

Не заполнены данные профиля педагога. [Заполнить данные профиля.](#)

Вы не установили свою фотографию. Установить её вы можете на своей странице. [Установить фотографию.](#)

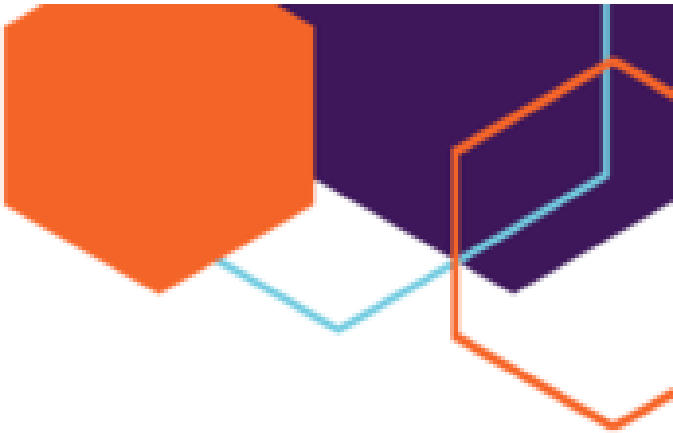
🔔 Хотите получать уведомления от нашего сайта?



Включить Не сейчас

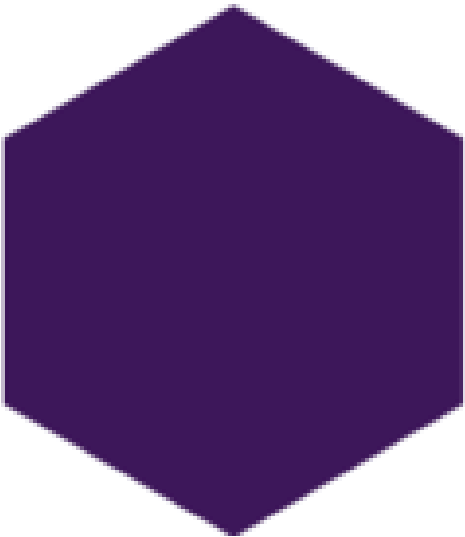
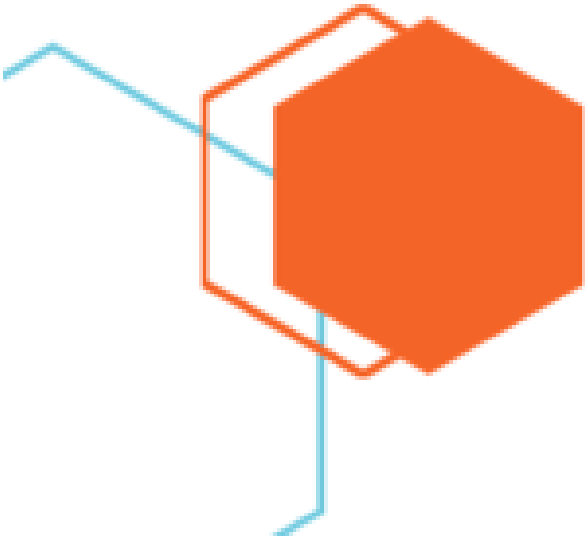
← [Информационные материалы](#) / [Методика проекта ЮК](#) / [Курсы и поурочные планы](#) / [\(09\)10-12\(13+\) лет](#) / [В мире Python](#) / [Python 2 модуль: Python-основы ООП](#) / Уроки

Уроки 🎓 Образовательный уровень: 1



Python

ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО
ПРОГРАММИРВАНИЯ



Урок 1

Объекты. Классы. Поля.

Мы начинаем новый курс по изучению языка Python – Основы объектно-ориентированного программирования (ООП). Все современные языки объектно-ориентированные и, в целом, вся сфера программирования построена на принципе ООП.

Эта тема очень важна, потому что, все современные инструменты программирования основаны на этом принципе и, не обладая знаниями принципа ООП, вы просто не сможете пользоваться этими инструментами. Программист без ООП – это певец без голоса или хоккеист без клюшки.

Итак, давайте разбираться...

Что значит - объектно-ориентированный? Ответ находится вокруг нас. Все в нашем мире является объектно-ориентированным и, вообще, весь мир вокруг нас и все что в нем находится, от элементарных частиц и атомов до вселенной является объектно-ориентированным.

Давайте разберем это понятие – «Объектно-ориентированный». Оно состоит из слова – «Объект» и слова – «Ориентированный», то есть, можно сказать, направленный на то, что все что нас окружает является объектами или воспринимается как объекты. То есть любой предмет, любые сооружения, явления – все это является объектом.

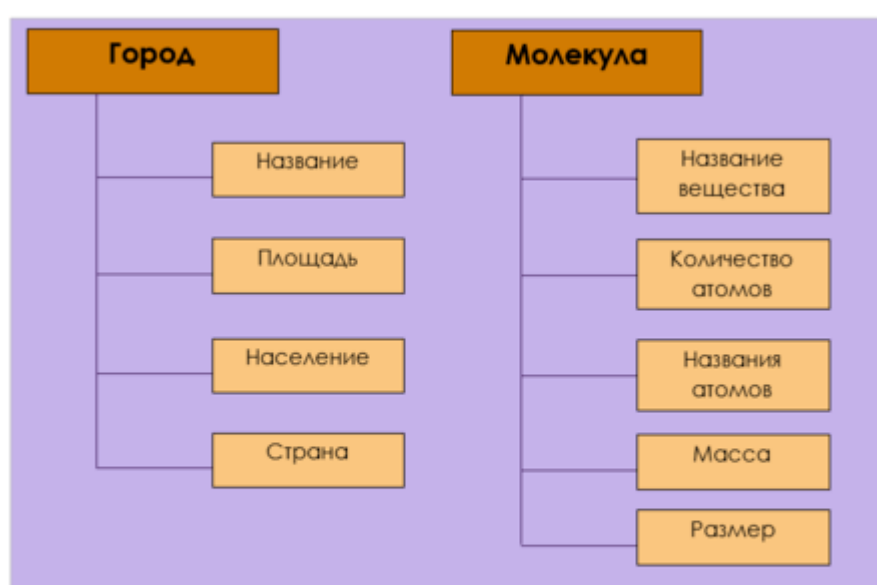
Примеры объектов: человек, планета, город, компьютер, воздух, молекула, ветер.

Но принцип ООП, заключается не только в том, что все вокруг нас – это объекты. Еще он заключается в том, что у каждого объекта есть свои свойства.

Пример: у объекта – Город есть такие свойства как: название, площадь, население, дата основания, регион, в котором он находится и т. д. Объект молекула имеет следующие свойства: название вещества, которое состоит из этих молекул, количество атомов, из которых состоит молекула, названия этих атомов, масса, размер, геометрическая структура.

Давайте представим это в виде схемы (рисунок 1)

Рисунок 1



! Проблемная задача: Составить схему какого-либо объекта с его свойствами (как на рисунке 1).

10 мин

Можно сказать, что такая схема описывает, что из себя представляет объект, но только не какой-то конкретный, а вообще, любой. Схема объекта – Город, описывает, что собой представляет любой город, по ней мы можем понять, что город – это некий объект, который имеет какое-то название, у него есть население (то есть в нем живут люди), он занимает какую то площадь и еще, город входит в состав определенной страны. Возможно, это не полное описание, что собой представляет такой объект, как город, но для понимания сути принципа ООП достаточное.

Итак, у нас есть схема, описывающая любой город, то есть - все города. И с помощью такой схемы, мы можем уже определить какой-нибудь конкретный город (рисунок 2).



На рисунке 2 с помощью схемы объекта мы отобрали уже конкретный объект – город Саратов. Также можем и любой другой город отобразить.

! Проблемная задача:

10 мин

Используя придуманную ранее схему объекта, изобразить 2-3 конкретных объекта.

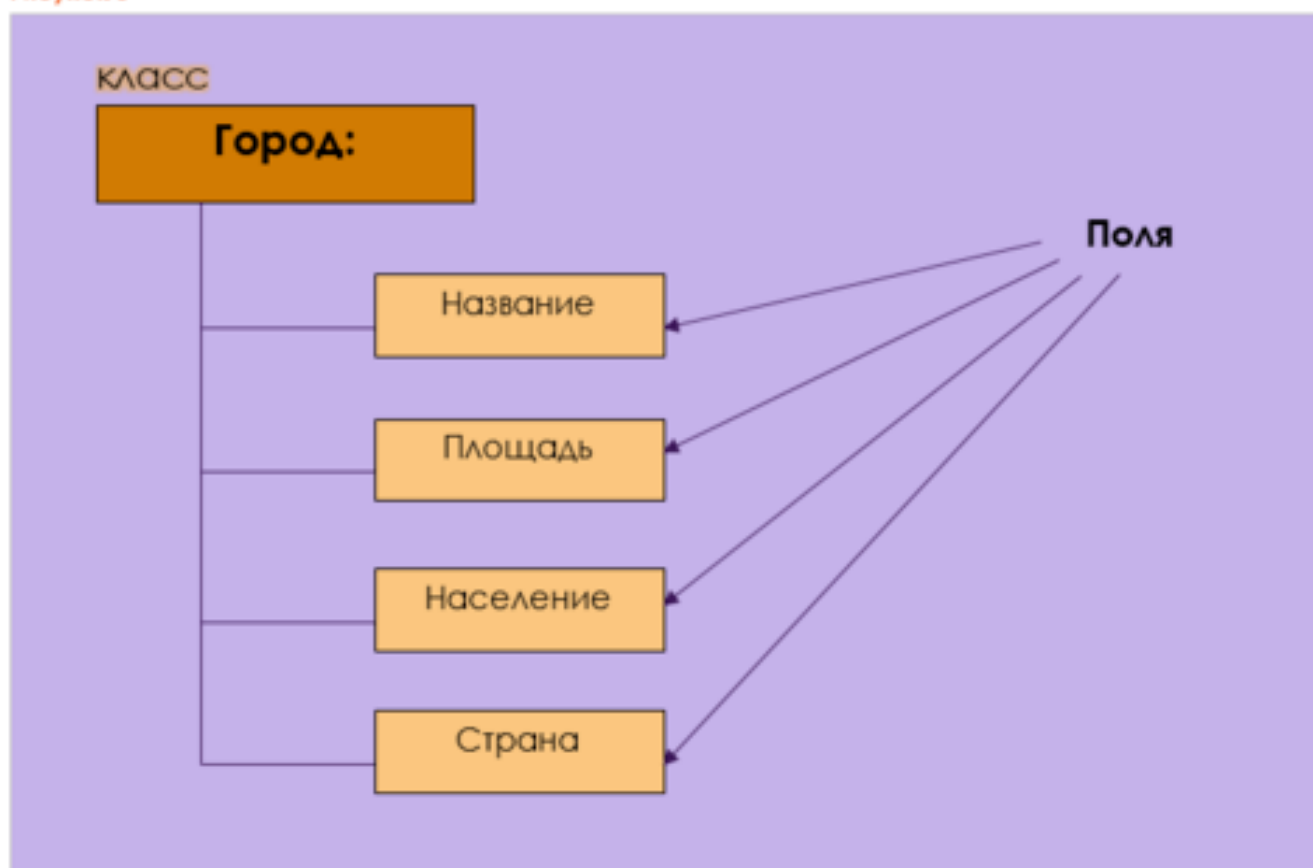
Как же все это используется в программировании?

В программировании такая схема объекта – называется класс. Слово класс в данном случае имеет такой же смысл как класс в биологии.

Например, в биологии есть класс – Птицы, он описывает в целом всех птиц или, можно сказать, описывает такой объект, как «Птица». Наша схема тоже описывает некий объект или все экземпляры такого объекта, поэтому будем называть такую схему – Класс, имея ввиду что-то схожее с понятием «класс» из биологии.

А свойства класса, в программировании, называются «Поля». То есть, например, свойство объекта «Город» – **население** – это поле в классе «Город» (рисунок 3).

Рисунок 3



Пора переходить к делу, создадим класс в Python.



указываются поля (свойства) класса.

Пример:

```
class Processor:
```

```
    name = "
```

```
    frequency = 0
```

```
    numcore = 0
```

В примере мы создали класс Процессор (это который в компьютере) и описали три его свойства: название (**name**), частота (**frequency**) и количество ядер (**numcore**). Но мы не просто указали эти свойства, а сразу присвоили им значения. Это потому, что эти свойства, по сути, являются переменными (ячейками памяти) и мы обязаны задать им какие-то значения по умолчанию.

Итак, мы описали класс Процессор, то есть схему объекта «Процессор»: у него есть имя, частота работы и количество ядер. И мы уже знаем, что класс описывает все объекты, то есть наш класс Процессор описывает все процессоры, а нам необходим конкретный процессор. В Python конкретный объект или экземпляр класса создается также, как и переменная (ячейка памяти) – пишется имя переменной, в нашем случае имя объекта, далее ставится знак "=" и потом указывается название класса.

Пример:

```
procIntel = Processor
```

Мы создали экземпляр класса Processor и назвали его – **procIntel**.

Чтобы использовать его поля (свойства) или, иначе говоря, обратиться к ним, необходимо написать название нашего экземпляра, а через точку указать имя поля (свойства).

Пример:

```
print(procIntel.name)
```

У нас уже получилась такая программа:

```
class Processor:
```

```
    name = "
```

```
    frequency = 0
```

```
    numcore = 0
```

```
procIntel = Processor
```

```
print(procIntel.name)
```

Запустим ее....

Вывелась пустая строка, это потому, что мы выводим на печать значение поля **name**, а оно по умолчанию равно пустой строке.

Попробуем изменить его, присвоить какое-нибудь значение, напишем команду вывода на экран **print** и снова запустим программу.

```
procIntel.name = 'Intel Core i3'
```

```
print(procIntel.name)
```

Вывелось:

! Проблемная задача:

Создать свой класс с 3-5 полями. Создать 2 экземпляра этого класса, задать значения полей экземпляров и вывести их на экран. Далее программа должна запросить у пользователя значение какого-нибудь поля, изменить это поле и снова вывести это поле на экран.

Урок 2

Методы

На прошлом занятии мы с вами познакомились с классами и полями (свойствами) классов. Также мы научились создавать экземпляры классов – конкретные объекты с конкретными свойствами.

Сегодня мы познакомимся еще с одним важным понятием – это «методы».

Методы – это функции объектов, то есть это то, что выполняет какое-то действие. Разберем на примере. Возьмем такой класс, как **Процессор**, который мы создали на прошлом уроке и добавим в него метод (функцию). Так-как метод – это функция, а с функциями вы уже знакомы, то и создается он точно также, как и функция – с помощью служебного слова **def**. Так, а какой метод будет у процессора?

Какие функции в компьютере выполняет процессор?

Одна из основных функций процессора – это выполнение арифметических операций. Давайте создадим метод **sum**, который будет складывать два числа.

```
class Processor:
```

```
    name = "
```

```
    frequency = 0
```

```
    numcore = 0
```

```
    def sum (a, b):
```

```
        return a + b
```

Теперь попробуем создать экземпляр нашего класса и вызовем у него наш метод **sum** и выведем на экран результат этого метода:

```
proclntel = Processor
```

```
print(proclntel.sum(9, 8))
```

Результат:

17

Как видим все работает, наш процессор умеет складывать два числа.

! Проблемная задача:

Придумать и создать метод в своем классе (можно придумать новый класс либо взять с прошлого урока). Создать экземпляр класса и вызвать этот метод. Вывести результат на экран.

Также как и функции, методы могут ничего не возвращать, а просто выполнять какое-то действие. Например, выводить что-нибудь на экран или изменять значения полей (свойств) объекта.

Давайте создадим такой метод в нашем классе **Процессор**, пускай он будет выводить на экран название процессора, то есть поле **name**:

```
class Processor:
```



```
numcore = 0

def sum (a, b):

    return a + b

def showname():

    print(Processor.name)
```

Как мы видим, в методе **showname** мы не просто написали команду **print (name)**, а указали через точку, что это поле класса **Processor**, иначе у нас вышла бы ошибка, что имя **name** – неизвестно.

Вызовем этот метод, перед этим задав значение полю **name**:

```
procIntel.name = 'Intel Core i3'

procIntel.showname()
```

Результат:

Intel Core i3

! Проблемная задача:

Придумать два класса. В каждом создать по два метода: один метод, который ничего не возвращает, второй – возвращает числовое либо строковое значение. Протестировать их работу, создав объекты классов и вызвав их методы.

Урок 3

Конструктор класса

На прошлом уроке мы познакомились с понятием «методы класса» и научились их создавать.

На этом уроке мы познакомимся тоже с методом, но с особенным методом, который называется – **конструктор**.

Чтобы лучше понять, что это за такой метод-конструктор и для чего он нужен, проведем небольшое исследование. Давайте создадим класс, например, класс домашний питомец, назовем его Pet (с англ. – домашний питомец) и создадим в нем несколько полей (кличка, тип животного, порода, возраст, цвет, размер):

```
class Pet:

    name = ""

    animal = ""

    breed = ""

    age = 0

    color = ""

    size = ""
```

Размер у нас будет строковым типом, и принимать одно из следующих значений: very small, small, middle, big, very big.

А теперь давайте создадим три экземпляра нашего класса **Pet**, три домашних питомца.

! Проблемная задача:

Создать три экземпляра класса Pet и задать значения всем полям у всех трех домашних питомцев.

Пример результата:

```
catTom = Pet
```



```
catTom.breed = 'British'
```

```
catTom.age = 1
```

```
catTom.color = 'grey'
```

```
catTom.size = 'middle'
```

```
dogRex = Pet
```

```
dogRex.name = 'Rex'
```

```
dogRex.animal = 'dog'
```

```
dogRex.breed = 'Labrodor'
```

```
dogRex.age = 3
```

```
dogRex.color = 'black'
```

```
dogRex.size = 'big'
```

```
catBarsik = Pet
```

```
catBarsik.name = 'Barsik'
```

```
catBarsik.animal = 'cat'
```

```
catBarsik.breed = 'Scotish'
```

```
catBarsik.age = 0.5
```

```
catBarsik.color = 'white'
```

```
catBarsik.size = 'small'
```

Как видно, получилось не очень оптимально, чтобы создать три экземпляра класса, пришлось написать больше 20 строчек кода.

! Вопрос: как вы думаете, есть ли возможность создавать экземпляры классов, сразу задавая значения полям (свойствам), в одну строчку кода?

Ответ: да. И сделать это можно так:

```
catTom = Pet('Murzic', 'cat', 'British', 1, 'grey', 'middle')
```

Один экземпляр – одна строка кода.

Но, чтобы так можно было создавать объекты (экземпляры), необходимо кое-что изменить в самом классе. А именно, создать конструктор.

Конструктор – это метод, только специальный, и у него особое название: **`__init__`**.

Именно с двумя подчеркиваниями до и после самого слова. Этот метод автоматически выполняется при создании экземпляра класса, то есть объекта.

Рассмотрим это на примере и разберем его:

```
class Pet:
```

```
    def __init__(self, name, animal, breed, age, color, size):
```

```
        self.name = name
```

```
        self.animal = animal
```

```
        self.breed = breed
```



```
self.size = size
```

Сопоставим этот метод-конструктор и код создания объекта:

```
def __init__(self, name, animal, breed, age, color, size):  
  
catTom = Pet('Murzic', 'cat', 'British', 1, 'grey', 'middle')
```

Одинаковым цветом отмечены поля (свойства) и соответствующие значения объекта. Они расположены в одинаковом порядке.

Ну и конечно, у многих возник вопрос – что означает слово **self**?

Слово **self** указывает на сам объект (экземпляр) класса, который создается. В нашем случае он означает объект **catTom** класса **Pet**.

self.name по сути означает **Pet.name**.

Это слово служебное и используется в конструкторе и других методах во всех классах и заменяет сам экземпляр класса.

! Проблемная задача:

Создать класс с конструктором, создать в классе еще один метод с использованием служебного слова **name**. Далее в программе создать 2-3 экземпляра класса с помощью конструктора. Вызвать методы созданных объектов и вывести на экран результаты их работы.

Урок 4

Наследование

Мы с вами уже узнали, что такое классы, как их создавать, узнали, что такое методы класса и свойства или поля класса. Но это еще не все «фишки» ООП.

Сегодня мы познакомимся с очень важной или, можно так сказать, «крутой» фишкой в ООП – это наследование.

Наследование – это такой механизм, только это не обычный механизм, на подобии механизма в пистолете или автомобиле, а механизм, который сделан на языке программирования Python.

Похожий механизм существует в природе и называется также – **наследование**. Он заключается в том, что детеныши наследуют свойства от своих родителей. Детеныши львов – тоже львы, птенцы орла – тоже орлы, не может быть такого, чтобы у бегемота появились детеныши крокодила. Но кроме того, детеныши наследуют от родителей не только внешний облик, но и различные свойства. У самого быстрого леопарда всего скорее будет очень быстрый детеныш, а хитрой лисы будут хитрые лисята.

Так вот, в Python такой механизм тоже существует, только для классов, и работает он по строгим четким правилам.

Как всегда, все разберем на примере.

Пусть у нас есть класс Спорт, который представляет все виды спорта. То есть, если мы создадим объект этого класса, то этот объект будет являться каким-то конкретным видом спорта, например, футболом или теннисом.

class Sport:

```
def __init__(self, name):
```

```
self.name = name
```

```
def show(self):
```

```
print(self.name)
```

```
Fb = Sport('Football')
```

```
Fb.show()
```




Мы создали экземпляр класса – объект **Fb** и вызвали метод **show**. Если запустить программу, то на экран выведется:

Football

Теперь представим, что нам нужно добавить еще поля в классе, ведь у видов спорта много различных свойств и в разных видах спорта разные свойства: в футболе – количество человек в команде, размер поля, материал мяча, да и сам факт, что игра идет мячом; в боксе – количество раундов, весовая категория, вес перчаток и т.д. В каждом виде спорта свои индивидуальные свойства. Как их объединить в одном классе для всех видов спорта? Можно, конечно, все свойства у всех видов спорта указать в одном классе, но это будет громоздко и неудобно, поэтому придется создавать отдельные классы для видов спорта, что мы и сделаем, создадим класс **Football** и класс **Boxing**:

```
class Football:
```

```
    def __init__ (self, name, dur_time):

        self.name = name

        self.durationTime = dur_time    #длительность тайма

    def show(self):

        print(self.name)
```

```
class Boxing:
```

```
    def __init__ (self, name, wC):

        self.name = name

        self.weightCategory = wC    #весовая категория

    def show(self):

        print(self.name)
```

Как мы можем заметить, у этих двух классов есть одинаковые свойство **name** и метод **show**. Это не очень оптимально. Вот тут нам и поможет принцип наследования. Так-как у нас уже есть класс **Sport**, в котором уже есть и свойство **name** и метод **show**, то мы можем сделать наши классы **Football** и **Boxing** наследниками класса **Sport** и они автоматически унаследуют все его свойства и методы и эти унаследованные свойства и методы уже не нужно будет прописывать.

Чтобы сделать класс наследником другого класса, необходимо при создании класса, после его имени в скобках указать класс-родитель. Сделаем это с нашими классами:

```
class Sport:
```

```
    def __init__ (self, name):

        self.name = name

    def show(self):

        print(self.name)
```

```
class Football(Sport):
```

```
    pass
```

```
class Boxing(Sport):
```

```
    pass
```



```
Bx = Boxing('Boxing')
```

```
Bx.show()
```

```
Fb.show()
```

Слово **pass** является служебным и обозначает пустой код, то есть если мы ничего в классе не прописываем, класс, как-бы пустой получается, то мы обязаны прописать в нем служебное слово **pass**.

Запустим программу и увидим результат:

```
Boxing
```

```
Football
```

Наследование работает. Мы ничего не прописывали в классах футбола и бокса, но указали, что они являются наследниками класса **Sport**, в результате они унаследовали и свойство **name** и метод **show**. Кроме того, они еще и унаследовали метод – конструктор **__init__**.

Сделаем эти классы не пустыми, пропишем в конструкторе класса Football свойство `durationTime` (длительность тайма), а в классе Boxing пропишем свойство `weightCategory` (весовая категория):

```
class Football(Sport):
```

```
    def __init__(self, durT):
```

```
        self.durationTime = durT
```

```
class Boxing(Sport):
```

```
    def __init__(self, wC):
```

```
        self.weightCategory = wC
```

```
Fb = Football('Football', 45)
```

```
Bx = Boxing('Boxing', 'superheavy')
```

```
Bx.show()
```

```
Fb.show()
```

! Проблемная задача:

Объяснить в чем ошибка в этой программе

Ошибка заключается в том, что так-как мы прописали в классах Football и Boxing конструкторы, то теперь у них свои конструкторы, а не конструктор класса-родителя Sport. И так-как при создании объектов Fb и Bx мы в скобках указываем два параметра, а в конструкторах у них по одному, выходит ошибка. Поэтому необходимо немного исправить их конструкторы, добавить аргумент **name**:

```
class Football(Sport):
```

```
    def __init__(self, name, durT):
```

```
        self.name = name
```

```
        self.durationTime = durT
```

А можно сделать еще круче, внутри конструктора класса Football вызвать конструктор родительского класса Sport:

```
class Football(Sport):
```

```
self.durationTime = durT
```

! Проблемная задача:

Написать класс-родитель (в нем 2-3 свойства и 1-2 метода) и три класса наследника. Создать экземпляры классов (объекты), вызвать в программе методы этих объектов.

Урок 5

Инкапсуляция. Переопределение.

Инкапсуляция – это, когда поле (свойство) в классе защищено, так сказать, спрятано от использования за пределами класса.

В языке Python, очень легко скрыть (защитить) поле в классе, достаточно имя поля задать с двойным подчеркиванием вначале имени.

Рассмотрим на примере. Создадим класс, описывающий бытовую технику:

```
class HomeTech:

    def __init__(self, name):

        self.__name = name
```

Как видно поле **__name** мы сделали с двойным подчеркиванием.

Создадим объект этого класса:

```
kettle = HomeTech('чайник')
```

и попробуем вывести на экран его поле **name**:

```
kettle = HomeTech('чайник')
```

```
print(kettle._name)
```

Результат:

```
AttributeError: 'HomeTech' object has no attribute '_name'
```

Мы защитили свойство **name**, если мы уберем двойное подчеркивание, то ошибки уже не будет.

Раз поле защищено, то получать его значение и изменять его мы можем только с помощью методов класса.

! Проблемная задача:

Создать два метода в классе HomeTech, метод getname, который возвращает значение поля name и метод setname, который изменяет значени поля name на новое.

Переопределение – это когда в классе наследнике вы переписываете метод, который уже существует в классе-родителе.

Опять же, рассмотрим на примере.

Создадим в нашем классе **HomeTech** метод **work**, который будет выводить на экран надпись “**я работаю**”. А также, создадим класс, который будет наследоваться от класса **HomeTech**. Назовем его **Blender**:

```
class HomeTech:

    def __init__(self, name):

        self.__name = name
```



Python - ООП, классы

```
class Blender(HomeTech):  
  
    pass
```

```
blender = Blender('Блендер')
```

```
blender.work()
```

Запустим программу, увидим, что вывелась надпись: “Я работаю”.

А теперь создадим в классе **Blender** метод **work**, как в классе родителя, только выводить он будет надпись: “Я смешиваю”.

```
class Blender(HomeTech):  
  
    def work(self):  
  
        print('Я смешиваю')
```

Запустим программу и увидим, что теперь вывелась надпись:

“Я смешиваю”

То есть, мы переопределили метод **work**.

! Проблемная задача:

Создать класс родитель и в нем метод. Создать два класса наследника, в одном из них переопределить метод родительского класса. Протестировать работу методов.

Урок 6

Библиотека Tkinter

Введение

На этом уроке мы начнем знакомство с чудесной библиотекой языка Python, которая называется Tkinter. С помощью этой библиотеки можно разрабатывать оконные программы, то есть программы с привычным нам интерфейсом, где есть окно программы, в окне есть различные надписи, поля для ввода текста, кнопки, списки и рисунки.

Данная библиотека полностью основана на принципе ООП, она содержит классы, которые описывают объекты, из которых как раз и состоит оконный интерфейс (окна, надписи, кнопки и т.д.)

Окно программы

Чтобы использовать какую-либо библиотеку, необходимо прописать в программе команду, которая импортирует (подключит) это библиотеку. Для этого используется служебное слово **import** и после него указывается название библиотеки:

```
import tkinter
```

В этой библиотеке есть класс **Tk**, который описывает такие объекты как окна – окна программ.

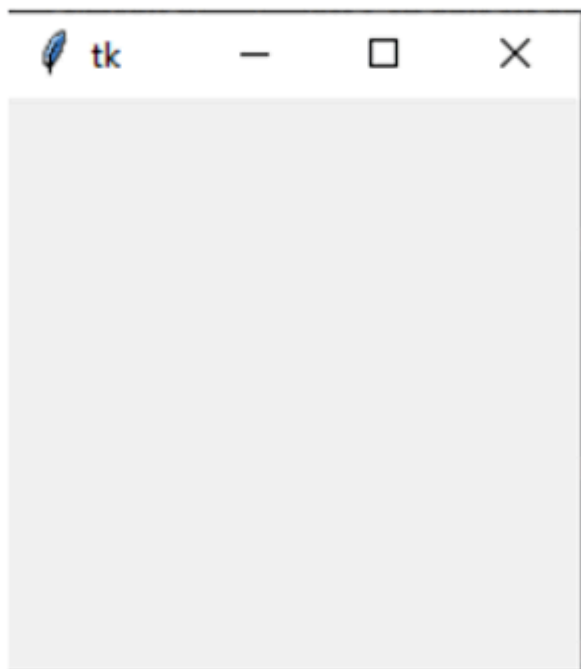
При создании объекта (экземпляра) этого класса, создается окно программы.

Давайте попробуем, но уточним один момент: чтобы обратиться к классу из библиотеки, необходимо сначала написать название библиотеки, а потом через точку название класса:

```
import tkinter
```

```
window = tkinter.Tk()
```

Запустим программу...

Рисунок 4

Примечание: в некоторых системах разработки окно может появиться и сразу исчезнуть, так как по правильному, программа создаст окно и сразу закончит работу.

Чтобы окно не закрывалось, необходимо вызвать у окна, то есть у объекта **window**, который мы создали, метод **mainloop()**. Этот метод переведет нашу программу, а точнее, окно программы в режим ожидания от пользователя каких-либо действий: нажатия на кнопку, щелчка или движения мышью, либо нажатия клавиши на клавиатуре.

Поэтому, чтобы программа правильно обрабатывала все действия пользователя, необходимо выполнить этот метод:

```
import tkinter

window = tkinter.Tk()

window.mainloop()
```

Также, команду **import** можно использовать немного по-другому. Если напишем такую команду:

```
from tkinter import *
```

То в нашу программу импортируются сразу все, что есть в библиотеке **tkinter**. И теперь, если нам понадобится создать объект какого-нибудь класса, уже не нужно будет писать название библиотеки и через точку название класса, можно просто указывать название класса:

Вместо:

```
window = tkinter.Tk()
```

пишем:

```
window = Tk()
```

В итоге наша программа выглядит так:

```
from tkinter import *

window = Tk()

window.mainloop()
```

Итак, мы научились создавать окно.

Но хотелось бы создавать окно необходимого размера. Чтобы задать размер окна, необходимо вызвать метод:

geometry(stringSettings)



Шаблон строки:

`‘widthxheight+left+top’`

Пример:

```
window.geometry('500x500+100+100')
```

Следующая команда, связанная с окном – это:

```
title(str)
```

где **str** – это строка

Этот метод задает заголовок окна. Пример:

```
window.title("My first window")
```

! Проблемная задача:

Написать программу, которая создает четыре окна, чтобы они образовали квадрат и при этом касались друг друг

Урок 7

Виджеты

На прошлом уроке мы научились создавать окно программы, задавать ему размеры и положение на экране монитора.

Сегодня мы узнаем, как отображать в окне текстовые элементы, задавать им размеры и расположение в окне.

Элементы, которые располагаются в окне приложения называются **виджеты**. Поэтому, текстовый элемент – это виджет.

Текстовый элемент является объектом класса **Label** (перевод с английского – метка).

Можно еще сказать, что текстовый элемент – это своего рода метка, только текстовая. В большинстве библиотек и языках программирования текстовые элементы интерфейса называются **Label**.

Итак, чтобы создать текстовый элемент(метку), необходимо создать объект класса **Label**:

```
Text1 = Label(window, text='Hello!')
```

Здесь первый параметр – это родительский элемент, в котором будет находится текст. В нашем случае родительским элементов является наше окно – **window**. Второй параметр мы указали сам текст, который будет выводится в этом текстовом элементе.

Но, если мы сейчас запустим программу, на окне ничего не появится, это потому, что наш виджет – Text1, необходимо расположить на окне.

Для этого у каждого виджета в библиотеке tkinter существуют специальные методы.

Всего таких методов три:

pack

grid

place

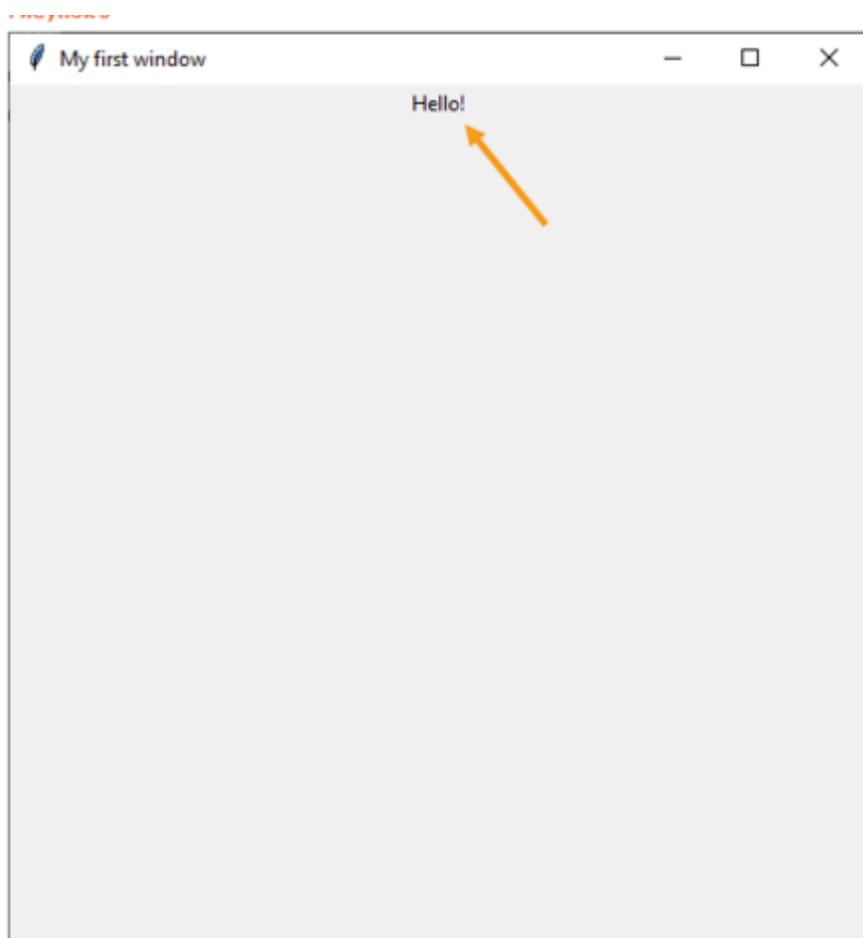
Сегодня мы познакомимся с методом pack, а на следующих уроках изучим остальные.

Метод **pack**

Как вы поняли, чтобы разместить элемент (виджет) в окне, необходимо вызвать у этого элемента метод pack.

Сделаем это:

```
Text1.pack()
```



Стоит обратить внимание, что наша надпись находится по центру окна в горизонтальной оси.

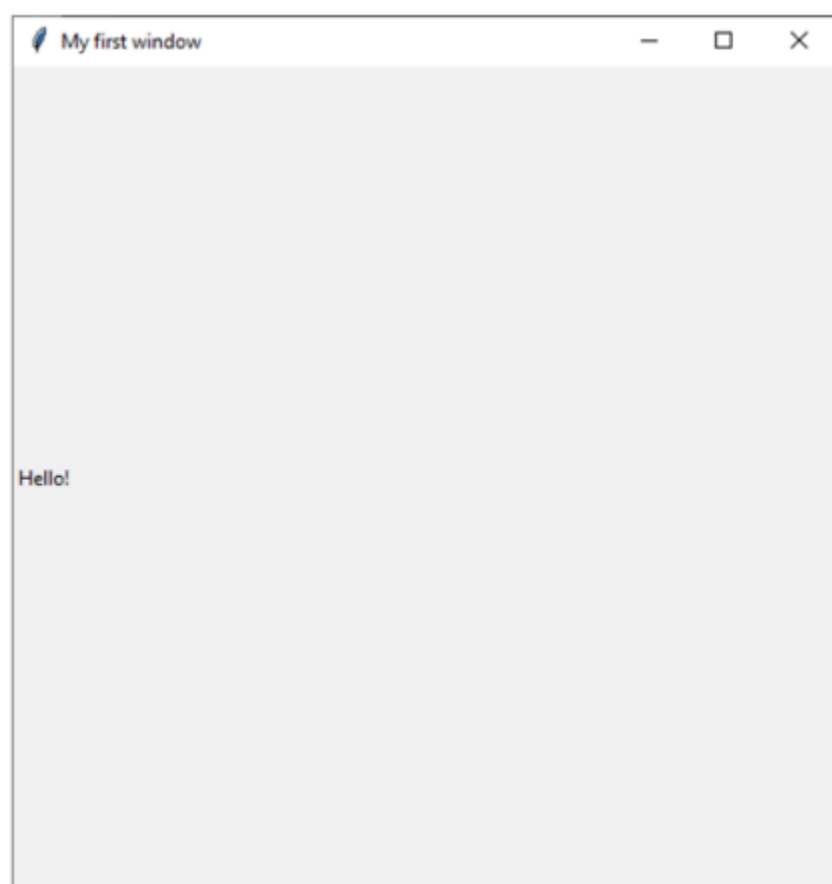
Если мы хотим задать расположение как то по-другому, то в методе **pack** в скобках можно указать специальные параметры.

Например, если мы хотим прижать надпись к левой границе окна, то в этом нам поможет параметр: **side** (перевод с англ. – сторона).

Например (рисунок 6):

```
Text1.pack(side = LEFT)
```

Рисунок 6



! Проблемная задача:

Расположить виджет Label снизу окна, а потом справа окна.

Решение:

```
Text1.pack(side = BOTTOM) # снизу
```

```
Text1.pack(side = RIGHT) # справа
```

```
Text1.pack(side = TOP) # сверху
```

TOP.

А теперь, попробуем расположить в окне несколько текстовых виджетов.

! Проблемная задача:

Создать несколько виджетов Label , 3-4 штуки и все их расположить в окне с помощью метода pack. Сделать все это в 4-х вариантах:

Используя метод без параметров pack() либо задав параметр side = TOP

Задав параметр side = LEFT

Задав параметр side = BOTTOM

Задав параметр side = RIGHT

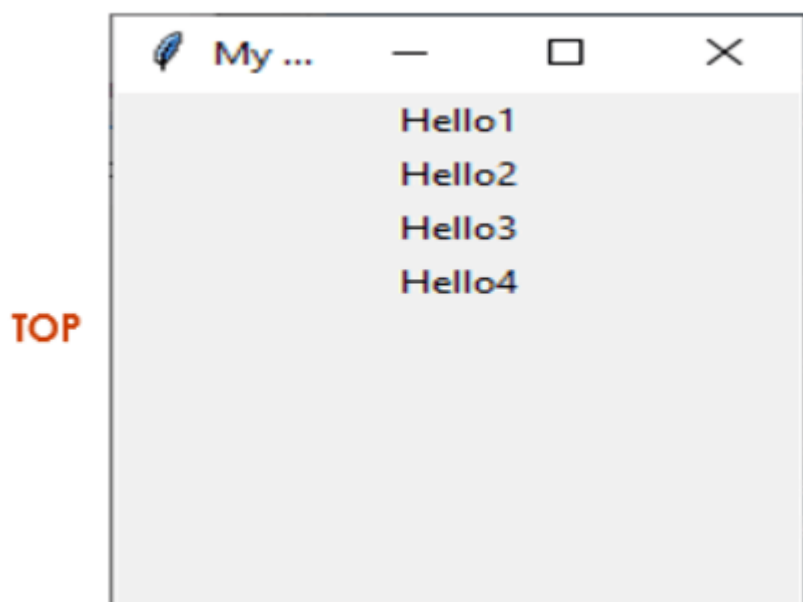
Решение:

```
Text1.pack(side = TOP)
```

```
Text2.pack(side = TOP)
```

```
Text3.pack(side = TOP)
```

```
Text4.pack(side = TOP)
```

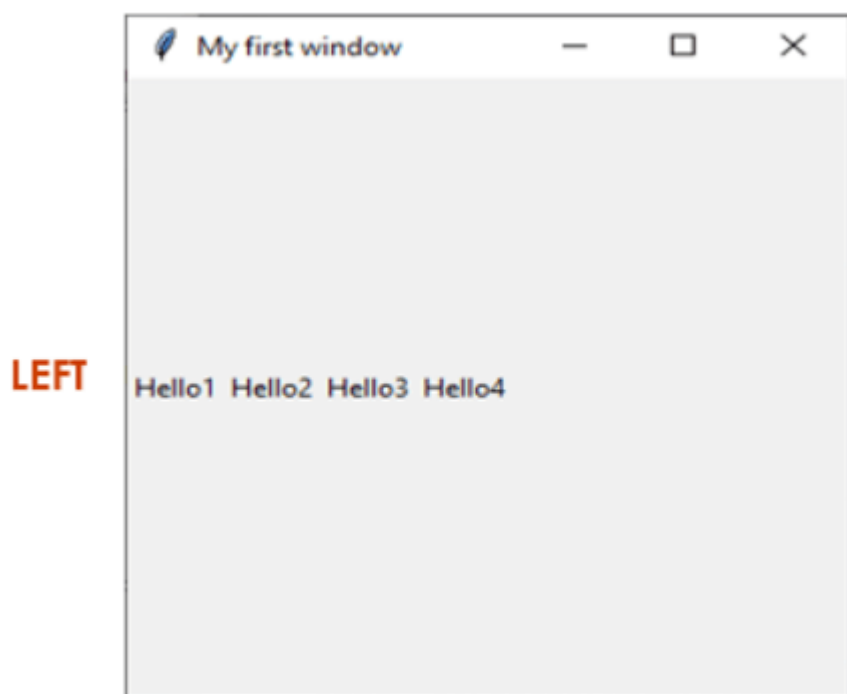


```
Text1.pack(side = LEFT)
```

```
Text2.pack(side = LEFT)
```

```
Text3.pack(side = LEFT)
```

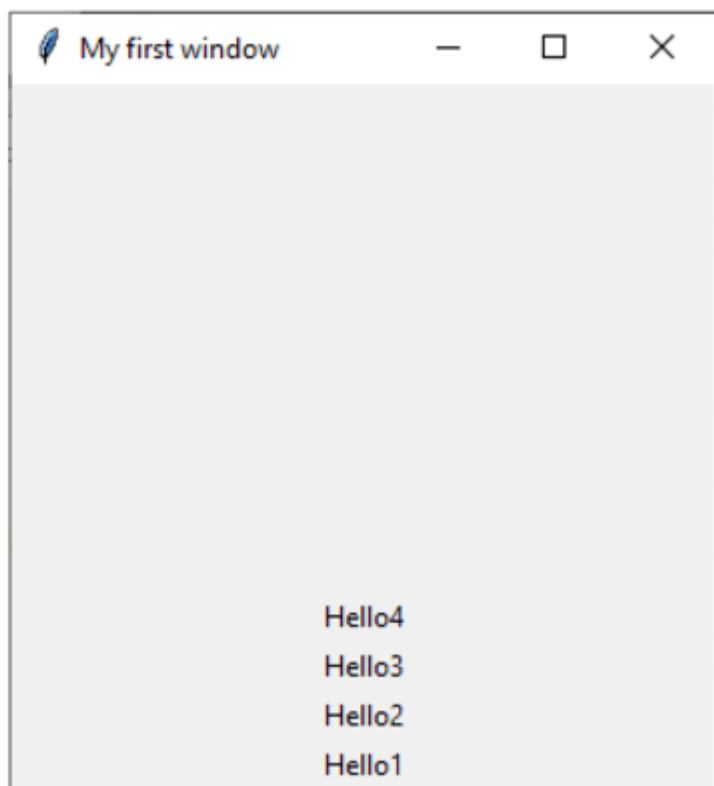
```
Text4.pack(side = LEFT)
```



```
Text1.pack(side = BOTTOM)
```

```
Text2.pack(side = BOTTOM)
```


BOTTOM



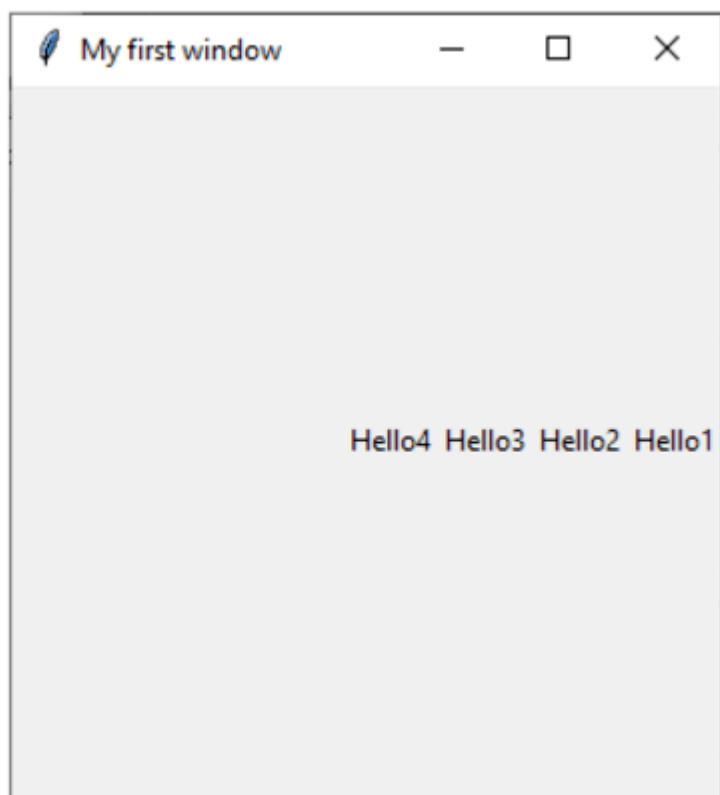
```
Text1.pack(side = RIGHT)
```

```
Text2.pack(side = RIGHT)
```

```
Text3.pack(side = RIGHT)
```

```
Text4.pack(side = RIGHT)
```

RIGHT



УРОК 8

КНОПКА

Пожалуй, самый часто используемый виджет – это кнопка.

Виджет кнопка описывает класс Button.

Создадим кнопку:

```
B = Button(text="Press me!", width=20, height=5)
```

```
B.pack()
```

Как видно у кнопки есть такие параметры:

text - задает текст на кнопке

width – задает ширину кнопки

height – задает высоту кнопки



действие, которое будет выполняться при нажатии.

Действие кнопки в **tkinter** привязывается к функции, которая создается с помощью знакомого нам служебного слова **def**.

Поэтому, чтобы задать кнопке действие при нажатии, нужно сначала в программе создать функцию, которая будет выполняться при нажатии, а потом при создании кнопки, в скобках задать параметр **command** и присвоить ему название созданной функции.

Рассмотрим на примере:

```
def createLabel ():
```

```
    L1 = Label(window, text = 'А вот и я!')
```

```
    L1.pack()
```

```
...
```

```
...
```

```
B = Button(text="Press me!", width=20, height=5, command = createLabel)
```

```
B.pack()
```

Мы привязали кнопку к функции **createLabel**. Теперь при нажатии на кнопку будет выполняться эта функция. А так как в коде функции создается виджет Label, то соответственно при нажатии кнопки в нашем окне появиться текстовое поле с надписью: «**А вот и я!**».

! Проблемная задача.

Создать 4 кнопки, каждая из которых создает виджет Label, но с разным расположением, то есть одна кнопка создает виджет с параметром side = TOP, вторая side = LEFT, третья side = BOTTOM и четвертая side = RIGHT.

Если мы хотим нажатием кнопки менять параметры виджетов, например, чтобы при нажатии кнопки менялся текст виджета Label, то для этого можно использовать метод config, который есть у всех виджетов. Этот метод меняет параметры на новые значения, которые задаются в скобках у этого метода.

Пример:

```
def changeLabel():
```

```
    L1.config(text = 'Это мой новый текст!')
```

```
.....
```

```
.....
```

```
L1 = Label(window, text = 'Привет!')
```

```
L1.pack()
```

```
B = Button(text="Press me!", width=20, height=5, command = changeLabel)
```

```
B.pack()
```

! Проблемная задача:

Создать 3 виджета Label. Создать три кнопки, каждая из которых меняет текст одного из виджетов Label. Создать кнопку, которая меняет текст у всех трех виджетов Label на тот который был изначально.

! Проблемная задача:

Создать свой класс MyButton, который является наследником класса Button, добавить в него поле mysize.

Запрограммировать кнопку так, чтобы при нажатии она увеличивалась, а при повторном нажатии уменьшалась. То есть, если кнопка маленькая, то при нажатии она становится большой, а если большая, то при нажатии становиться маленькой.

Использовать для этого поле mysize.

Виджет для ввода информации от пользователя

Все мы знаем, что в различных программах и приложениях есть такие поля, куда пользователь вводит какую-либо информацию.

В **tkinter** тоже есть такое поле, и отвечает за эти поля класс **Entry**.

Создадим такое поле:

```
E = Entry(window, width=20)
```

```
E.pack()
```

Теперь нам нужно, получить текст, который в нем содержится, то есть то, что напечатает в этом поле пользователь. Для этого в классе **Entry** есть метод **get()**. Этот метод возвращает строку, которая содержится в виджете.

Пример:

```
def click():
```

```
    L1.configure(text = 'Hello, ' + E.get())
```

```
...
```

```
...
```

```
L1 = Label(window, text = 'Привет!')
```

```
L1.pack()
```

```
B = Button(text="Press me!", width=20, height=5, command = click)
```

```
B.pack()
```

```
E = Entry(window, width=20)
```

```
E.pack()
```

В данном примере, при нажатии на кнопку, текст виджета Label изменяется на “Hello, <текст введенный пользователем в виджет Entry>”

! Проблемная задача:

Написать простой калькулятор с четырьмя операциями: +, -, *, /.

Создать два виджета Entry. В них пользователь вводит два числа.

Создать 4 кнопки, соответствующие математическим операциям.

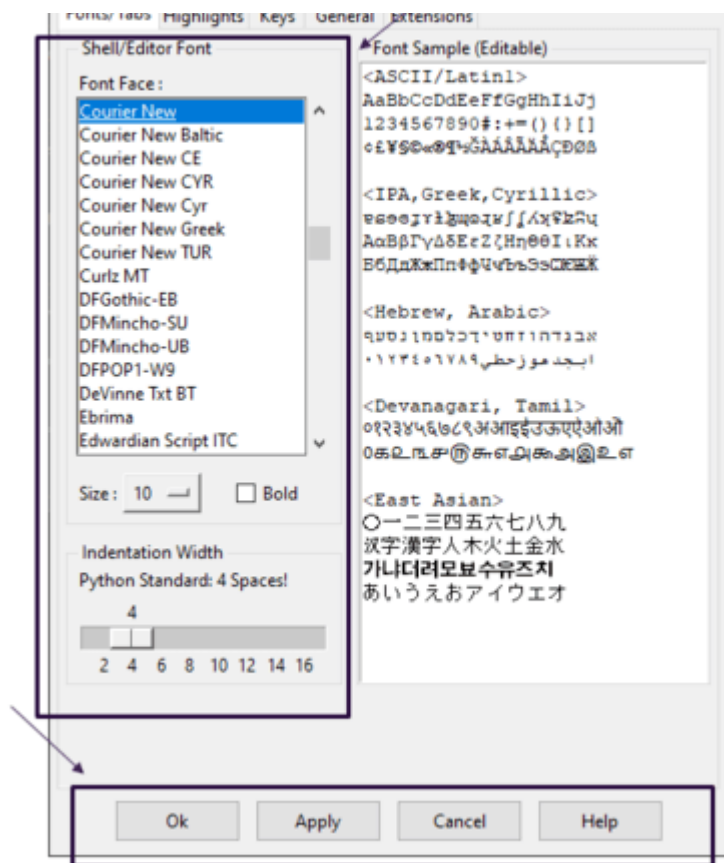
Создать виджет Label, для вывода результата.

Урок 10

Виджет Frame

Бывают случаи, когда нам необходимо визуально разделить окно на несколько частей, и в каждой части, чтобы располагались свои виджеты.

Пример (рисунок 7):



В **tkinter** для этого существует такой виджет, который называется **Frame**, а точнее так называется класс, который описывает эти виджеты.

Frame – это такой виджет, который является контейнером для других виджетов. Визуально он выглядит как прямоугольная область, в которой размещаются другие виджеты.

Давайте создадим два виджета Frame и в каждом из них разместим по три виджета. Будем называть виджет Frame – **рамкой**. Итак, в первой рамке разместим три виджета Label, а во второй рамке разместим три виджета Entry:

```
F1 = Frame(window, borderwidth = 1, relief=GROOVE)
```

```
F1.pack(side=LEFT)
```

```
F2 = Frame(window, borderwidth = 5, relief=GROOVE)
```

```
F2.pack(side=LEFT)
```

```
L1 = Label(F1, text = 'Имя')
```

```
L2 = Label(F1, text = 'Фамилия!')
```

```
L3 = Label(F1, text = 'Отчество!')
```

```
L1.pack()
```

```
L2.pack()
```

```
L3.pack()
```

```
E1 = Entry(F2, width=20)
```

```
E2 = Entry(F2, width=20)
```

```
E3 = Entry(F2, width=20)
```

```
E1.pack()
```

```
E2.pack()
```

```
E3.pack()
```

При создании объекта Frame, мы задали параметр borderwidth, который задает ширину контура рамки в пикселах и второй параметр – relief, который задает тип этого контура.

! Проблемная задача:



RAISED

SUNKEN

RIDGE

Разместить в каждом виджете Frame по три виджета: Label, Entry, Button

Урок 11

Создаем меню

Сегодня мы познакомимся со следующим виджетом – это всем хорошо знакомое **МЕНЮ**.

За меню в **tkinter** отвечает класс **Menu**.

Создадим объект этого класса:

```
M = Menu(window)
```

```
M.add_command(label='Файл')
```

```
window.configure(menu=M)
```

Метод **add_command** добавляет пункт в меню, в скобках можно задать параметры этого пункта. В нашем примере мы задали параметр **label**, который задает текст в пункте меню.

Также мы вызвали уже знакомый нам метод **configure** у объекта **window**. Так-как мы изменили объект **M**, который представляет собой наше меню, то мы должны как бы обновить его в нашем окне, поэтому мы заново присваиваем параметру **menu** у окна **window** наше обновленное меню.

Чтобы создать в пункте меню выпадающее подменю, необходимо использовать метод

add_cascade

Этому методу необходимо в качестве параметра **menu** передать другой объект класса **Menu**.

Рассмотрим на примере:

```
M = Menu(window)
```

```
Mm1 = Menu(M)
```

```
Mm1.add_command(label='Открыть')
```

```
Mm1.add_command(label='Сохранить')
```

```
Mm2 = Menu(M)
```

```
Mm2.add_command(label='Копировать')
```

```
Mm2.add_command(label='Вставить')
```

```
M.add_cascade(label='Файл', menu=Mm1)
```

```
M.add_cascade(label='Редактировать', menu=Mm2)
```

```
window.configure(menu=M)
```

В примере мы сначала создали два подменю **Mm1** и **Mm2**, потом в основное меню **M** добавили в качестве элементов эти два подменю.

например 3, 4 и 5 пунктов.

Чтобы при выборе определенного пункта меню выполнялась написанная нами функция, необходимо также как и виджета **Button** задать параметр **command**.

! Проблемная задача:

К созданным пунктам меню придумать и добавить действия, например, изменение текста виджетов или появление новых виджетов.

Урок 12

Метод grid ()

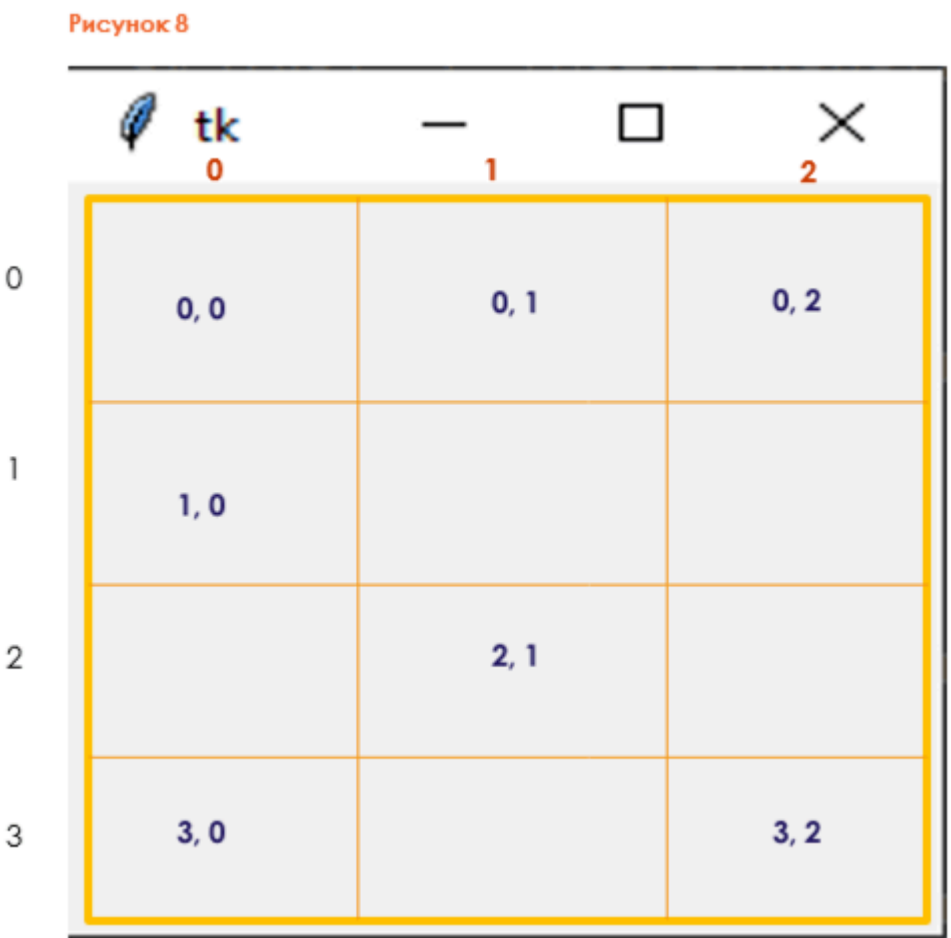
До сегодняшнего урока, чтобы расположить виджеты в окне или в контейнере **Frame**, мы использовали метод **pack()**.

Сегодня мы рассмотрим еще один метод для расположения виджетов – это метод **grid()**.

Суть этого метода такая:

Представьте, что окно нашей программы (приложения) разбито на клетки как таблица (рисунок 8):

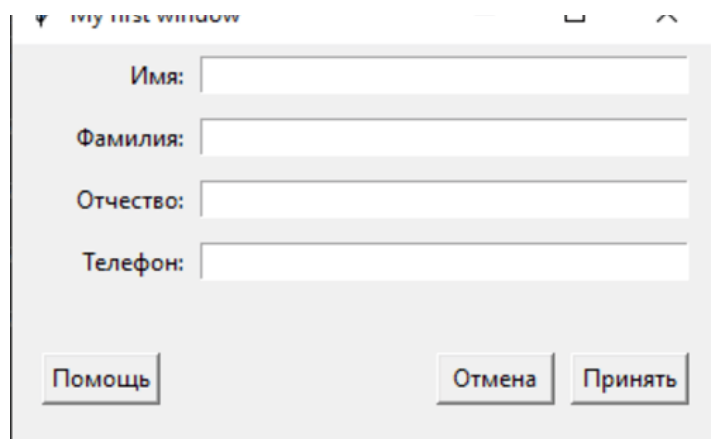
Рисунок 8



И каждая клетка имеет свой номер, состоящий из двух числе: первое число, означает номер строки в таблицу, а второе – номер столбца. А также, нужно иметь ввиду нюанс, что нумерация строк и столбцов начинается с нуля.

Далее в методе **grid** в качестве параметра **row** задается номер строки, а в качестве параметра **column** задается номер столбца.

Давайте попробуем создать вот такое окно для ввода информации о пользователе (рисунок 9):



Разделим визуально окно на ячейки, как таблицу:

	0	1	2	3
0	Имя:			
1	Фамилия:			
2	Отчество:			
3	Телефон:			
4	Помощь		Отмена	Принять

Исходя из такой разметки создадим виджет Label с текстом “Имя:”

```
L1 = Label(window, text = 'Имя:')
```

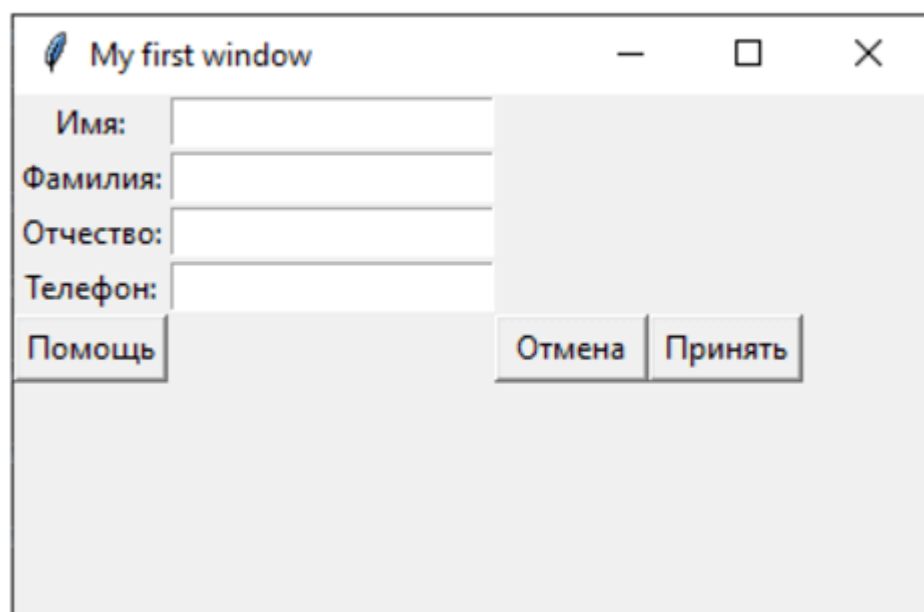
```
L1.grid(row=0, column=0)
```

! Проблемная задача:

Создать остальные виджеты и расположить их с помощью метода grid, как на рисунке 9.

Если вы все сделали правильно, то у вас получится вот так (рисунок 10):

Рисунок 10



Код программы:

```
L1 = Label(window, text = 'Имя:')
```

```
L2 = Label(window, text = 'Фамилия:')
```

```
L3 = Label(window, text = 'Отчество:')
```

```
L4 = Label(window, text = 'Телефон:')
```



```
        L1.grid(row=0, column=0)
```

```
L2.grid(row=1, column=0)
```

```
L3.grid(row=2, column=0)
```

```
L4.grid(row=3, column=0)
```

```
B1 = Button(window, text='Помощь', width=7)
```

```
B2 = Button(window, text='Отмена', width=7)
```

```
B3 = Button(window, text='Принять', width=7)
```

```
B1.grid(row=4, column=0)
```

```
B2.grid(row=4, column=2)
```

```
B3.grid(row=4, column=3)
```

```
E1 = Entry(window)
```

```
E2 = Entry(window)
```

```
E3 = Entry(window)
```

```
E4 = Entry(window)
```

```
E1.grid(row=0, column=1)
```

```
E2.grid(row=1, column=1)
```

```
E3.grid(row=2, column=1)
```

```
E4.grid(row=3, column=1)
```

Выглядит, мягко говоря, не очень.

Чтобы отрегулировать положения виджетов, как нам нужно, рассмотрим еще несколько параметров у метода grid.

Параметр **columnspan** , устанавливает количество столбцов, которые будет занимать виджет или, говоря по-другому, объединяет ячейки по горизонтали.

Пример:

```
E1.grid(row=0, column=1, columnspan=3)
```

Теперь поле ввода для имени будет занимать три ячейки по горизонтали. Увеличим ширину виджета, чтобы он занял все пространство:

```
E1 = Entry(window, width=40)
```

...

```
E1.grid(row=0, column=1, columnspan=3)
```

! Проблемная задача:

С помощью параметра columnspan вытянуть остальные виджеты для ввода информации.

Результат:

```
E1 = Entry(window, width=40)
```

```
E2 = Entry(window, width=40)
```

```
E3 = Entry(window, width=40)
```

```
E4 = Entry(window, width=40)
```



```
E2.grid(row=1, column=1, columnspan=3)
```

```
E3.grid(row=2, column=1, columnspan=3)
```

```
E4.grid(row=3, column=1, columnspan=3)
```

Стало лучше, но необходимо еще задать отступы между виджетами, чтобы они не располагались вплотную друг к другу.

В этом нам помогут параметры у метода grid:

`padx` – отступы по горизонтали.

`pady` – отступы по вертикали.

Пример:

```
L1.grid(row=0, column=0, pady=5)
```

```
L2.grid(row=1, column=0, pady=5)
```

```
L3.grid(row=2, column=0, pady=5)
```

```
L4.grid(row=3, column=0, pady=5)
```

...

...

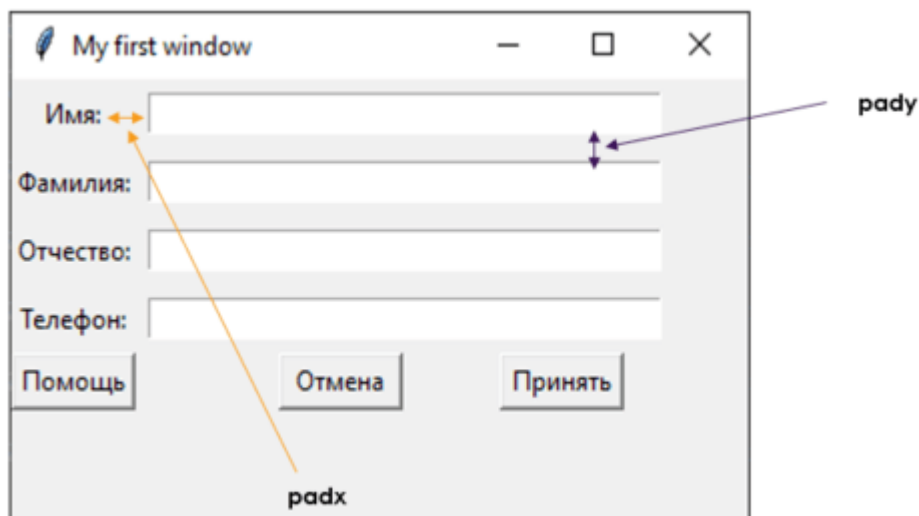
```
E1.grid(row=0, column=1, columnspan=3, padx=5)
```

```
E2.grid(row=1, column=1, columnspan=3, padx=5)
```

```
E3.grid(row=2, column=1, columnspan=3, padx=5)
```

```
E4.grid(row=3, column=1, columnspan=3, padx=5)
```

Получилось:



И еще один параметр, который позволяет прижимать виджеты либо к границам ячейки. Можно прижимать к левой, правой, верхней, нижней и еще можно комбинировать: например, верхней и левой или нижней и левой и т.д.

За это отвечает параметр:

sticky

Ему можно задавать следующие значения:

W – левая граница, E – правая граница, N – верхняя, S – нижняя, SE – нижняя и правая и т.д.

Как вы могли догадаться, буквы обозначают стороны света, как на компасе.

! Проблемная задача:

Доделать нашу форму, чтобы виджеты были расположены, как на рисунке 9.



Виджет Canvas

В библиотеке **tkinter** есть виджет, с помощью которого можно создавать графику, рисовать геометрические фигуры и отображать собственные изображения из файлов.

Класс, который задает такие виджеты, называется **Canvas**.

Создадим такой виджет и нарисуем линию красного цвета:

```
C = Canvas(window)
```

```
C.pack()
```

```
C.create_line(10, 10, 200, 100, fill='red')
```

Мы создали экземпляр класса **Canvas**, назвали его **C**, и вызвали у него метод **create_line**, который рисует линию.

В методе **create_line**, первые четыре параметра это координаты начала линии и конца.

Начало координат находится в левом верхнем углу родительского контейнера (в нашем случае родительский контейнер – это окно программы)

Параметр **fill**, задает цвет линии.

Также мы можем нарисовать прямоугольник:

```
C.create_rectangle(10, 105, 190, 190,  
                  fill='#FF4500', outline='#48D1CC', width=5)
```

Здесь мы добавили параметры:

outline – задает цвет границы

width – задает толщину границы в пикселах

Цвет можно задавать не только английским словом, но и в шестнадцатеричном формате.

Круг или овал можно нарисовать с помощью метода:

create_oval

Пример:

```
C.create_oval(200,50, 280, 130, fill='#D2691E', activefill='#191970')
```

Здесь мы использовали еще один параметр – **activefill**.

Он задает цвет, на который будет меняться овал при наведении на него мышью.

Все параметры, которые мы с вами изучили на этом уроке применимы ко всем геометрическим фигурам.

Каждый из методов создания геометрической фигуры возвращает указатель на эту фигуру. Поэтому, если мы хотим после создания, например, прямоугольника производить с ним какие-либо операции, необходимо присвоить результат метода `create_rectangle` переменной:

```
rect = C.create_rectangle(10, 105, 190, 190,  
                          fill='#FF4500', outline='#48D1CC', width=5)
```

Теперь, если мы захотим удалить прямоугольник, то можно вызвать метод **delete** у виджета **Canvas** и в качестве параметра передать указатель на прямоугольник – **rect**.

Пример:

```
C.delete(rect)
```



Три кнопки, каждая рисует свою фигуру, например, круг, прямоугольник и треугольник.

При повторном нажатии на кнопку, соответствующая ей фигура исчезает.

Урок 14

Анимация.

Давайте попробуем создать простую анимацию, например, чтобы при нажатии на кнопку, круг начинал двигаться от левого края виджета **Canvas** до правого.

Создадим круг, и кнопку:

```
C = Canvas(window, width=300, height=150)
```

```
C.pack()
```

```
oval = C.create_oval(10,50, 60, 100, fill='#D2691E', activefill='#191970')
```

```
B = Button(window, text = 'GO')
```

```
B.pack()
```

Чтобы заставить двигаться наш овал `oval`, воспользуемся новым для нас методом виджета **Canvas**:

move(*figure*, *dx*, *dy*)

Этот метод сдвигает фигуру **figure** по горизонтальной оси на значение **dx**, а по вертикальной на значение **dy**.

Чтобы посмотреть, как работает этот метод, давайте создадим функцию **go**, в этой функции вызовем метод **move**, а выполнение функции назначим на кнопку:

```
def go():
```

```
    C.move(oval, 1, 0)
```

```
...
```

```
oval = C.create_oval(10,50, 60, 100, fill='#D2691E', activefill='#191970')
```

```
B = Button(window, text = 'GO', command=go)
```

```
B.pack()
```

Запустим программу и увидим, что при нажатии на кнопку круг по чуть-чуть сдвигается.

Теперь сделаем так, чтобы круг сам двигался до правой границы окна.

Для этого нам понадобится метод **after** у нашего окна **window**.

В него передается два параметра:

after (*time*, *function*)

time – время в миллисекундах, а **function** – имя функции, которая вызовется через время **time**.

Вызовем этот метод в функции **go**:

```
def go():
```

```
    C.move(oval, 1, 0)
```

```
    window.after(10, go)
```

Но в таком виде запускать функцию не стоит, можно догадаться, что она заиклится до бесконечности и наш круг будет бесконечно сдвигаться вправо. Нам нужно установить проверку, чтобы вызов функции внутри себя остановился, если координата **x** правого нижнего угла круга будет больше, чем ширина окна.

Для этого воспользуемся методом **coords** у виджета **Canvas**, в который в качестве параметра передается фигура, а возвращает этот метод список из 4-х координат левого верхнего угла и правого нижнего.

```
def go():

    C.move(oval, 1, 0)

    if C.coords(oval)[2] < 290:

        window.after(10, go)
```

Запустим программу – круг двигается!

! Проблемная задача:

Создать несколько фигур, задать анимацию, чтобы фигуры двигались хаотично.

Урок 15

События от пользователя

В **tkinter** можно привязывать различные события (нажатия клавиш клавиатуры и мыши, движения мышью) к различным виджетам, а также к фигурам, которые нарисованы на виджете Canvas.

Для того, чтобы привязать событие к виджету используется метод:

bind (event, function)

event – это событие, оно задается специальной строкой.

function – это функция, которая будет выполняться при возникновении события.

Например, сделаем так, чтобы при нажатии правой клавишей мыши по виджету Label, этот виджет менял цвет:

```
def changecolorL(event):

    L.config(bg='red')

...

L = Label(window, text='Chameleon')

L.pack()

L.bind('<Button-3>', changecolorL)
```

Как видно событие нажатия правой клавиши мыши задается специальной строкой:

<Button-3>

Вот список некоторых строк, обозначающих события:

<Button-1> нажата клавиша мыши 1, вместо 1 может быть другая цифра

<Double-Button-1> двойное нажатие клавиши мыши



<Key> нажата любая клавиша клавиатуры

a – нажата буква **"a"** на клавиатуре, может быть любая другая буква, используется без скобок **< >**.

Также необходимо указывать при создании функции параметр **event**, он передается по умолчанию и его можно использовать в самой функции для получения дополнительных сведений о событии, например о координатах указателя мыши.

Чтобы привязать событие к нарисованной фигуре, например к прямоугольнику, используется метод виджета Canvas:

tag_bind (figure, event, function)

Рассмотрим на примере. Нарисуем прямоугольник и сделаем так, чтобы, когда мы щелкаем левой клавишей мышки по прямоугольнику, он менял цвет.

```
def changecolor(event):  
    C.itemconfig(rect, fill='#00BFFF')  
  
...  
  
C.tag_bind(rect,'<Button-1>',changecolor)
```

! Проблемная задача:

Отобразить в окне прямоугольник и создать кнопку, чтобы при нажатии на левую кнопку мыши, прямоугольник увеличивался, а при нажатии на правую кнопку мыши, прямоугольник уменьшался.

Использовать для этого метод виджета Canvas:

scale (figure, dx, dy, scalex, scaley)

dx, dy – смещения по осям, но вместо них лучше использовать координаты левого верхнего угла прямоугольника.

scalex, scaley – числа, обозначающие во сколько раз растянется фигура по соответствующей оси.

Урок 16

Проекты учеников

На основе изученных тем ученикам к концу курса предлагается написать собственные проекты.

Вот некоторые примеры предлагаемых проектов:

- Физический калькулятор: программа, рассчитывающая различные физические величины по формулам.
- Игра морской-бой
- Текстовый редактор
- Программа, которая рисует графики известных функций из математики
- Исторический справочник

Предлагаемая схема проведения урока:

- Жеребьевка очередности учеников
- Рассказ ученика о своей программе, презентация ее работы, рассказ об используемых алгоритмах, виджетах
- Вопросы ученику

