() отображает matplotlib графику.

```
canvas=canvasAgg.get_tk_widget() # объект стандартного холста
canvas.pack(fill=BOTH, expand=1)
frm.pack(fill=BOTH, expand=1)
# создание объекта imgauthor для отображение фотографии
```

```
photoauthor = ".\\Graphics\\author.gif"
imgauthor = PhotoImage(file=photoauthor)
```

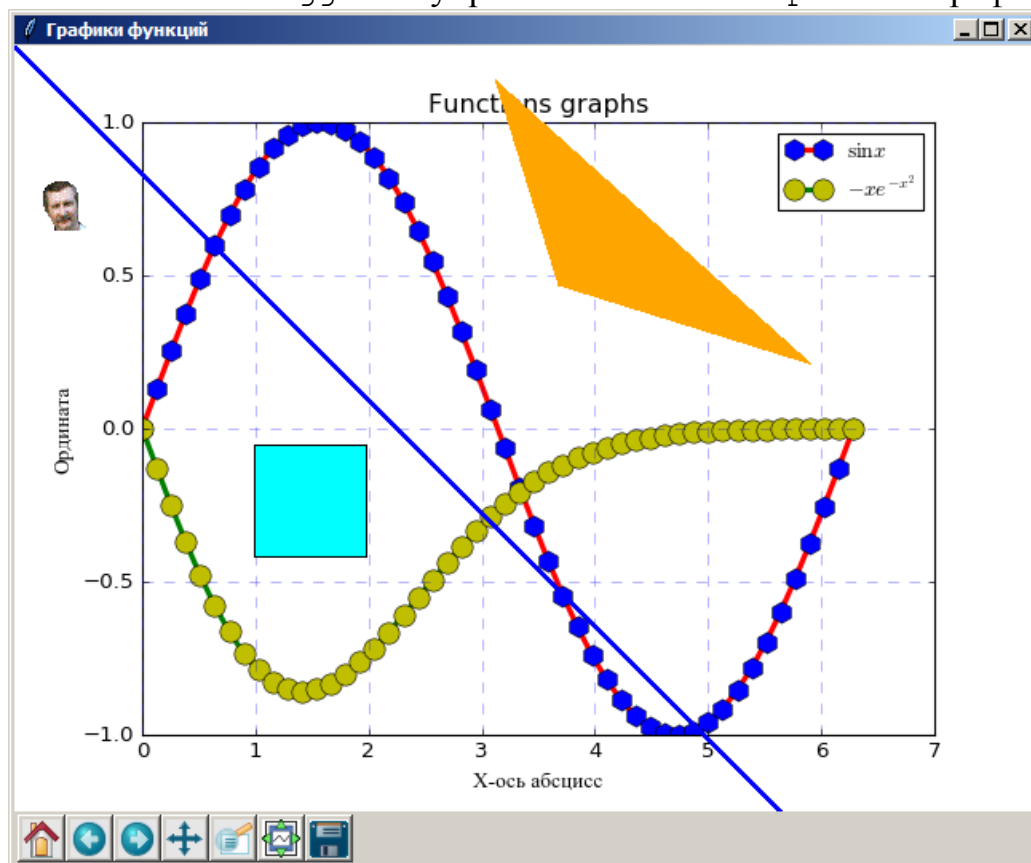
Используя метод `canvasAgg.get_tk_widget()`, мы получаем ссылку `canvas` на объект «стандартного» холста **Canvas**. Она используется при его размещении упаковщиком `pack()`. Следом размещаем виджет рамку и создаем объекта `imgauthor`, который используется в функции `mydraw()` при отображении фотографии. Инструкция `canvas.bind('<Configure>', mydraw, '+')`

добавляет функцию `mydraw()` к обработке события `<Configure>`, которое первый раз происходит при открытии окна программы, а затем каждый раз, когда окно меняет размер, положение или становится видимым после смещения или удаления перекрывающего окна другой программы.

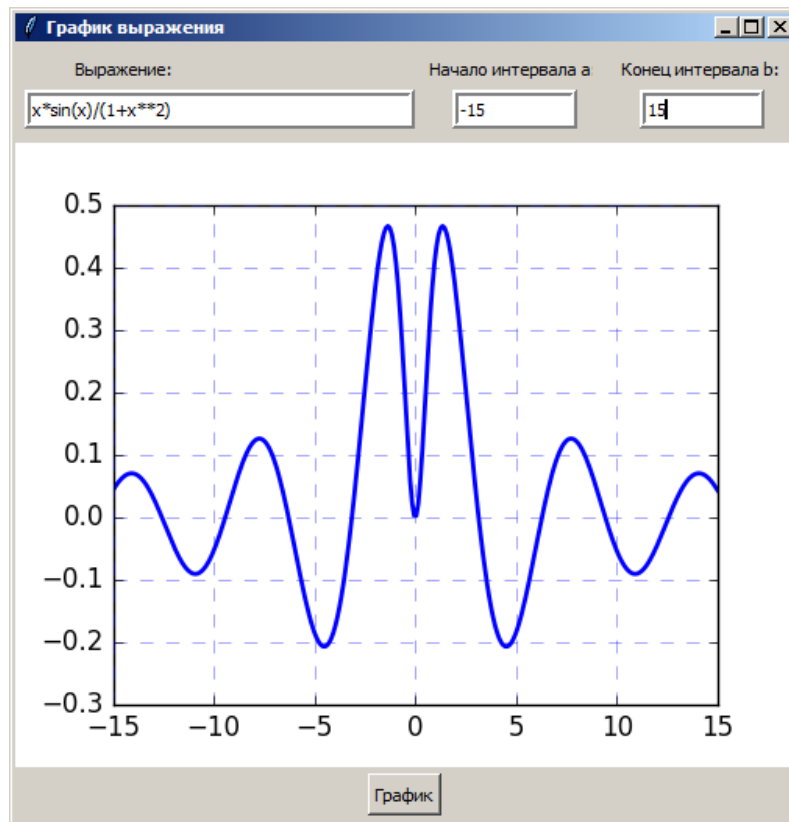
В завершении мы добавляем стандартную панель с кнопками управления `matplotlib` графикой.

```
toolbar = NavigationToolbar2TkAgg(canvasAgg, root )
toolbar.update()
root.mainloop()
```

Окно программы показано на следующем рисунке. Попробуйте менять его размер, а также протестируйте работу кнопок панели `NavigationToolbar2TkAgg`. Они управляют только `matplotlib` графикой.



Пример 4. *Интерактивное построение графика выражения.* В примере показана возможность совместного использования графики `matplotlib` и элементов управления `tkinter`. Окно программы приведено на следующем рисунке.



В верхней части окна расположено текстовое поле (элемент **Entry**), предназначенное для ввода выражения относительно переменной x , график которого нужно построить. В два текстовых поля справа вводятся значения параметров a и b , задающие интервал $[a, b]$ изменения аргумента x . Нажатие клавиши **Enter** или кнопки «График» строит кривую.

Ниже приведен код примера, который чередуется поясняющим текстом.

```
from tkinter import *
from numpy import *
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from tkinter.messagebox import showerror
import warnings
```

Программа начинается с импортирования необходимых модулей и функций.

```
def evaluate(event):
    try:
        mystr=entry.get()
        exec('f = lambda x:'+ mystr,globals())
        a=float(strA.get())
        b=float(strB.get())
        X=linspace(a,b,300)
        Y=[f(x) for x in X]
        ax.clear() # очистить графическую область
```

```

ax.plot(X, Y, linewidth=2)
ax.grid(color='b',alpha=0.5,linestyle='dashed',linewidth=0.5)
canvasAgg.draw() # перерисовать «составной» холст
return

```

except: # реакция на любую ошибку

```
showerror('Ошибка',"Неверное выражение или интервал [a,b].")
```

Далее мы создаем функцию `evaluate`. Она рисует график кривой и вызывается при нажатии на клавишу **Enter**. Инструкция `mystr=entry.get()` читает выражение (формулу) из виджета `entry` в строковую переменную `mystr`. Функция `exec(strcode,...)` рассматривает свой первый аргумент – строку `strcode` как набор команд Python и выполняет их. В нашем случае строка `'f = lambda x: '+ mystr` содержит код лямбда – функции аргумента `x`, вычисляемой по формуле, которая содержится в строке `mystr`. Вторым аргументом функции `exec(...)` определяет пространство имен, в котором будут располагаться переменные создаваемого кода. В результате выполнения инструкции `exec('f = lambda x: '+ mystr,globals())` функция `f(x)` будет размещена в пространстве имен модуля.

С текстовыми виджетами `entryA` и `entryB`, в которые вводятся границы интервала $[a,b]$ изменения независимой переменной, мы связали переменные слежения `strA` и `strB` (см. код далее). Две последующие инструкции преобразуют содержимое этих переменных в числовые значения a и b .

Выражения, вводимые пользователем в текстовом виджете `entry`, а также текстовые поля a и b могут содержать синтаксические или другие ошибки. Поэтому описанные выше инструкции следует окаймлять оператором `try ... except`, используемым для обработки исключительных ситуаций. В нашем примере реакция на любую ошибку пользователя состоит в отображении окна сообщений `showerror(...)`.

Используя значения a и b , мы создаем массив `X` значений аргумента и массив `Y` значений функции `f`. Перед последующим построением графическая область очищается от графика, нарисованного ранее. Инструкция `ax.plot(X, Y, linewidth=2)` создает новый график, а инструкция `ax.grid(...)` рисует координатную сетку. Графическая область `ax` располагается в графическом окне `fig`, которое находится на «составном» холсте `canvasAgg` (см. код далее). После перерисовки графической области `ax` следует обновить содержимое этого холста командой `canvasAgg.draw()`.

```

def evaluate2(event): # чтобы кнопка отжималась при ошибке
    root.after(100,evaluate,event)

```

Функция `evaluate2` является копией функции `evaluate`, но запускается с задержкой 100 миллисекунд. Она вызывается при нажатии кнопки «**График**», а задержка нужна для того, чтобы процедуры перерисовки кнопки успевали ее «отжать», если в процедуре `evaluate` сгенерирована исключительная ситуация (ошибка).

После создания функций идет код построения окна приложения.

```
root = Tk()
root.wm_title("График выражения")
Следующая инструкция
warnings.filterwarnings("error")
```

указывает исполнительной системе Python интерпретировать предупреждения как ошибки. Без этой команды некоторые ошибки генерируют предупреждения (warnings), а не исключительные ситуации. Мы хотим все предупреждения и ошибки обрабатывать единообразно в обработчике исключительных ситуаций. Функция `filterwarnings("error")` находится в модуле `warnings`, который мы импортировали в начале программы. Иногда вам захочется игнорировать все предупреждения, тогда вы можете использовать эту функцию с аргументом `"ignore"`.

Далее идут инструкции создания интерфейса программы. Мы создаем рамку **Frame** с метками и текстовыми виджетами, предназначенными для ввода выражения и переменных a и b . К текстовым виджетам прикрепляем процедуру `evaluate()` обработки события `<Return>` нажатия на клавишу **Enter**.

```
frameUp=Frame(root,relief=SUNKEN, height=64)
frameUp.pack(side=TOP,fill=X)
Label(frameUp, text="Выражение:").place(x = 20, y = 4,
                                         width = 100,height=25)
Label(frameUp, text="Начало интервала a:").place(x = 250, y = 4,
                                                  width = 140,height=25)
Label(frameUp, text="Конец интервала b:").place(x = 370, y = 4,
                                                  width = 140,height=25)
entry = Entry(frameUp,relief=RIDGE,borderwidth=4)
entry.bind("<Return>", evaluate)
entry.place(x = 6, y = 30, width = 250,height=25)

strA=StringVar()
strA.set(0)
entryA = Entry(frameUp,relief=RIDGE,borderwidth=4, textvariable=strA)
entryA.place(x = 280, y = 30, width = 80,height=25)
entryA.bind("<Return>", evaluate)
```

```
strB=StringVar()
strB.set(1)
entryB = Entry(frameUp,relief=RIDGE,borderwidth=4, textvariable=strB)
entryB.place(x = 400, y = 30, width = 80,height=25)
entryB.bind("<Return>", evaluate)
```

Для текстовых виджетов `entryA` и `entryB` также создаются переменные слежения `strA` и `strB`. После этого создается графическое окно **Figure**, которое помещается на «составной» холст **FigureCanvasTkAgg**.

```
fig = Figure(figsize=(5, 4), dpi=100, facecolor='white')
ax = fig.add_subplot(111)
canvasAgg = FigureCanvasTkAgg(fig, master=root)
canvasAgg.show()
```

```
canvas=canvasAgg.get_tk_widget()
canvas.pack(fill=BOTH,expand=1)
```

Затем создается кнопка **Button** построения графика, к которой присоединяется процедура `evaluate2` обработки «клика».

```
btn=Button(root, text = 'График')
btn.bind("<Button-1>", evaluate2)
btn.pack(ipady=2, pady=4,padx=10)
```

Завершается код двумя стандартными инструкциями. Одна подключает процедуру `destroy()` завершения приложения к комбинации клавиш **CTRL-z**. Другая – запускает цикл обработки сообщений.

```
root.bind('<Control-z>', lambda event: root.destroy() )
root.mainloop()
```

■

Эта короткая программа демонстрирует только основные возможности совместного использования библиотеки `tkinter` и графических возможностей библиотеки `matplotlib`. Надеемся, что читатель сможет доработать эту программу под свои потребности.

Заключительные замечания.

В пособии приведены некоторые сведения о языке программирования Python и дано введение в его «научные возможности». Однако многие «научные» пакеты и функции из уже описанных модулей даже не были упомянуты. Автор надеется продолжить работу над пособием и, вероятно, многие темы будут добавлены. Однако в первой части, существенные изменения могут произойти только в ее последней шестой главе: «Научные вычисления с пакетом `SciPy`». Во второй части добавлений, вероятно, будет больше.