«ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
МІНІСТЕРСТВА ОСВІТИ І НАУКІ УКРАЇНИ

С.Ю. Борю

Модули и пакеты в Python

учебно-методическое пособие для студентов
естественнонаучных специальностей

Затверджено вченою радою ЗНУ
Протокол № ___ від _____

Запорожье
2021

УДК:

ББК:

Борю С.Ю. Модули и пакеты в Python. Учебно-методическое пособие для студентов естественно научных специальностей – Запорожье; ЗНУ, 2021, - 286 с.

Рассматриваются популярные модули и пакеты системы программирования Python. Приведится большое количество примеров и фрагментов программ, найденных на сайтах, посвященных системе программирования Питон и адаптированных для версии 3.х. Пособие ориентированно на самостоятельную работу студентов естественнонаучных специальностей.

Рецензент – Кеберле Н.Г.

Ответственный за выпуск - Матвейшена Н.В.

Модули и пакеты в Python

Оглавление

Модули и пакеты в Python

Установка python-пакетов с помощью pip

pip - это система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python. Начиная с Python версии 3.4, pip поставляется вместе с интерпретатором python.

Начало работы

Попробуем с помощью pip установить какой-нибудь пакет, например, **numpy** (https://pythonworld.ru/numpy) в консоле (в режиме администратора):

```
Linux:
sudo pip3 install numpy

Windows:
pip3 install numpy
```

Может возникнуть ошибка: *"pip3" не является внутренней или внешней командой, исполняемой программой или пакетным файлом*.

Тогда необходимо указывать полный путь к программе, например (если Питон установлен в `C:\Python36\`):

```
C:\Python36\Tools\Scripts\pip3.exe install numpy
```

Либо добавлять папку `C:\Python36\Tools\Scripts\` в PATH вручную.

Либо сделав эту папку с pip3 текущей:

```
cd "C:\Python36\Tools\Scripts\"
pip3 install numpy
```

В последних версиях python вызов **pip** удобно выполнять из консоли:

```
python -m pip <command> [options]
```

Что ещё умеет делать pip3

Основные команды **pip3**:

pip help - помощь по доступным командам.

pip install package_name - установка пакета(ов).

pip uninstall package_name - удаление пакета(ов).

pip list - список установленных пакетов.

pip show package_name - показывает информацию об установленном пакете.

pip search - поиск пакетов по имени.

pip --proxy user:passwd@proxy.server:port - использование с прокси.

pip install -U - обновление пакета(ов).

pip install --force-reinstall - при обновлении, переустановить пакет, даже если он последней версии.

Пример проверки, установки и переустановки пакета matplotlib

```
"C:\Program Files\Python36-32\Scripts\pip" install -U matplotlib

Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC
v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import matplotlib as mpl
>>> print ('Current version on matplotlib library is',
                                                    mpl.__version__)
Current version on matplotlib library is 2.1.1
>>>
# де-факто стандарт вызова pyplot в python
import matplotlib.pyplot as plt
```

Модули в Python

Система модулей позволяет логически организовать ваш **код на Python**. Группирование кода в модули значительно облегчает процесс написания и понимания программы. **Модуль в Python** это просто файл, содержащий код на Python. Каждый **модуль в Python** может содержать переменные, объявления классов и функций. Кроме того, в модуле может находиться исполняемый код.

Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python, а например, на C или C++.

Подключение модуля из стандартной библиотеки

Подключить модуль можно с помощью инструкции **import**. К примеру, подключим модуль os для получения текущей директории:

```
>>> import os
>>> os.getcwd()
'C:\\Python33'
```

После ключевого слова **import** указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода. Импортируем модули **time** и **random**.

```
>>>
>>> import time, random
>>> time.time()
1376047104.056417
>>> random.random()
0.9874550833306869
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе e, расположенной в модуле math:

```
>>>
>>> import math
>>> math.e
2.718281828459045
```

Стоит отметить, что если указанный атрибут модуля не будет найден, возбудится исключение **AttributeError**. А если не удастся найти модуль для импортирования, то **ImportError**.

```
>>>
>>> import notexist
Traceback (most recent call last):
  File "", line 1, in
    import notexist
ImportError: No module named 'notexist'
>>> import math
>>> math.Ё
Traceback (most recent call last):
  File "", line 1, in
    math.Ё
AttributeError: 'module' object has no attribute 'Ё'
```

Использование псевдонимов

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова **as**.

```
>>>
>>> import math as m
>>> m.e
2.718281828459045
```

Теперь доступ ко всем атрибутам модуля math осуществляется только с помощью переменной m, а переменной math в этой программе уже не будет (если, конечно, вы после этого не напишете **import** math, тогда модуль будет доступен как под именем m, так и под именем math).

Инструкция from

Подключить определенные атрибуты модуля можно с помощью инструкции **from**. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1>
                            [ as <Псевдоним 1> ],
                [<Атрибут 2> [ as <Псевдоним 2> ] ...]

from <Название модуля> import *
```

Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова **as**.

```
>>>
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(4.6)
5
```

Импортируемые атрибуты можно разместить на нескольких строках, если их много (для лучшей читаемости кода):

```
>>>
>>> from math import (sin, cos,
...                   tan, atan)
```

Второй формат инструкции **from** позволяет подключить все (точнее, почти все) переменные из модуля. Для примера импортируем все атрибуты из модуля `sys`:

```
>>>
>>> from sys import *
>>> version
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43)
[MSC v.1600 32 bit (Intel)]'
>>> version_info
sys.version_info(major=3, minor=3, micro=2,
releaselevel='final', serial=0)
```

Следует заметить, что не все атрибуты будут импортированы. Если в модуле определена переменная **__all__** (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная **__all__** не определена, *то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания*.

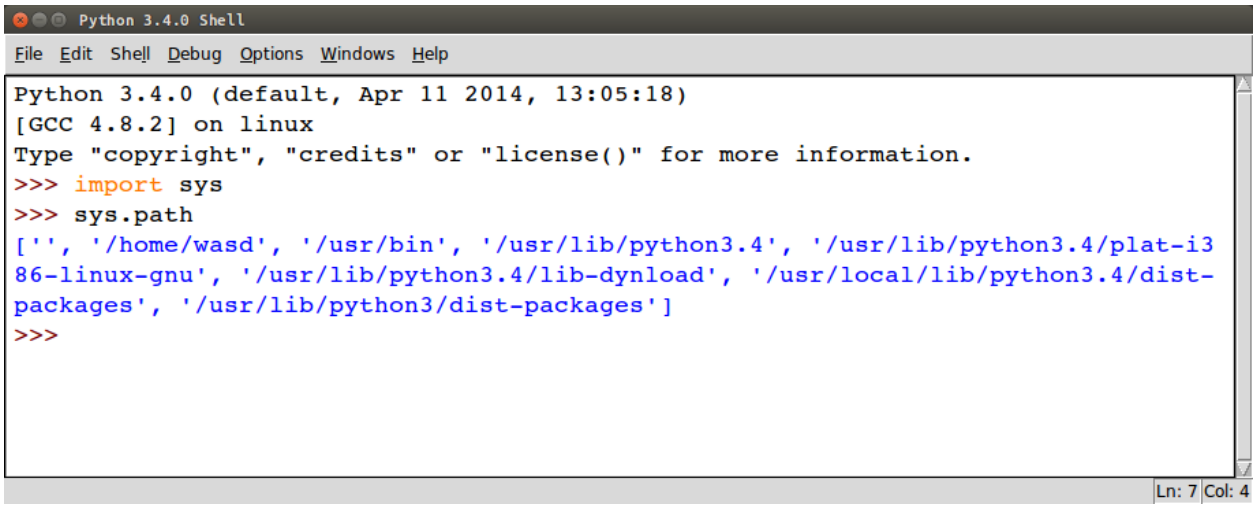
Кроме того, необходимо учитывать, что импортирование всех атрибутов из модуля может нарушить пространство имен главной программы, так как переменные, имеющие одинаковые имена, будут перезаписаны.

Местонахождение модулей в Python:

Когда вы импортируете модуль, интерпретатор Python ищет этот модуль в следующих местах:

1. Директория, в которой находится файл, в котором вызывается команда импорта
2. Если модуль не найден, Python ищет в каждой директории, определенной в консольной переменной PYTHONPATH.
3. Если и там модуль не найден, Python проверяет путь заданный по умолчанию

Путь поиска модулей сохранен в системном модуле sys в переменной path. Переменная **sys.path** содержит все три вышеописанных места поиска модулей.

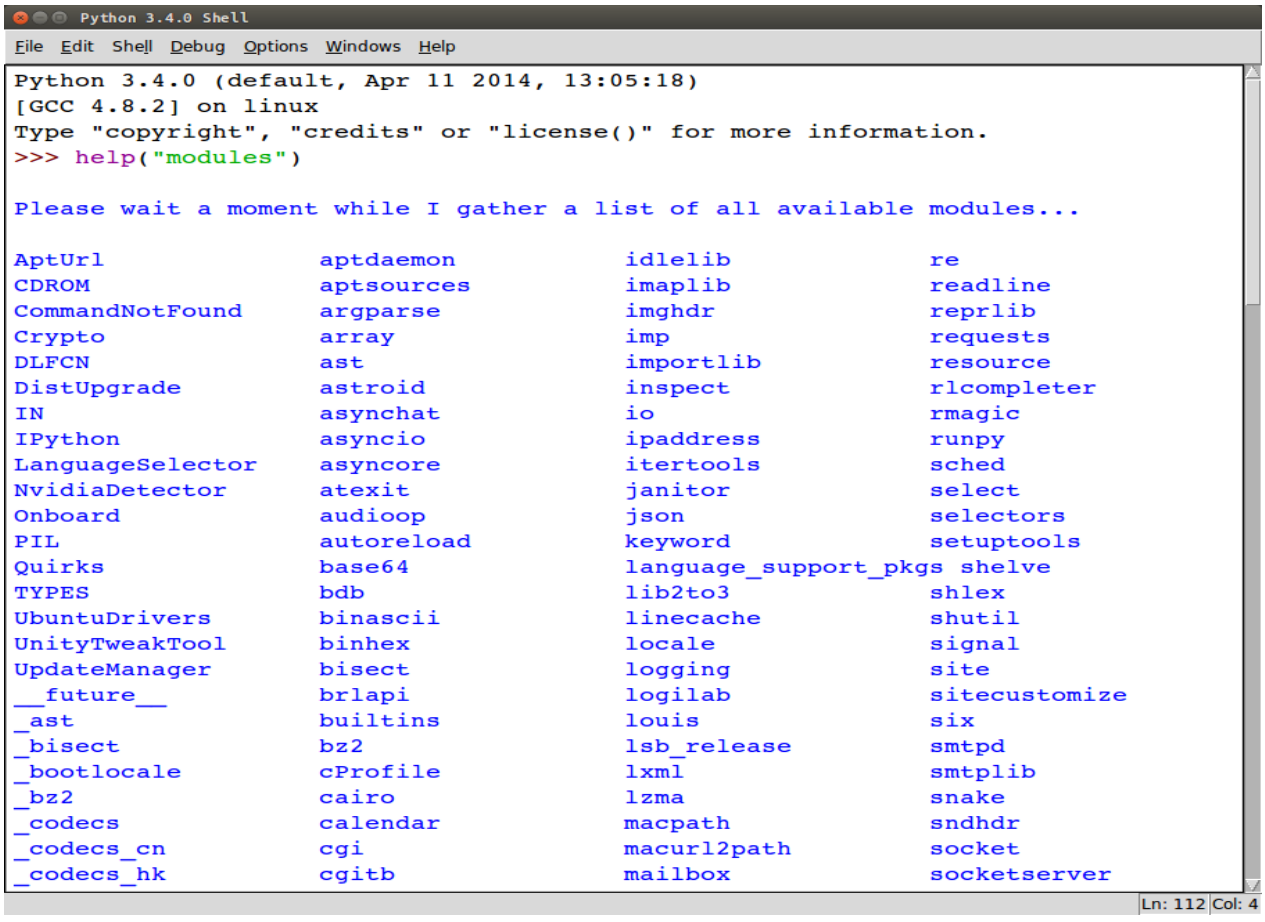


Получение списка всех модулей Python, установленных на компьютере

Для того, чтобы получить список всех модулей, установленных на вашем компьютере достаточно выполнить команду:

```
help("modules")
```

Через несколько секунд вы получите список всех доступных модулей.



Создание своего модуля в Python:

Чтобы создать свой **модуль в Python** достаточно сохранить ваш скрипт с расширением `.py` Теперь он доступен в любом другом файле. Например, создадим два файла: `module_1.py` и `module_2.py` и сохраним их в одной директории. В первом запишем:

```
def hello():
    print ("Hello from module_1")
```

А во втором вызовем эту функцию:

```
from module_1 import hello

hello()
```

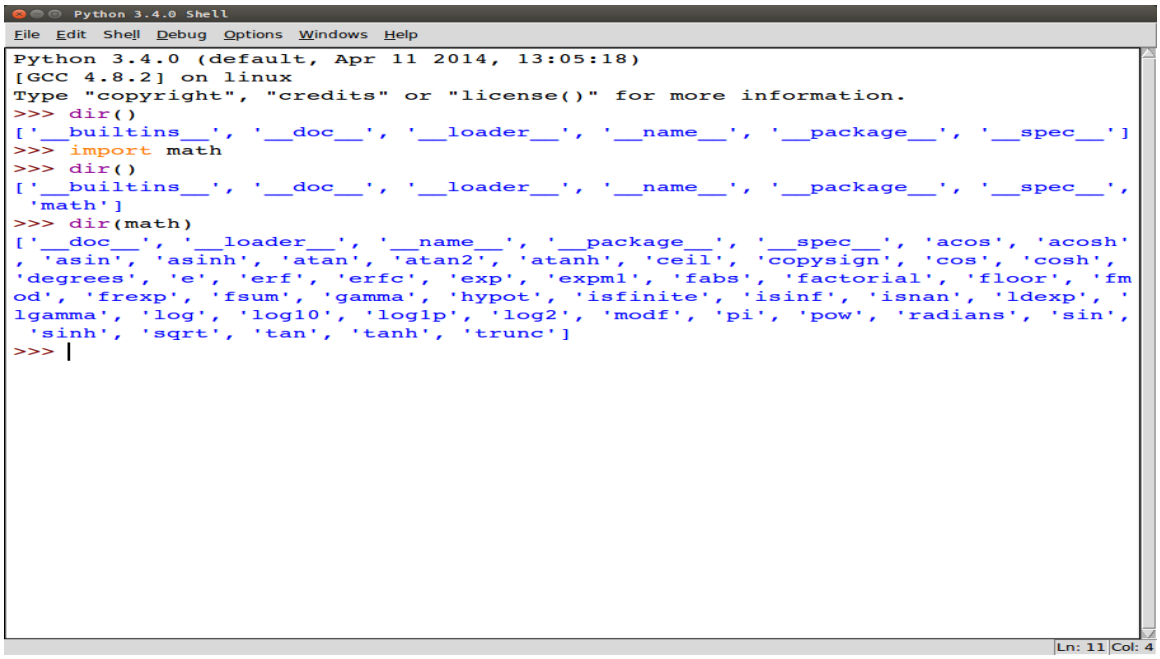
Выполнив код второго файла получим:

```
Hello from module_1
```

Функция dir():

Встроенная **функция `dir()`** возвращает отсортированный список строк, содержащих все имена, определенные в модуле.

```
# на данный момент нам доступны лишь
                                        встроенные функции
dir()
# импортируем модуль math
import math
# теперь модуль math в списке доступных имен
dir()
# получим имена, определенные в модуле math
dir(math)
```



Архитектура программы на Python

Код на Python может быть организован следующим образом:

1. Первый уровень это обычные команды на Python.
2. Команды на Python могут быть собраны в функции.
3. Функции могут быть частью класса.
4. Классы могут быть определены внутри модулей.
5. Наконец, модули могут составляться в пакеты модулей.

Пакеты модулей в Python

Отдельные файлы-модули с кодом на Python могут объединяться в **пакеты модулей**. Пакет это директория (папка), содержащая несколько отдельных файлов-скриптов.

Например, имеем следующую структуру:

```
|_ my_file.py
|_ my_package
         |_ __init__.py
         |_ inside_file.py
```

В файле `inside_file.py` определена некая функция `foo`. Тогда чтобы получить доступ к функции `foo`, в файле `my_file` следует выполнить следующий код:

```
from my_package.inside_file import foo
```

Так же обратите внимание на наличие внутри директории `my_package` файла `__init__.py`. Это может быть пустой файл, который сообщает **Python**, что данная директория является **пакетом модулей**. В Python 3.3 и выше включать файл `__init__.py` в **пакет модулей** стало необязательно, однако, рекомендуется делать это ради поддержки обратной совместимости.

I. NumPy

NumPy — это расширение языка Python, добавляющее поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых математических функций для операций с этими массивами.

NumPy начало работы

NumPy — это библиотека языка Python, добавляющая поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

Установка NumPy

Linux:

```
sudo pip3 install numpy
```

Windows:

```
pip3 install numpy
```

Интересная ссылка
https://pyprog.pro/reference_manual.html

Наиболее важные атрибуты

Основным объектом NumPy является однородный многомерный массив (в **numpy** называется **numpy.ndarray**). Это многомерный массив элементов (обычно чисел), одного типа.

Наиболее важные атрибуты объектов **ndarray**:

ndarray.ndim - число измерений (чаще их называют "оси") массива.

ndarray.shape - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из n строк и m столбов, shape будет (n,m). Число элементов кортежа shape равно ndim.

ndarray.size - количество элементов массива. Очевидно, равно произведению всех элементов атрибута shape.

ndarray.dtype - объект, описывающий тип элементов массива. Можно определить **dtype**, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: bool_, character, int8, int16, int32, int64, **float**8, **float**16, **float**32, **float**64, complex64, object_, так и возможность определить собственные типы данных, в том числе и составные.

ndarray.itemsize - размер каждого элемента массива в байтах.

ndarray.data - буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

Создание массивов

В NumPy существует много способов создать массив.

Один из наиболее простых - создать массив из обычных списков или кортежей Python. Для этого используетя функция **numpy**.**array**() (**array** - функция, создающая объект типа **ndarray**):

```
>>>
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

Функция **array**() трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности (но можно и переопределить его в момент создания).

```
>>>
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Можно также переопределить тип в момент создания:

```
>>>
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]],
                                    dtype=np.complex)
>>> b
array([[ 1.5+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+0.j]])
```

Функция **array()** не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива — **float64**).

Функция

zeros() создает массив из нулей,

а функция

ones() — массив из единиц.

Обе функции принимают кортеж с размерами, и аргумент **dtype**:

```
>>>
>>> np.zeros((3, 5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> np.ones((2, 2, 2))
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]])
```

Функция **eye()** создаёт единичную матрицу (двумерный массив)

```
>>>
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Функция

empty() создает массив без его заполнения. Исходное содержимое случайно и зависит от состояния памяти на момент создания массива (то есть от того мусора, что в ней хранится):

```
>>>
>>> np.empty((3, 3))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920149e-310],
       [  6.93920058e-310,   6.93920058e-310,   6.93920058e-310],
       [  6.93920359e-310,   0.00000000e+000,   6.93920501e-310]])
>>> np.empty((3, 3))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920147e-310],
       [  6.93920149e-310,   6.93920146e-310,   6.93920359e-310],
       [  6.93920359e-310,   0.00000000e+000,   3.95252517e-322]])
```

Полный формат вызова функций:

```
numpy.zeros(shape, dtype = float, order = 'C')
numpy.ones(shape, dtype = float, order = 'C')
numpy.empty(shape, dtype = float, order = 'C')
```

где:

shape - обязательный аргумент, **кортеж** требуемой размерности массива;

dtype - опциональный аргумент, тип элементов массива, по умолчанию **float**;

order - опциональный аргумент, строка, способ представления данных массива в памяти, два возможных значения 'C' и 'F' – «как в C» или «как в фортране».

Для создания последовательностей чисел, в NumPy имеется функция

arange(), аналогичная встроенной в Python **range(),** только вместо списков она возвращает массивы, и принимает не только целые значения:

```
>>>
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 1, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,
0.8,  0.9])
```

Полный формат вызова функции:

numpy.arange([start,] stop[, step,], dtype = None),

где:

start - опциональный аргумент, начальное значение интервала, по умолчанию равен 0;

stop - обязательный аргумент, конечное значение интервала, не входящее в сам интервал, интервал замыкает значение stop – step;

step - опциональный аргумент, шаг итерации, разность между каждым последующим и предыдущим значениями интервала, по умолчанию равен 1;

dtype - тип элементов массива, по умолчанию None, в этом случае тип элементов определяется по типу переданных функции аргуметов start, stop.

Массив может быть создан так же с помощью функции **numpy.linspace()**.

Вообще, при использовании **arange()** с аргументами типа **float**, сложно быть уверенным в том, сколько элементов будет получено (из-за ограничения точности чисел с плавающей запятой). Поэтому, в таких случаях обычно лучше использовать функцию:

linspace(), которая, *вместо шага в качестве одного из аргументов принимает число, равное количеству нужных элементов*:

```
>>>
>>> np.linspace(0, 2, 9) #9 чисел от 0 до 2 включительно
array([  0. ,  0.25,  0.5 ,  0.75, 1. , 1.25, 1.5 , 1.75, 2. ])
```

Полный формат вызова функции:

numpy.**linspace**(start, stop, num = 50,
endpoint = **True**, retstep = **False**),

где:

start - обязательный аргумент, первый член последовательности элементов массива;

stop - обязательный аргумент, последний член последовательности элементов массива;

num - опциональный аргумент, количество элементов массива, по умолчанию равен 50;

endpoint - опциональный аргумент, логическое значение, по умолчанию **True**. Если передано **True**, то stop последний элемент массива. Если установлено в **False**, то последовательность элементов формируется от start до stop для num + 1 элементов, *при этом в возвращаемый массив последний элемент не входит*;

```
>>> d = np.linspace(1.0, 6.0, 5, endpoint = False)
>>> d
array([ 1.,  2.,  3.,  4.,  5.])
```

retstep - опциональный аргумент, логическое значение, по умолчанию **False**. Если передано **True** то, функция возвращает кортеж из двух членов, первый - массив, последовательность элементов, второй - число, приращение между элементами последовательности.

```
>>> f = np.linspace(.5, -.5, 5, retstep = True)
>>> f
(array([ 0.5 ,  0.25,  0.  , -0.25, -0.5 ]), -0.25)
```

fromfunction(): применяет функцию ко всем комбинациям индексов

```
>>>
>>> def f1(i, j):
...     return 3 * i + j
...
>>> np.fromfunction(f1, (3, 4))
array([[ 0.,  1.,  2.,  3.],
       [ 3.,  4.,  5.,  6.],
       [ 6.,  7.,  8.,  9.]])

>>> np.fromfunction(f1, (3, 3))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

Ну и последний из рассматриваемых способов с помощью функций **numpy.zeros_like()**, **numpy.ones_like()**, **numpy.empty_like()**.

Обязательный аргумент, принимаемый функциями - массив, функции возвращают массив такой же структуры, содержащий, соответственно, нули, единицы и то, что оказалось в памяти на момент создания массива.

```
>>> d
array([ 1.,  2.,  3.,  4.,  5.])

>>> dz = np.zeros_like(d)
>>> dz
```

```
array([ 0.,  0.,  0.,  0.,  0.])

>>> do = np.ones_like(d)
>>> do
array([ 1.,  1.,  1.,  1.,  1.])

>>> de = np.empty_like(d)
>>> de
array([  2.24122267e+201,   6.32526950e-317,
                                  0.00000000e+000,
         2.24686637e+201,   6.34874355e-321])
```

Полный формат вызова функций:

numpy.zeros_like(a)
numpy.ones_like(a)
numpy.empty_like(a)

где:

a - обязательный аргумент, массив, структуру которого необходимо повторить;

Массивы можно использовать в различных итерациях:

```
>>> for i in a:
...     print( i )
...
0.1
0.2
0.3
0.4
0.5

>>> for i in a3:
...     for j in i:
...             print( j )
...
[1 2]
[3 4]
[5 6]
[7 8]
[ 9 10]
[11 12]
```

Печать массивов

Если массив слишком большой, чтобы его печатать, NumPy автоматически скрывает центральную часть массива и выводит только его уголки.

```
>>>
>>> print(np.arange(0, 3000, 1))
[   0    1    2 ..., 2997 2998 2999]
```

Если вам действительно нужно увидеть весь массив, используйте функцию
numpy.set_printoptions:

```
np.set_printoptions(threshold=np.nan)
```

И вообще, с помощью этой функции можно настроить печать массивов "под себя".

Функция **numpy.set_printoptions** принимает несколько аргументов:

precision : количество отображаемых цифр после запятой (по умолчанию 8).

threshold : количество элементов в массиве, вызывающее обрезание элементов (по умолчанию 1000).

edgeitems : количество элементов в начале и в конце каждой размерности массива (по умолчанию 3).

linewidth : количество символов в строке, после которых осуществляется перенос (по умолчанию 75).

suppress : если **True**, не печатает маленькие значения в scientific notation (по умолчанию False).

nanstr : строковое представление NaN (по умолчанию 'nan').

infstr : строковое представление inf (по умолчанию 'inf').

formatter : позволяет более тонко управлять печатью массивов. Здесь не рассматриваем – смотрите
https://docs.scipy.org/doc/numpy/reference/generated/numpy.set_printoptions.html

Пользуйтесь официальной документацией по numpy - https://docs.scipy.org/doc/numpy/reference/.

NumPy, базовые операции над массивами

Математические операции над массивами выполняются поэлементно. Создается новый массив, который заполняется результатами действия оператора.

```
>>>
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
```

```
>>> a + b
array([20, 31, 42, 53])
>>> a - b
array([20, 29, 38, 47])
>>> a * b
array([  0,  30,  80, 150])
>>> a / b  # При делении на 0 возвращается inf (бесконечность)
array([        inf,  30.        ,  20.        ,  16.66666667])
<string>:1: RuntimeWarning: divide by zero encountered in true_divide
>>> a ** b
array([     1,     30,   1600, 125000])
>>> a % b  # При взятии остатка от деления на 0 возвращается 0
<string>:1: RuntimeWarning: divide by zero encountered in remainder
array([0, 0, 0, 2])
```

Для этого, естественно, массивы должны быть одинаковых размеров.

```
>>>
>>> c = np.array([[1, 2, 3], [4, 5, 6]])
>>> d = np.array([[1, 2], [3, 4], [5, 6]])
>>> c + d
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: operands could not be broadcast together
with shapes (2,3) (3,2)
```

Также можно производить математические операции между массивом и числом. В этом случае к каждому элементу прибавляется (или что вы там делаете) это число.

```
>>>
>>> a + 1
array([21, 31, 41, 51])
>>> a ** 3
array([  8000,  27000,  64000, 125000])
>>> a < 35  # И фильтрацию можно проводить
array([ True,  True, False, False], dtype=bool)
```

NumPy также предоставляет множество математических операций для обработки массивов:

```
>>>
>>> np.cos(a)
array([ 0.40808206,  0.15425145, -0.66693806,
0.96496603])
>>> np.arctan(a)
array([ 1.52083793,  1.53747533,  1.54580153,
1.55079899])
>>> np.sinh(a)
array([  2.42582598e+08,   5.34323729e+12,
1.17692633e+17,
```

```
       2.59235276e+21])
```

Полный список можно посмотреть
https://docs.scipy.org/doc/numpy/reference/routines.math.html

Список полезных математических функций пакета NumPy

Во всех определениях ниже **a** массив или скаляр.

numpy.**abs**(a) - абсолютное значение a;

numpy.**around**(a, **decimals** = 0, **out** = None) - округляет a до заданного количества десятичных разрядов, по умолчанию до целого. Придерживается следующего правила округления. Если значение a находиться точно по середине между двумя целыми, округление производиться до ближайшего **четного** целого. Так если a равно 1.5 или 2.5 будет возвращено 2, если a равно -0.5 или 0.5 будет возвращено 0.0. Аргумент **decimals** - целое, десятичный разряд после запятой, до которого производиться округление, если **decimals** отрицательное, разряд отсчитывается влево от запятой.

```
Print(np.around(np.array([0.5, 1.8, 2.5, 3.5])))
[ 0.  2.  2.  4.]

print(np.around(np.array([1, 5, 15, 45]), decimals= 1))
[ 1  5 15 45]

Print(np.around(np.array([1, 5, 15, 45]),decimals= -1))
[ 0  0 20 40]
```

Аргумент **out** - массив, в который будет передан результат, структура массива out, должна совпадать со структурой возвращаемого массива. Если None (по умолчанию) будет создан новый массив.

numpy.**fix**(a, **out** = None) - отбрасывает дробную часть a;

numpy.**ceil**(a, **out** = None) - округляет a до меньшего из целых больших или равных a;

```
>>>  print (np.ceil(np.array([-2.7,
                          -1.2, -0.5, 1.8, 2.4, 3.6])))
[-2. -1. -0.  2.  3.  4.]
```

numpy.**floor**(a, **out** = None) - округляет a до большего из целых меньших или равных a;

```
>>> print np.floor(np.array([-2.7, -1.2, -0.5, 1.8,
2.4, 3.6]))
```

```
[-3. -2. -1.  1.  2.  3.]
```

numpy.**sign**(a) - возвращает -1 если a < 0, 0 если a == 0, 1 если a > 0;

numpy.**degrees**(a) - конвертирует a из радиан в градусы;

numpy.**radians**(a) - конвертирует a из градусов в радианы;

numpy.**cos**(a), **numpy**.**sin**(a), **numpy**.**tan**(a) - возвращает косинус, синус, тангенс a. a в радианах;

numpy.**cosh**(a), **numpy**.**sinh**(a), **numpy**.**tanh**(a) - возвращает гиперболические косинус, синус, тангенс a. a в радианах;

numpy.**arccos**(a), **numpy**.**arcsin**(a), **numpy**.**arctan**(a) - возвращает арккосинус, арксинус, арктангенс a в радианах. Для арккосинуса в диапазоне [0, **nump**.**pi**], для арксинуса [-**numpy**.pi/2, **numpy**.pi/2], для арктангенса [-**numpy**.pi/2, **numpy**.pi/2];

numpy.**arccosh**(a), **numpy**.**arcsinh**(a), **numpy**.**arctanh**(a) - возвращает гиперболические косинус, синус, тангенс a. a в радианах;

numpy.**exp**(a) - возвращает основание натурального логарифма (число e) в степени a;

numpy.**log**(a), **numpy**.**log10**(a), **numpy**.**log2**(a) - возвращает натуральный логарифм a, логарифм a по основанию 10, логарифм a по основанию 2;

numpy.**log1p(a**) - возвращает натуральный логарифм a + 1;

numpy.**sqrt(a)** - возвращает положительный квадратный корень a;

numpy.**square**(a) - возвращает квадрат a.

В выражения можно включать несколько массивов. В случае если атрибуты **ndarray**.shape этих массивов совпадают, результат очевиден - действия над массивами будут производиться поэлементно:

```
>>> a1 = np.array([ [1.0, 2.0], [3.0, 4.0] ])
>>> a2 = np.array([ [5.0, 6.0], [7.0, 8.0] ])
>>> a3 = np.array([ [9.0, 10.0], [11.0, 12.0] ])

>>> print (a1 + a2 + a3)
[[ 15.  18.]
 [ 21.  24.]]

>>> print (a1 * a2)
[[  5.  12.]
 [ 21.  32.]]
```

```
>>> print a3 / a1
[[ 9.          5.        ]
 [ 3.66666667  3.        ]]

>>> print (a3 - a1) * a2
[[ 40.  48.]
 [ 56.  64.]]
```

Если атрибуты **ndarray.shape** не совпадают - действия над массивами производятся в соответствии с концепцией транслирования (***broadcasting***).

Операция транслирования - расширение одного или обоих массивов операндов до массивов с равной размерностью.

Для начала несколько примеров:

```
>>> a2 = np.array([1, 2])
>>> print a2
[1 2]
>>> print a2.shape
(2,)

>>> a22 = np.array([ [1, 2], [3, 4] ])
>>> print a22
[[1 2]
 [3 4]]
>>> print a22.shape
(2, 2)

>>> a32 = np.array([ [1, 2], [3, 4], [5, 6] ])
>>> print a32
[[1 2]
 [3 4]
 [5 6]]
>>> print a32.shape
(3, 2)

>>> a222 = np.array([ [ [1, 2], [3, 4] ], [ [5, 6], [7,
8] ] ])
>>> print a222
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
>>> print a222.shape
(2, 2, 2)
```

```
>>> print a22 + a2
[[2 4]
 [4 6]]

>>> print a32 + a2
[[2 4]
 [4 6]
 [6 8]]

>>> print a222 + a2
[[[ 2  4]
  [ 4  6]]

 [[ 6  8]
  [ 8 10]]]

>>> print a32 + a22
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast
to a single shape

>>> print a222 + a22
[[[ 2  4]
  [ 6  8]]

 [[ 6  8]
  [10 12]]]

>>> print a222 + a32
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast
to a single shape
```

И так **правило**. Если длины осей массивов, начиная с замыкающей, попарно равны или в каждой из сравниваемых пар длин хотя бы одна равна единице, то к таким массивам может быть применена операция транслирования.

Длина замыкающей оси - длина массива вложенного наиболее глубоко.

```
>>> a = np.ones((7, 3, 4, 9, 8))
>>> b = np.ones((4, 9, 8))
>>> c = np.ones((4, 3, 8))
```

```
>>> print (a + b).shape
(7, 3, 4, 9, 8)

>>> print (a + c).shape
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast
to a single shape

>>> d = np.ones((9, 7, 1, 6, 1))
>>> e = np.ones((7, 2, 1, 4))
>>> print (d + e).shape
(9, 7, 2, 6, 4)
```

В случае если в массивах присутствуют оси единичной длины, расширяться могут оба массива.

```
>>> f = np.array([1, 2])
>>> print f
[1 2]
>>> print f.shape
(2,)

>>> g = np.array([ [3], [4], [5] ])
>>> print g
[[3]
 [4]
 [5]]
>>> print g.shape
(3, 1)

>>> h = f + g
>>> print h
[[4 5]
 [5 6]
 [6 7]]
>>> print h.shape
(3, 2)
```

Трансляция массивов происходит и при вызове некоторых функций.

Например, функция **numpy**.power(a1, a2), где a1, a2 массив или скаляр, возвращает a1 в степени a2:

```
>>> print np.power(np.array([-2.0, -1.0, 0.0, 2.0, 3.0,
4.0]), 4)
[  16.    1.    0.   16.   81.  256.]
```

```
>>> print np.power(np.array([-2.0, -1.0, 0.0, 2.0, 3.0,
4.0]), -4)
[ 0.0625      1.                 Inf  0.0625
0.01234568  0.00390625]
```

в случае если у переданных массивов не совпадает структура, проводит трансляцию.

```
>>> print np.power(np.array([3, 7]), np.array([ [1],
[2], [3] ]))
[[  3   7]
 [  9  49]
 [ 27 343]]
```

Многие унарные операции, такие как, например, вычисление суммы всех элементов массива, представлены также и в виде методов класса **ndarray**.

```
>>>
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.sum(a)
21
>>> a.sum()
21
>>> a.min()
1
>>> a.max()
6
```

По умолчанию, эти операции применяются к массиву, как если бы он был списком чисел, независимо от его формы. Однако, указав параметр **axis,** можно применить операцию для указанной оси массива:

```
>>>
>>> a.min(axis=0)
                    # Наименьшее число в каждом столбце
array([1, 2, 3])
>>> a.min(axis=1)
                    # Наименьшее число в каждой строке
array([1, 4])
```

Индексы, срезы, итерации

Одномерные массивы осуществляют операции индексирования, срезов и итераций очень схожим образом с обычными списками и другими последовательностями Python (разве что удалять с помощью срезов нельзя).

```
>>>
>>> a = np.arange(10) ** 3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[1]
1
```

```
>>> a[3:7]
array([ 27,  64, 125, 216])
>>> a[3:7] = 8
>>> a
array([  0,   1,   8,   8,   8,   8,   8, 343, 512, 729])
>>> a[::-1]
array([729, 512, 343,   8,   8,   8,   8,   8,   1,   0])
>>> del a[4:6]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: cannot delete array elements
>>> for i in a:
...     print(i ** (1/3))
...
0.0
1.0
2.0
2.0
2.0
2.0
2.0
7.0
8.0
9.0
```

У многомерных массивов на каждую ось приходится один индекс. Индексы передаются в виде последовательности чисел, разделенных запятыми (то есть, **кортежами**):

```
>>>
>>> b = np.array([[ 0,  1,  2,  3],
...               [10, 11, 12, 13],
...               [20, 21, 22, 23],
...               [30, 31, 32, 33],
...               [40, 41, 42, 43]])
...
>>> b[2,3]  # Вторая строка, третий столбец
23
>>> b[(2,3)]
23
>>> b[2][3]  # Можно и так
23
>>> b[:,2]  # Третий столбец
array([ 2, 12, 22, 32, 42])
>>> b[:2]  # Первые две строки
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
>>> b[1:3, : : ]  # Вторая и третья строки
```

```
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Когда индексов меньше, чем осей, отсутствующие индексы предполагаются дополненными с помощью срезов:

```
>>>
>>> b[-1]  # Последняя строка. Эквивалентно b[-1,:]

array([40, 41, 42, 43])
```

b[i] можно читать как b[i, <столько символов ':', сколько нужно>].
В NumPy это также может быть записано *с помощью точек*, как b[i, ...].

Например, если x имеет ранг 5 (то есть у него 5 осей), тогда
x[1, 2, ...] эквивалентно x[1, 2, :, :, :],
x[... , 3] то же самое, что x[:, :, :, :, 3] и
x[4, ... , 5, :] это x[4, :, :, 5, :].

```
>>>
>>> a = np.array(([[0, 1, 2], [10, 12, 13]], [[100, 101, 102],
                                               [110, 112, 113]]))
>>> a.shape
(2, 2, 3)
>>> a[1, ...]  # то же, что a[1, : , :] или a[1]

array([[100, 101, 102],
       [110, 112, 113]])

>>> c[... ,2]  # то же, что a[: , : ,2]
array([[  2,  13],
       [102, 113]])
```

Итерирование многомерных массивов начинается с первой оси:

```
>>>
>>> for row in a:
...     print(row)
...
[[ 0  1  2]
 [10 12 13]]
[[100 101 102]
 [110 112 113]]
```

Однако, если нужно перебрать поэлементно весь массив, как если бы он был одномерным, для этого можно использовать атрибут **flat**:

```
>>>
>>> for el in a.flat:
...     print(el)
```

```
...
0
1
2
10
12
13
100
101
102
110
112
113
```

Манипуляции с формой

Как уже говорилось, у массива есть форма (**shape**), определяемая числом элементов вдоль каждой оси:

```
>>>
>>> a
array([[[  0,   1,   2],
        [ 10,  12,  13]],

       [[100, 101, 102],
        [110, 112, 113]]])
>>> a.shape
(2, 2, 3)
```

Форма массива может быть изменена с помощью различных команд:

```
>>>
>>> a.ravel()  # Делает массив плоским
array([  0,   1,   2,  10,  12,  13, 100, 101, 102, 110, 112, 113])

>>> a.shape = (6, 2)  # Изменение формы
>>> a
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])

>>> a.transpose()  # Транспонирование
array([[  0,   2,  12, 100, 102, 112],
       [  1,  10,  13, 101, 110, 113]])
>>> a.reshape((3, 4))  # Изменение формы
array([[  0,   1,   2,  10],
```

```
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

Порядок элементов в массиве в результате функции **ravel**() соответствует обычному "C-стилю", то есть, чем правее индекс, тем он "быстрее изменяется": за элементом a[0,0] следует a[0,1].

Если одна форма массива была изменена на другую, массив переформировывается также в "C-стиле".

Функции ravel() и reshape() также могут работать (при использовании дополнительного аргумента) в FORTRAN-стиле, в котором быстрее изменяется более левый индекс.

```
>>>
>>> a
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.reshape((3, 4), order='F')
array([[  0, 100,   1, 101],
       [  2, 102,  10, 110],
       [ 12, 112,  13, 113]])
```

Метод **reshape**() возвращает ее аргумент с измененной формой, в то время как метод **resize**() изменяет сам массив:

```
>>>
>>> a.resize((2, 6))
>>> a
array([[  0,   1,   2,  10,  12,  13],
       [100, 101, 102, 110, 112, 113]])
```

Если при операции такой перестройки один из аргументов задается как -1, то он автоматически рассчитывается в соответствии с остальными заданными:

```
>>>
>>> a.reshape((3, -1))
array([[  0,   1,   2,  10],
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

Объединение массивов

Несколько массивов могут быть объединены вместе вдоль разных осей с помощью функций **hstack** и **vstack**.

hstack() объединяет массивы по первым осям,

vstack() — по последним:

```
>>>
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> np.vstack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.hstack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Функция **column_stack()** объединяет одномерные массивы в качестве столбцов двумерного массива:

```
>>>
>>> np.column_stack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Аналогично для строк имеется функция

row_stack().

```
>>>
>>> np.row_stack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

Функция **concatenate()** соединяет массивы вдоль указанной оси.

np.concatenate((a1, a2, ..., aN), axis=0)

Параметры: a1, a2, ..., aN - последовательность подобных массиву объектов. Любые объекты которые могут быть преобразованы в массивы NumPy. Данные объекты должны иметь одинаковую форму (количество осей), но не размер по указанной оси

axis - целое число (*необязательный*). Определяет ось вдоль которой соединяются массивы. По умолчанию *axis=0*, что соответствует первой оси.

Возвращает: ndarray - массив NumPy - массив который состоит из указанных массивов, соединенных вдоль указанной оси.

Разбиение массива

Используя **hsplit()** вы можете разбить массив вдоль горизонтальной оси, указав либо число возвращаемых массивов одинаковой формы, либо номера столбцов, после которых массив разрезается "ножницами":

```
>>>
>>> a = np.arange(12).reshape((2, 6))
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> np.hsplit(a, 3)  # Разбить на 3 части
[array([[0, 1], [6, 7]]),
 array([[2, 3], [8, 9]]),
 array([[ 4,  5], [10, 11]])]
>>> np.hsplit(a, (3, 4))  # Разрезать a после третьего
                                    и четвёртого столбца
[array([[0, 1, 2], [6, 7, 8]]),
 array([[3], [9]]),
 array([[ 4,  5], [10, 11]])]
```

Функция
vsplit() разбивает массив вдоль вертикальной оси, а
array_split() позволяет указать оси, вдоль которых произойдет разбиение.

Копии и представления

При работе с массивами, их данные иногда необходимо копировать в другой массив, а иногда нет. Это часто является источником путаницы. *Возможно 3 случая*:

Вообще никаких копий

Простое присваивание не создает ни копии массива, ни копии его данных:

```
>>>
>>> a = np.arange(12)
>>> b = a  # Нового объекта создано не было
>>> b is a  # a и b это два имени для одного и того же
объекта ndarray
True
>>> b.shape = (3,4)  # изменит форму a
>>> a.shape
(3, 4)
```

Python передает изменяемые объекты как ссылки, поэтому вызовы функций также не создают копий.

Представление или поверхностная копия

Разные объекты массивов могут использовать одни и те же данные. Метод **view**() создает новый объект массива, являющийся представлением тех же данных.

```
>>>
>>> c = a.view()
>>> c is a
False
>>> c.base is a  # c это представление данных,
принадлежащих a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = (2,6)  # форма a не поменяется
>>> a.shape
(3, 4)
>>> c[0,4] = 1234  # данные a изменятся
>>> a
array([[  0, 1, 2, 3],
       [1234, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

Срез массива это представление:

```
>>>
>>> s = a[:,1:3]
>>> s[:] = 10
>>> a
array([[  0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

Глубокая копия

Метод **copy()** создаст настоящую копию массива и его данных:

```
>>>
>>> d = a.copy()  # создается новый объект массива с
                                        новыми данными
>>> d is a
False
>>> d.base is a  # d не имеет ничего общего с a
False
>>> d[0, 0] = 9999
>>> a
array([[  0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

NumPy случайные числа

Рассмотрим, как создавать массивы из случайных элементов и как работать со случайными элементами в NumPy.

Списки в массивы

Создавать списки, используя встроенный модуль **random**, а затем преобразовывать их в **numpy**.**array**:

```
>>>
>>> import numpy as np
>>> import random

>>> np.array([random.random() for i in range(10)])

array([ 0.99538667,  0.16860511,  0.78952804,
                     0.09676316,  0.86110208,
        0.89674666,  0.56401347,  0.63431468,
                         0.51110935,  0.64944844])
```

Но есть способ лучше.

Модуль numpy.random

Для создания массивов со случайными элементами служит модуль **numpy.random**.

```
>>>
>>> import numpy as np  # Импортировать numpy
                                  и писать np.random
>>> np.random
<module 'numpy.random' from <module 'numpy.random' from
'C:\\Program Files\\Python36\\lib\\site-
packages\\numpy\\random\\__init__.py'>
>>> import numpy.random as rand  # Можно и присвоить
                                 отдельное имя. Вопрос вкуса
>>> rand
<module 'numpy.random' from 'C:\\Program Files\\Python36\\lib\\site-
packages\\numpy\\random\\__init__.py'>
```

Создание массивов со случайными элементами

Самый простой способ задать массив со случайными элементами - использовать функцию **sample** (или **random**, или **random_sample**, или **ranf** - это всё одна и та же функция).

```
>>>
>>> np.random.sample()
0.6336371838734877
>>> np.random.sample(3)
```

```
array([ 0.53478558,  0.1441317 ,  0.15711313])
>>> np.random.sample((2, 3))
array([[ 0.12915769,  0.09448946,  0.58778985],
       [ 0.45488207,  0.19335243,  0.22129977]])
```

Без аргументов возвращает просто число в промежутке [0, 1),

- с одним целым числом - одномерный массив,

- с кортежем - массив с размерами, указанными в кортеже (все числа - из промежутка [0, 1)).

С помощью функции **randint** или **random_integers** можно создать массив из целых чисел.

Аргументы: **low, high, size:** от какого, до какого числа (**randint** не включает в себя это число, а **random_integers** включает), и **size** - размеры массива.

```
>>>
>>> np.random.randint(0, 3, 10)
array([0, 2, 0, 1, 1, 0, 2, 2, 2, 0])
>>> np.random.random_integers(0, 3, 10)
array([2, 2, 3, 3, 1, 1, 0, 2, 3, 2])
>>> np.random.randint(0, 3, (2, 10))
array([[0, 1, 2, 0, 0, 0, 1, 1, 1, 2],
       [0, 0, 2, 2, 2, 0, 1, 2, 2, 1]])
```

Также можно генерировать числа согласно различным распределениям (Гаусса, Парето и другие). Чаще всего нужно равномерное распределение, которое можно получить с помощь функции **uniform**.

```
>>>
>>> np.random.uniform(2, 8, (2, 10))
array([[ 3.1517914 ,  3.10313483,  2.84007134,
                        3.21556436,  4.64531786,
         2.99232714,  7.03064897,  4.38691765,
                        5.27488548,  2.63472454],
       [ 6.39470358,  5.63084131,  4.69996748,
                        7.07260546,  7.44340813,
         4.10722203,  7.52956646,  4.8596943 ,
                       3.97923973,  5.64505363]])
```

Выбор и перемешивание

Перемешать NumPy массив можно с помощью функции **shuffle**:

```
>>>
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.random.shuffle(a)
>>> a
array([2, 8, 7, 3, 5, 0, 4, 9, 1, 6])
```

Также можно перемешать массив с помощью функции **permutation** (она, в отличие от **shuffle**, возвращает перемешанный массив). Также она, вызванная с одним аргументом (целым числом), возвращает перемешанную последовательность от 0 до N.

```
>>>
>>> np.random.permutation(10)
array([1, 2, 3, 8, 7, 9, 4, 6, 5, 0])
```

Сделать случайную выборку из массива можно с помощью функции **choice**.

numpy.random.choice(a, **size**=None, **replace**=**True**, **p**=None)

- a : одномерный массив или число. Если массив, будет производиться выборка из него. Если число, то выборка будет производиться из **np.arange**(a).
- size : размерности массива. Если None, возвращается одно значение.
- replace : если **True**, то одно значение может выбираться более одного раза.
- p : вероятности. Это означает, что элементы можно выбирать с неравными вероятностями. Если не заданы, используется равномерное распределение.

```
>>>
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.random.choice(a, 10, p=[0.5, 0.25, 0.25, 0, 0,
                               0, 0, 0, 0, 0])
array([0, 0, 0, 0, 1, 2, 0, 0, 1, 1])
```

Инициализация генератора случайных чисел

seed(число) - инициализация генератора.

```
>>>
>>> np.random.seed(1000)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                    0.4821914 ,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,
                             0.2331322 ,  0.84174072])
>>> np.random.seed(1000)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                             0.4821914 ,  0.87247454,
```

```
        0.21233268,  0.04070962,  0.39719446,
                           0.2331322 ,  0.84174072])
```

get_state и **set_state** - возвращают и устанавливают состояние генератора.

```
>>>
>>> np.random.seed(1000)
>>> state = np.random.get_state()
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                              0.4821914 ,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,
                           0.2331322 ,  0.84174072])
>>> np.random.set_state(state)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                            0.4821914 ,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,
                          0.2331322 ,  0.84174072])
```

Некоторые полезные функции

numpy.min(a, axis = None, **out** = None),
numpy.max(a, axis = None, **out** = None)

возвращает минимальное, максимальное значение элементов массива соответственно:

```
>>> import numpy as np
>>> print np.min(np.array([ [1.0, -0.5, 3.0], [4.0,
3.0, -0.5] ]))
-0.5
```

axis - опциональный аргумент, индекс оси (измерения) массива по которому проводится поиск минимального, максимального значения. Под индексом оси понимается индекс в кортеже **ndarray.shape.**

```
>>> a = np.array([ [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0]
], [ [7.0, 8.0, 9.0], [11, 12, 13] ] ])
>>> print a
[[[  1.   2.   3.]
  [  4.   5.   6.]]

 [[  7.   8.   9.]
  [ 11.  12.  13.]]]
>>> print a.shape
(2, 2, 3)
```

```
>>> print np.min(a, axis = 0)
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

>>> print np.min(a, axis = 1)
[[ 1.  2.  3.]
 [ 7.  8.  9.]]

>>> print np.min(a, axis = 2)
[[  1.   4.]
 [  7.  11.]]
```

out - опциональный аргумент, массив, в который будет помещен результат. Структура массива **out** должна соответствовать структуре массива результата. Если **out** = None (по умолчанию) будет создан новый массив.

numpy.argmin(a, axis = None),
numpy.argmax(a, axis = None) -
возвращает индекс минимального, максимального значения элементов массива соответственно:

```
>>> print np.argmin(np.array([ [1.0, -0.5, 3.0], [4.0,
3.0, -0.5] ]))
1

>>> print np.argmin(a, axis = 0)
[[0 0 0]
 [0 0 0]]

>>> print np.argmin(a, axis = 1)
[[0 0 0]
 [0 0 0]]

>>> print np.argmin(a, axis = 2)
[[0 0]
 [0 0]]
```

numpy.sum(a, axis = None, **dtype** = None, **out** = None),
numpy.prod(a, axis = None, **dtype** = None, **out** = None)

возвращает сумму, произведение элементов массива соответственно:

```
>>> print np.sum(np.array([ [1.0, -0.5, 3.0], [4.0,
3.0, -0.5] ]))
10.0
>>> print np.sum(a, axis = 0)
```

```
[[  8.  10.  12.]
 [ 15.  17.  19.]]
>>> print np.sum(a, axis = 1)
[[  5.   7.   9.]
 [ 18.  20.  22.]]
>>> print np.sum(a, axis = 2)
[[  6.  15.]
 [ 24.  36.]]
```

NumPy linalg некоторые операции линейной алгебры

Рассмотрим модуль **numpy**.**linalg**, позволяющий делать многие операции из линейной алгебры.

Возведение в степень

linalg.matrix_power(M, n) - возводит матрицу в степень n.

Разложения

linalg.cholesky(a) - разложение Холецкого.
linalg.qr(a[, mode]) - QR разложение.
linalg.svd(a[, full_matrices, compute_uv]) - сингулярное разложение.

Некоторые характеристики матриц

linalg.eig(a) - собственные значения и собственные векторы.
linalg.norm(x[, ord, axis]) - норма вектора или оператора.
linalg.cond(x[, p]) - число обусловленности.
linalg.det(a) - определитель.
linalg.slogdet(a) - знак и логарифм определителя (для избежания переполнения, если сам определитель очень маленький).

Системы уравнений

linalg.solve(a, b) - решает систему линейных уравнений Ax = b.
linalg.tensorsolve(a, b[, axes]) - решает тензорную систему линейных уравнений Ax = b.
linalg.lstsq(a, b[, rcond]) - метод наименьших квадратов.
linalg.inv(a) - обратная матрица.

Замечания:

- **linalg.LinAlgError** - исключение, вызываемое данными функциями в случае неудачи (например, при попытке взять обратную матрицу от вырожденной).
- Подробная документация: https://docs.scipy.org/doc/**numpy**/reference/routines.linalg.html
- Массивы большей размерности в большинстве функций **linalg** интерпретируются как набор из нескольких массивов нужной

размерности. Таким образом, можно одним вызовом функции проделывать операции над несколькими объектами.

```
>>>
>>> a = np.arange(18).reshape((2,3,3))
>>> a
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]]])

>>> np.linalg.det(a)
array([ 0.,  0.])
```

Исключения numpy.linalg.LinAlgError

numpy.linalg.LinAlgError
exception numpy.linalg.LinAlgError

Объект linalg.LinAlgError генерирует исключения Python, вызванные функциями модуля linalg.

Данный объект образован классом исключений общего назначения Python и является производным от него. Данный класс вызывается, только в том случае если дальнейшая работа какой-либо функции модуля *linalg* невозможна.

Примеры

```
>>> import numpy as np
>>> from numpy import linalg as LA
>>>
>>> a = [[0, 0], [0, 0]]
>>>
>>> try:
...       LA.inv(a)
... except LA.LinAlgError:
...       print('Матрица "a" является либо вырожденной либо прямоугольной')
...
```

Матрица "a" является либо вырожденной либо прямоугольной

Примеры

Произведение одномерных массивов представляет собой скалярное произведение векторов:

```
>>> a = np.array([1,2])
>>> b = np.array([3,4])
```

```
>>>
>>> np.dot(a,b)
11
```

Произведение двумерных массивов по правилам линейной алгебры так же возможно:

```
>>> a = np.arange(2,6).reshape(2,2)
>>> a
array([[2, 3],
       [4, 5]])
>>>
>>> b = np.arange(6,10).reshape(2,2)
>>> b
array([[6, 7],
       [8, 9]])
>>>
>>> np.dot(a,b)
array([[36, 41],
       [64, 73]])
```

При этом размеры матриц (массивов) должны быть либо равны, а сами матрицы квадратными, либо быть согласованными, т.е. если размеры матрицы A равны [m,k], то размеры матрицы B должны быть равны [k,n]:

```
>>> a = np.arange(2,8).reshape(2,3)
>>> a
array([[2, 3, 4],
       [5, 6, 7]])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4, 5],
       [6, 7],
       [8, 9]])
>>>
>>> np.dot(a,b)
array([[ 58,  67],
       [112, 130]])
```

Так же по правилам умножения матриц, мы можем умножить матрицу на вектор (одномерный массив). При этом в таком умножении вектор столбец должен находиться справа, а вектор строка слева:

```
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4, 5],
       [6, 7],
       [8, 9]])
>>>
```

```
>>> np.dot(a,b)
array([40, 46])
>>>
>>> a = np.arange(1,3).reshape(2,1)
>>> a
array([[1],
       [2]])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4, 5],
       [6, 7],
       [8, 9]])
>>>
>>> np.dot(b,a)
array([[14],
       [20],
       [26]])
```

Квадратные матрицы можно возводить в степень n т.е. умнажать сами на себя n раз:

```
>>> a = np.arange(1,5).reshape(2,2)
>>> a
array([[1, 2],
       [3, 4]])
>>>
>>> np.dot(a,a)     #  Равносильно a**2
array([[ 7, 10],
       [15, 22]])
>>>
>>> np.linalg.matrix_power(a,2)
array([[ 7, 10],
       [15, 22]])
>>>
>>> np.linalg.matrix_power(a,5)
array([[1069, 1558],
       [2337, 3406]])
>>>
>>> np.linalg.matrix_power(a,0)
array([[1, 0],
       [0, 1]])
```

Довольно часто приходится вычислять ранг матриц:

```
>>> a = np.arange(1,10).reshape(3,3)
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>>
>>> np.linalg.matrix_rank(a)
```

```
2
>>>
>>> b = np.arange(1,24,2).reshape(3,4)
>>> b
array([[ 1,  3,  5,  7],
       [ 9, 11, 13, 15],
       [17, 19, 21, 23]])
>>>
>>> np.linalg.matrix_rank(b)
2
```

Еще чаще приходится вычислять определитель матриц ,хотя результат вас может немного удивить:

```
>>> a = np.array([[1,3],[4,3]])
>>> a
array([[1, 3],
       [4, 3]])
>>>
>>> np.linalg.det(a)
-8.999999999999982
>>>
>>> 1*3 - 3*4      #  Результат должен быть целым числом
-9
```

В данном случае, из-за двоичной арифметики, результат не целое число и округлять до ближайшего целого прийдется вручную. Это связано с тем, что алгоритм вычисления определителя использует LU-разложение - это намного быстрее чем обычный алгоритм, но за скорость все же приходится немного заплатить ручным округлением (конечно, если таковое требуется):

```
>>> np.linalg.det(a)
-8.999999999999982
>>> round(np.linalg.det(a))
-9.0
>>>
>>> b = np.arange(1,48,3).reshape(4,4)
>>> np.linalg.det(b)
-1.0223637331664275e-27
>>> round(np.linalg.det(b))
-0.0
```

Транспонирование матриц:

```
>>> a
matrix([[1, 3],
        [4, 3]])
>>>
>>> a.T
matrix([[1, 4],
        [3, 3]])
>>>
>>> b
array([[ 1,  4,  7, 10],
       [13, 16, 19, 22],
       [25, 28, 31, 34],
```

```
      [37, 40, 43, 46]])
>>>
>>> b.T
array([[ 1, 13, 25, 37],
      [ 4, 16, 28, 40],
      [ 7, 19, 31, 43],
      [10, 22, 34, 46]])
```

Вычисление обратных матриц:

```
>>> a = np.array([[1,3],[4,3]])
>>> a
array([[1, 3],
      [4, 3]])
>>>
>>> b = np.linalg.inv(a)
>>> b
array([[-0.33333333,  0.33333333],
      [ 0.44444444, -0.11111111]])
>>>
>>> np.dot(a,b)
array([[ 1.,  0.],
      [ 0.,  1.]])
```

Решение систем линейных уравнений:

```
... #  система из двух линейных уравнений:
...
... #  1*x1 + 5*x2 = 11
... #  2*x1 + 3*x2 = 8
...
>>> a = np.array([[1,5],[2,3]])
>>> b = np.array([11,8])
>>>
>>> x = np.linalg.solve(a,b)
>>> x
array([ 1.,  2.])
>>>
>>> np.dot(a,x)
array([ 11.,   8.])
```

NumPy Преобразование фурье

По-простому, **преобразование Фурье** — разложение некоторого сигнала на гармонические (синусы или косинусы) колебания (спектр) (по материалам http://old.pynsk.ru/posts/2015/Nov/09/matematika-v-python-preobrazovanie-fure/).

Преобразование Фурье является **интегральным преобразованием**. Если речь идёт о дискретном сигнале, то интеграл обращается в сумму (и становится дискретным преобразованием Фурье, ДПФ). Чтобы посчитать такую сумму N элементов, надо совершить N^2 операций с комплексными

числами. Но достаточно давно (Cooley, Tukey, 1965 г, а ещё сам Гаусс в 1805 г.) придумал алгоритм, вычисляющий ДПФ N элементов в N*log(N) операций (большая часть из которых над действительными числами), что существенно экономит вычислительное время. Такой алгоритм часто называют «быстрое преобразование Фурье, БПФ (Fast Fourier Trans**for**m, FFT)». Именно так реализовано ДПФ в современных компьютерных программах.

В библиотеке NumPy содержится все, что нужно для дискретного преобразования Фурье. Всё это лежит в модуле **`numpy.fft`**.

Вот эти функции.

Общий случай: сигнал может быть как из действительных чисел, так и из комплексных.

`fft(a, n=None, axis=-1)` — прямое одномерное ДПФ.

`ifft(a, n=None, axis=-1)` — обратное одномерное ДПФ.

Здесь:

`a` — "сигнал", входной массив (массив `numpy array` или даже питоновский `список` или `кортеж`, если в нём только числа).

Массив может быть и многомерным, тогда будет вычисляться много `ОДНОМЕРНЫХ ПФ` по строкам (по умолчанию) или столбцам, в зависимости от параметра `axis`.

Например, `a` — двухмерный, `a[n][m]`:

при `axis=1` или `-1` будет такое (под `fourier(a...)` понимается результат действия ПФ на `a`):

```
[fourier(a[0][j]), fourier(a[1][j]), ...
                                    foirier(a[n][j])]
```

При axis=0 такое:

```
[fourier(a[i][0]), fourier(a[i][1]), ...
                                      foirier(a[i][m])]
```

`n` — сколько элементов массива брать. Если меньше длины массива, то обрезать, если больше, то дополнить нулями, по умолчанию `len(a)`.

`fft2(a, s=None, axes=(-2, -1))` — прямое двухмерное ПФ.

`ifft2(a, s=None, axes=(-2, -1))` — обратное двухмерное ПФ.

`fftn(a, s=None, axes=None)` — прямое многомерное ПФ.

`ifftn(a, s=None, axes=None)` — обратное многомерное ПФ.

Всё так же, как и для одномерных, но **`s`** и **`axes`** теперь кортежи для каждой размерности. О размерности **`fftn`**, **`ifftn`** догадаются по размерности входных массивов или **`s`** и **`axes`**.

Когда сигнал действительный (`real`) (пожалуй, самый распространённый случай):

`rfft(a, n=None, axis=-1`) — прямое одномерное ДПФ (для действительных чисел).

irfft(a, n=None, axis=-1) — обратное одномерное ДПФ.

rfft2(a, s=None, axes=(-2, -1)) — прямое двухмерное ДПФ.

irfft2(a, s=None, axes=(-2, -1)) — обратное двухмерное ДПФ.

rfftn(a, s=None, axes=None) — прямое многомерное ДПФ.

irfftn(a, s=None, axes=None) — обратное многомерное ДПФ.

Всё так же, как и для общего случая.

Все эти функции возвращают массив соответствующей размерности, в котором записан результат ДПФ.

Разница такая. Если длина входного массива (или какой-либо его размерности) **N**, то в общем случае (с комплексным сигналом) длина выходного массива **N**.

Там содержатся сначала положительные частоты от нуля до частоты Котельникова (Найквиста), потом отрицательные в порядке возрастания.

В случае действительного сигнала отрицательные частоты полностью симметричны положительным, и тогда нет нужды их записывать: длина выходного массива **N/2+1**, частоты от нуля до частоты Котельникова.

Если спектр сигнала действительный (а сигнал обладает "эрмитовой симметрией": его половины симметричны относительно центра по модулю и являются комплексно сопряжёнными друг другу), то можно применить такие функции:

hfft(a, n=None, axis=-1) — прямое одномерное ДПФ.

ihfft(a, n=None, axis=-1) — обратное одномерное ДПФ.

Длина входного массива **N**, а выходного **2*N+1**.

Кроме того, есть вспомогательные функции (будет понятнее из примера):

fftfreq(n, d=1.0) — возвращает частоты для выходных массивов функций fft*.

rfftfreq(n, d=1.0) — возвращает частоты для выходных массивов функций rfft*.

Здест - **n** — длина входного массива, **d** — период дискретизации (обратная частота дискретизации).

fftshift(x, axes=None) — преобразует массив (с результатом ДПФ, от функций fft*) так, чтобы нулевая частота была в центре.

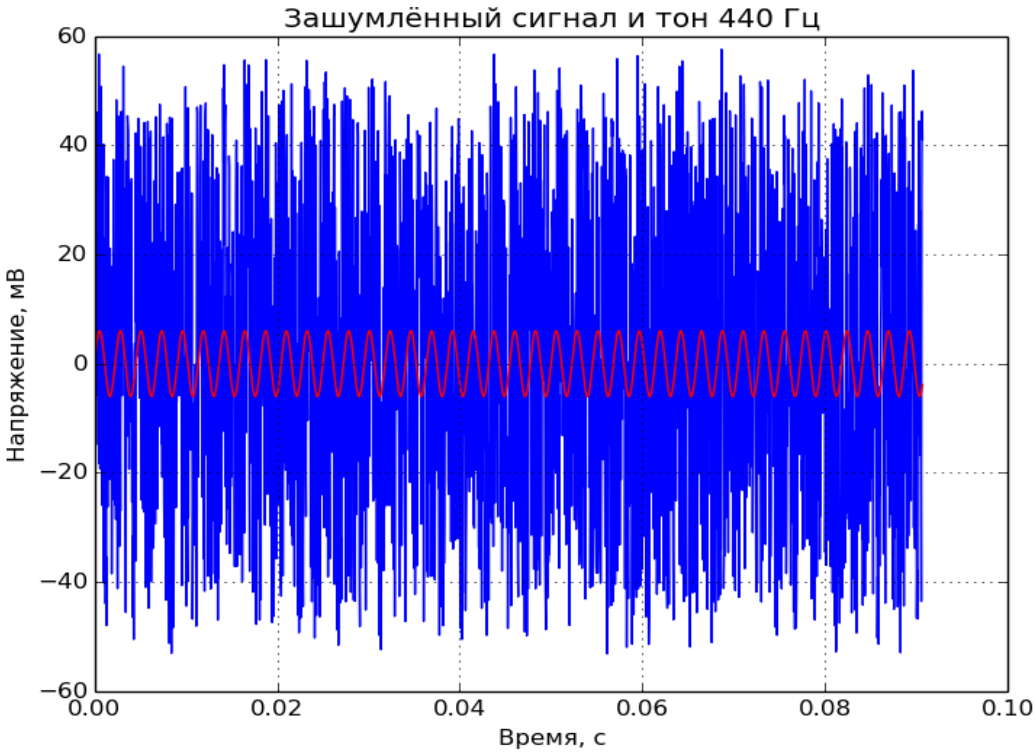
ifftshift(x, axes=None) — делает обратную операцию.

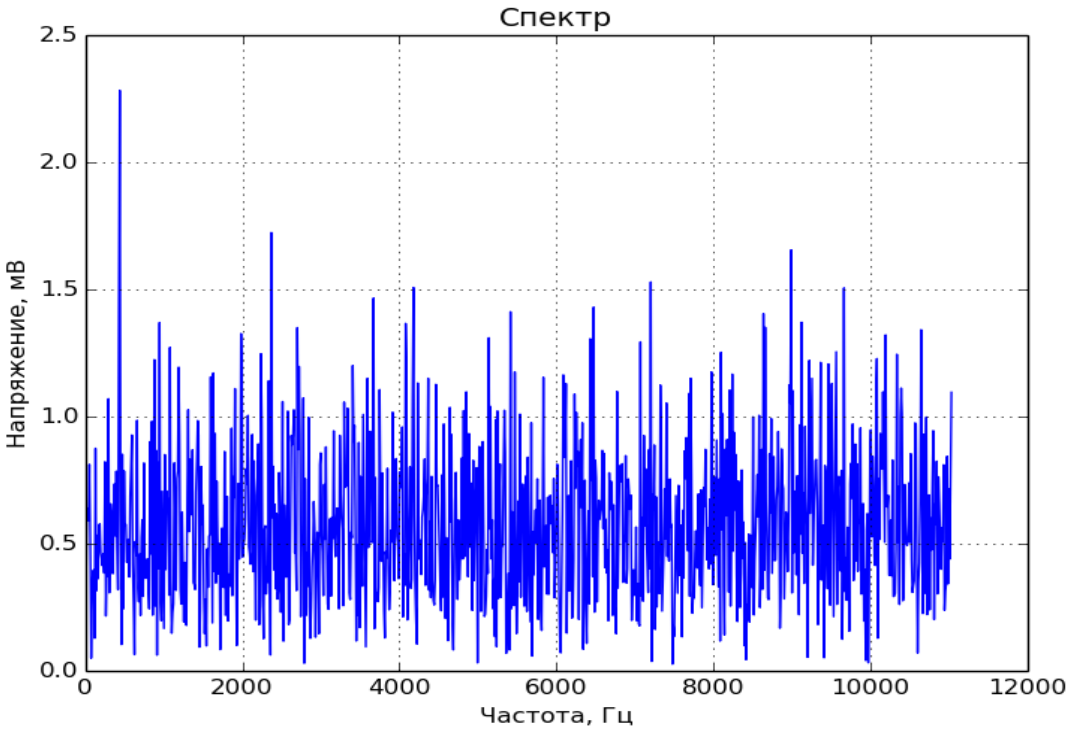
Приведём такой пример. Допустим, записали мы микрофоном какой-то шум, и надо определить, есть ли там какой-нибудь тон.

```
from numpy import array, arange, abs as np_abs
from numpy.fft import rfft, rfftfreq
from numpy.random import uniform
from math import sin, pi
```

```
import matplotlib.pyplot as plt
# а можно импортировать numpy и писать: numpy.fft.rfft
FD = 22050 # частота дискретизации, отсчётов в секунду
# а это значит, что в дискретном сигнале представлены
# частоты от нуля до 11025 Гц (это и есть теорема
# Котельникова)
N = 2000  # длина входного массива, 0.091 секунд при
          #такой частоте дискретизации
# сгенерируем сигнал с частотой 440 Гц длиной N
pure_sig =
   array([6.*sin(2.*pi*440.0*t/FD) for t in range(N)])
# сгенерируем шум, тоже длиной N (это важно!)
noise = uniform(-50.,50., N)
# суммируем их и добавим постоянную составляющую 2 мВ
#(допустим, не очень хороший микрофон попался.
# Или звуковая карта или АЦП)
sig = pure_sig + noise + 2.0
# в numpy так перегружена функция сложения
# вычисляем преобразование Фурье.
#Сигнал действительный, поэтому надо
# использовать rfft, это быстрее, чем fft
spectrum = rfft(sig)

# нарисуем всё это, используя matplotlib
# Сначала сигнал зашумлённый и тон отдельно
plt.plot(arange(N)/float(FD), sig) # по оси времени
#                                              секунды!
plt.plot(arange(N)/float(FD), pure_sig, 'r') # чистый
#                         сигнал будет нарисован красным
plt.xlabel(u'Время, с')
plt.ylabel(u'Напряжение, мВ')
plt.title(u'Зашумлённый сигнал и тон 440 Гц')
plt.grid(True)
plt.show()
# когда закроется этот график, откроется следующий
# Потом спектр
plt.plot(rfftfreq(N, 1./FD), np_abs(spectrum)/N)
# rfftfreq сделает всю работу по преобразованию
#                    номеров элементов массива в герцы
# нас интересует только спектр амплитуд,
# поэтому используем abs из numpy
#   (действует на массивы поэлементно)

# делим на число элементов, чтобы амплитуды были в
# милливольтах, а не в суммах Фурье.
```

```
# Проверить просто — постоянные составляющие должны
# совпадать в сгенерированном сигнале и в спектре

plt.xlabel(u'Частота, Гц')
plt.ylabel(u'Напряжение, мВ')
plt.title(u'Спектр')
plt.grid(True)
plt.show()
```




Спектр
Напряжение, мВ
Частота, Гц
0.0
0.5
1.0
1.5
2.0
2.5
0
2000
4000
6000
8000
10000
12000

II. Matplotlib

Введение

matplotlib - набор дополнительных модулей (библиотек) языка Python. Предоставляет средства для построения самых разнообразных 2D графиков и диаграмм данных. Достоинства этой библиотеки - простота использования. Для построения весьма мудреных и красочно оформленных диаграмм достаточно нескольких строк кода. При этом качество получаемых изображений более чем достаточно для их опубликования. Работа этого модуля обычно подразумевает так же использование модуля **NumPy**.

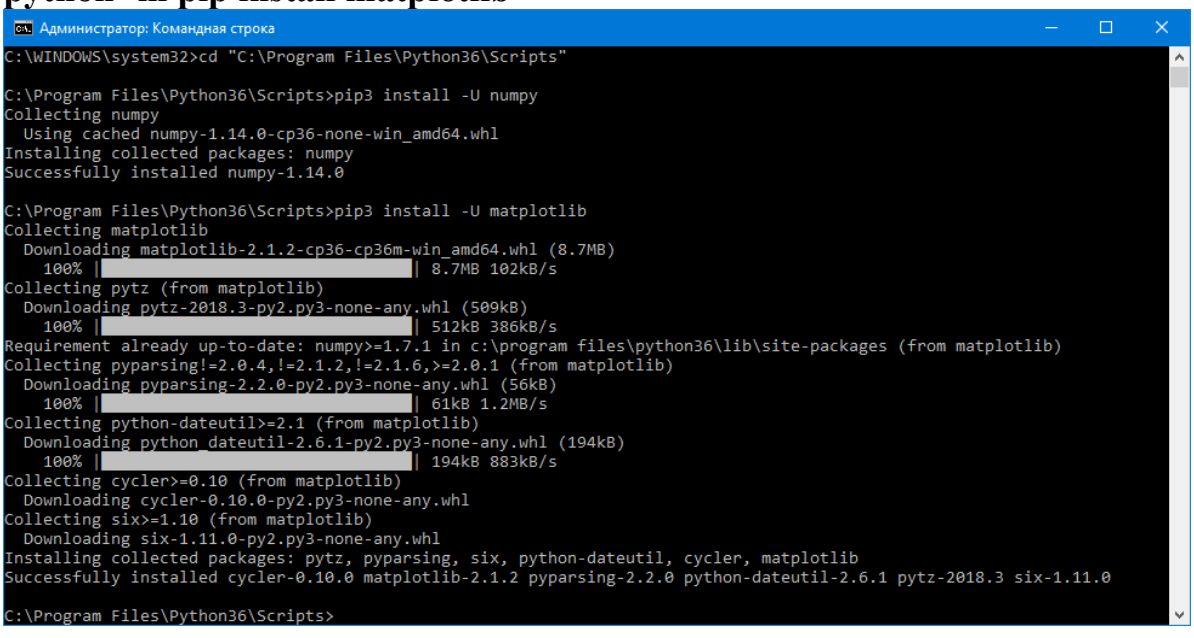
Сетевой ресурс https://github.com/whitehorn/Scientific_graphics_in_python содержит общедоступный учебник. Много информации по этой теме размещено на http://jenyay.net/Programming/Python3d и https://pyprog.pro/.

Установка

В Windows установка и matplotlib и NumPy не должна вызвать ни каких проблем. Используем **pip3** - систему управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python.

Если Python, например, установлен в C:\Program Files\Python36\, то в командной строке администратора (это важно) необхожимо выполнить следующие команды:

cd ":\Program Files\Python36\Scripts"
pip3 install **numpy**
pip3 install **matplotlib**
или так
python -m pip install numpy
python -m pip install matplotlib



Протокол работы:

```
C:\WINDOWS\system32>cd "C:\Program Files\Python36\Scripts"

C:\Program Files\Python36\Scripts>pip3 install -U numpy
Collecting numpy
  Using cached numpy-1.14.0-cp36-none-win_amd64.whl
Installing collected packages: numpy
Successfully installed numpy-1.14.0

C:\Program Files\Python36\Scripts>pip3 install -U matplotlib
Collecting matplotlib
  Downloading matplotlib-2.1.2-cp36-cp36m-win_amd64.whl (8.7MB)
    100% |████████████████████████████████| 8.7MB 102kB/s
Collecting pytz (from matplotlib)
  Downloading pytz-2018.3-py2.py3-none-any.whl (509kB)
    100% |████████████████████████████████| 512kB 386kB/s
Requirement already up-to-date: numpy>=1.7.1 in c:\program
files\python36\lib\site-packages (from matplotlib)
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib)
  Downloading pyparsing-2.2.0-py2.py3-none-any.whl (56kB)
    100% |████████████████████████████████| 61kB 1.2MB/s
Collecting python-dateutil>=2.1 (from matplotlib)
  Downloading python_dateutil-2.6.1-py2.py3-none-any.whl (194kB)
    100% |████████████████████████████████| 194kB 883kB/s
Collecting cycler>=0.10 (from matplotlib)
  Downloading cycler-0.10.0-py2.py3-none-any.whl
Collecting six>=1.10 (from matplotlib)
  Downloading six-1.11.0-py2.py3-none-any.whl
Installing collected packages: pytz, pyparsing, six, python-dateutil,
cycler, matplotlib
Successfully installed cycler-0.10.0 matplotlib-2.1.2 pyparsing-2.2.0
python-dateutil-2.6.1 pytz-2018.3 six-1.11.0

C:\Program Files\Python36\Scripts>
```

Примечание –в более старших версиях MatPlotLib возможны проблемы при инсталяции. В этом случае следует установливать Python и необходимые модули для одного пользователя (указывается в диалоге установки Python).

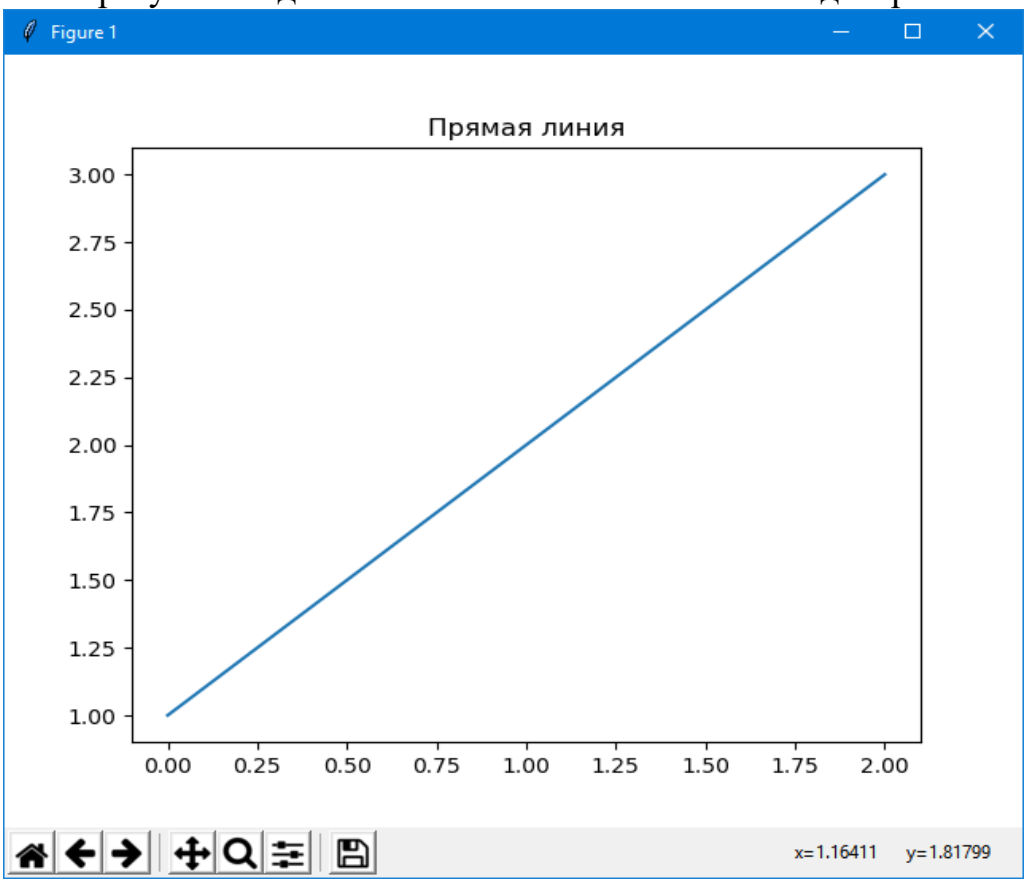
Проверка

После установки проверяем работоспособность. Запускаем консоль Python, вводим:

```
>>> import matplotlib as mpl
>>> print ('Current version on matplotlib library is',
mpl.__version__)
Current version on matplotlib library is 2.1.2

# де-факто стандарт вызова pyplot в python
import matplotlib.pyplot as plt
plt.plot([1,2,3])
[<matplotlib.lines.Line2D object at 0x0301E810>]
plt.title('Прямая линия')
```

```
<matplotlib.text.Text object at 0x03062430>
plt.show()
```

В результате должно появиться вот такое окно диаграммы:



Что было сделано?

Из пакета matplotlib импортирован модуль pyplot под именем plt.

Модуль pyplot содержит функции (похожие на команды) создания диаграмм и изменения свойств их элементов.

Функция **`plot`**() строит прямоугольные двумерные диаграммы (графики) в координатах X - Y.

Если функции **`plot`** передан один аргумент (в нашем случае - список [1,2,3]), она рассматривает его как совокупность значений откладываемых по оси Y, тогда по оси X ему будет соответствовать автоматически сгенерированный набор чисел 0,1,2...N - 1, где N - число элементов в переданном списке.

Функция **`title`**() задает заголовок диаграммы, а функция show() выводит интерактивное окно диаграммы. В общем все очень просто

Настройка

Пакет matplotlib можно легко настроить через конфигурационный файл matplotlibrc. Если Python установлен в папку C:\Program Files\Python36\, то настроечный файл располагается в C:\Program Files\Python36\Lib\site-packages\matplotlib\mpl-data\.

Вносить изменения можно прямо здесь, но лучше скопировать файл в папку пользователя: C:\Documents and Settings\UserName\.matplotlib\, иначе при переустановке пакета конфигурационный файл будет перезаписан. Параметры пакета задаются в файле в виде пар `свойство : значение`, символ # отделяет комментарий.

Свойство **`font.family`** определяет тип активного шрифта по умолчанию, может принимать пять значений: **`serif`**, **`sans-serif`**, **`cursive`**, **`fantasy`**, **`monospace`**.

В свою очередь свойства **`font.serif`**, **`font.sans-serif`**, **`font.cursive`**, **`font.fantasy`**, **`font.monospace`** содержат списки имен шрифтов (через запятую, в порядке уменьшения приоритета) соответствующих каждому типу.

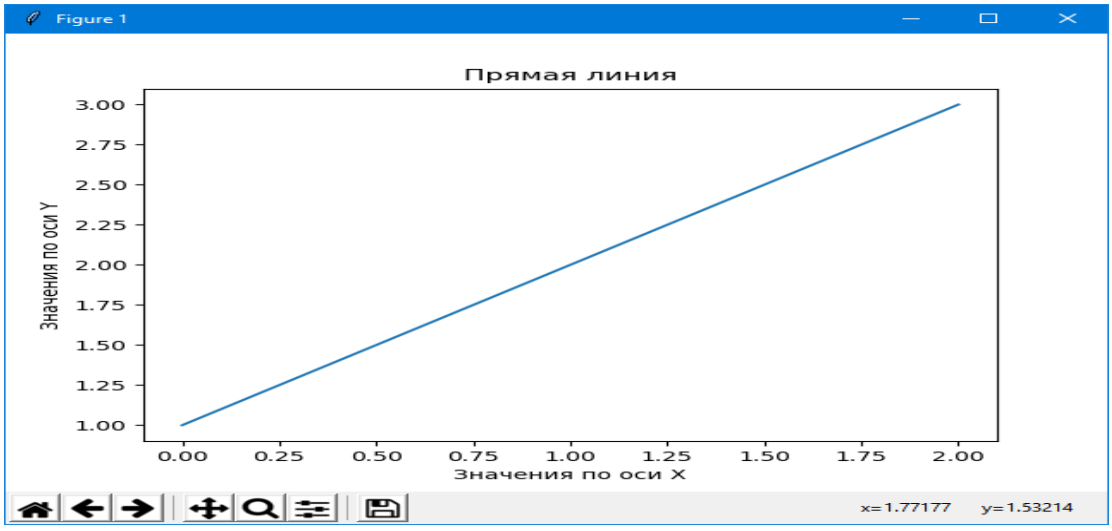
Например, для "русификации" matplotlib возможно придётся изменить имена шрифтов в списке, на имена доступных в системе шрифтов, содержащих кириллические символы. Например так:

```
font.serif : Verdana, Arial
font.sans-serif : Tahoma, Arial
font.cursive : Courier New, Arial
font.fantasy : Comic Sans MS, Arial
font.monospace : Arial
```

Заметим, что последние версии пакета инсталлируются с корректным настроечным файлом.

Рассмотрим пример:

```
>>> plt.plot([1,2,3])
[<matplotlib.lines.Line2D object at 0x000001BB6DFCA588>]
>>> plt.title('Прямая линия')
Text(0.5,1,'Прямая линия')
>>> plt.xlabel(u'Значения по оси X')
Text(0.5,0,'Значения по оси X')
>>> plt.ylabel(u'Значения по оси Y')
Text(0,0.5,'Значения по оси Y')
>>> plt.show()
```



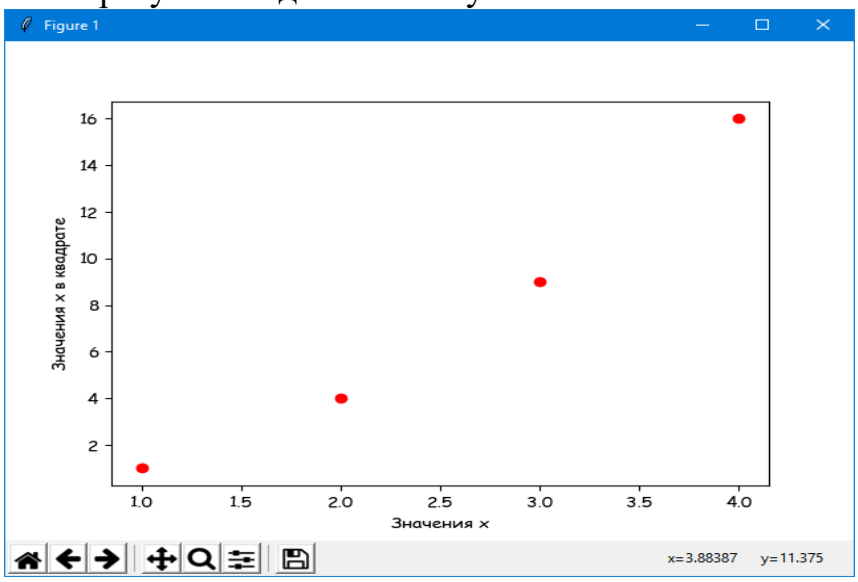
Функции `pyplot.xlabel()`, `pyplot.ylabel()`, как легко догадаться по названию, устанавливают подписи к осям X и Y соответственно.

Пакет matplotlib можно настраивать и динамически (во время выполнения программы). В пакете определена структура `matplotlib.rcParams`, с которой работают как со словарем. Ключевые слова в данном случае - имена свойств файла `matplotlibrc`.

Настройки шрифтов через `matplotlib.rcParams` можно изменить, например, так:

```
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
>>> mpl.rcParams['font.family'] = 'fantasy'
>>> mpl.rcParams['font.fantasy'] = 'Comic Sans MS, Arial'
>>> plt.plot([1,2,3,4], [1,4,9,16], 'ro')
[<matplotlib.lines.Line2D object at 0x000001BB6E03B470>]
>>> plt.xlabel('Значения x')
Text(0.5,0,'Значения x')
>>> plt.ylabel('Значения x в квадрате')
Text(0,0.5,'Значения x в квадрате')
>>> plt.show()
```

В результате должно получиться вот что:



Функции `pyplot.plot()` можно передать произвольное число пар аргументов типов `list` или **`array`** (тип **`array`** определен в модуле **`numpy`**).

В примере выше передана одна пара: [1,2,3,4], [1,4,9,16]. Первый элемент каждой пары функция рассматривает как значения, откладываемые по оси X, второй элемент как значения по оси Y. Соответственно для каждой пары элементов строиться своя кривая на диаграмме.

После каждой пары данных может быть передан дополнительный аргумент типа `string` - строка форматирования, которая определяет внешний вид кривой.

Формат строки позаимствован из системы Matlab. Строка включает три подстроки.

- первая подстрока задает цвет графического элемента,
- вторая стиль маркера,
- третья стиль линии.

В примере 'ro' = 'r' + 'o', где 'r' - красный, 'o' - кружок. По умолчанию 'b-' - синяя непрерывная линия.

Архитектура matplotlib

Одна из основных задач, которую выполняет matplotlib — предоставление набора функций и инструментов для представления и управления Figure (так называется основной объект) вместе со всеми внутренними объектами, из которого он состоит. Но в matplotlib есть также инструменты для обработки событий и, например, анимации. Благодаря им эта библиотека способна создавать интерактивные графики на основе событий по нажатию кнопки или движению мыши.

Архитектура matplotlib логически разделена на три слоя, расположенных на трех уровнях. Коммуникация непрямая — каждый слой может взаимодействовать только с тем, что расположен под ним, но не над.

Вот эти слои:

- Слой сценария
- Художественный слой
- Слой бэкенда

Слой бэкенда

Слой `Backend` является нижним на диаграмме с архитектурой всей библиотеки. Он содержит все API и набор классов, отвечающих за реализацию графических элементов на низком уровне.

- `FigureCanvas` — это объект, олицетворяющий область рисования.
- `Renderer` — объект, который рисует по `FigureCanvas`.
- `Event` — объект, обрабатывающий ввод от пользователя (события с клавиатуры и мыши)

Художественный слой

Средним слоем выступает художественный (`artist`). Все элементы, составляющие график, такие как название, метки осей, маркеры и так далее, являются экземплярами этого объекта. Каждый из них играет свою роль в иерархической структуре.

Есть два художественных класса: примитивный и составной.

- Примитивный — это объекты, которые представляют собой базовые элементы для формирования графического представления графика, например, Line2D, или геометрические фигуры, такие как прямоугольник круг или даже текст.

- Составные — объекты, состоящие из нескольких базовых (примитивных). Это оси, шкалы и диаграммы.

На этом уровне часто приходится иметь дело с объектами, занимающими высокое положение в иерархии: график, система координат, оси. Поэтому важно полностью понимать, какую роль они играют. Ниже перечислены три основных художественных (составных объекта), которые часто используются на этом уровне.

- `Figure` — объект, занимающий верхнюю позицию в иерархии. Он соответствует всему графическому представлению и может содержать много систем координат.
- `Axes` — это тот самый график. Каждая система координат принадлежит только одному объекту `Figure` и имеет два объекта `Axis` (или три, если речь идет о трехмерном графике). Другие объекты, такие как название, метки `x` и `y`, принадлежат отдельно осям.
- `Axis` учитывает числовые значения в системе координат, определяет пределы и управляет обозначениями на осях, а также соответствующим каждому из них текстом. Положение шкал определяется объектом `Locator`, а внешний вид — `Formatter`.

Слой сценария (pyplot)

Художественные классы и связанные с ними функции (`API matplotlib`) подходят всем разработчикам, особенно тем, кто работает с серверами веб-приложений или разрабатывает графические интерфейсы. Но для вычислений, в частности для анализа и визуализации данных, лучше всего подходит слой сценария. Он включает интерфейс `pyplot`.

pylab и pyplot

С точки зрения пользователя существуют две библиотеки: `pylab` и `pyplot`. `Pylab` — это модуль, устанавливаемый вместе с `matplotlib`, а `pyplot` — внутренний модуль `matplotlib`. На оба часто ссылаются в скриптах:

```
from pylab import *

# и

import matplotlib.pyplot as plt
import numpy as np
```

`Pylab` объединяет функциональность `pyplot` с возможностями `NumPy` в одном пространстве имен, поэтому отдельно импортировать `NumPy` не нужно. Более того, при импорте `pylab` функции из `pyplot` и `NumPy` можно вызывать без ссылки на модуль (пространство имен).

```
plot(x,y)
array([1,2,3,4])
# вместо
plt.plot()
np.array([1,2,3,4]
```

Пакет `pyplot` предлагает классический интерфейс Python для программирования, имеет собственное пространство имеет и требует отдельного импорта NumPy. В последующих материалах используется этот подход. Его же применяет большая часть программистов на Python.

Назначении кнопок интерактивного окна диаграммы.

Кнопка 

(pan/zoom) предназначена для прокрутки/масштабирования диаграммы.

Нажимаем на кнопку, указатель мыши приобретает вид перекрещенных стрелок.

Если двигать мышь и удерживать левую кнопку - диаграмма будет прокручиваться по направлению движения указателя.

Если двигать мышь и удерживать правую кнопку - масштаб диаграммы будет изменяться, при движении вверх/вправо - уменьшаться, вниз/влево - увеличиваться.

При нажатых клавишах "**x**" и "**y**" перемещение/масштабирование диаграммы будет происходить по осям X и Y соответственно.

При нажатой клавише **Ctrl** диаграмма будет изменяться так, чтобы сохранились её пропорции (aspect ratio).

Кнопка 

при нажатии включает режим прямоугольного масштабирования.

Если нажать на кнопку указатель мыши приобретет вид креста.

При нажатой левой кнопки мыши на диаграмме можно выделить прямоугольную область. После освобождения кнопки масштаб диаграммы будет изменен так, чтобы выделенная область заняла как можно большую площадь диаграммы.

Кнопки 

позволяют перемещаться по истории изменения диаграммы. Все изменения вносимые в диаграмму пользователем (масштабирование, прокрутка) запоминаются.

Кнопки со стрелками позволяют перемещаться вперед/назад по имеющимся вариантам диаграммы.

Кнопка с домиком возвращает диаграмму к исходному состоянию.

Кнопка 
вызывает стандартный для системы диалог сохранения файла.

Позволяет сохранить диаграмму в файл. Можно выбрать следующие форматы файла диаграммы: **png**, **pdf**, **svg**, **eps**, **ps**.

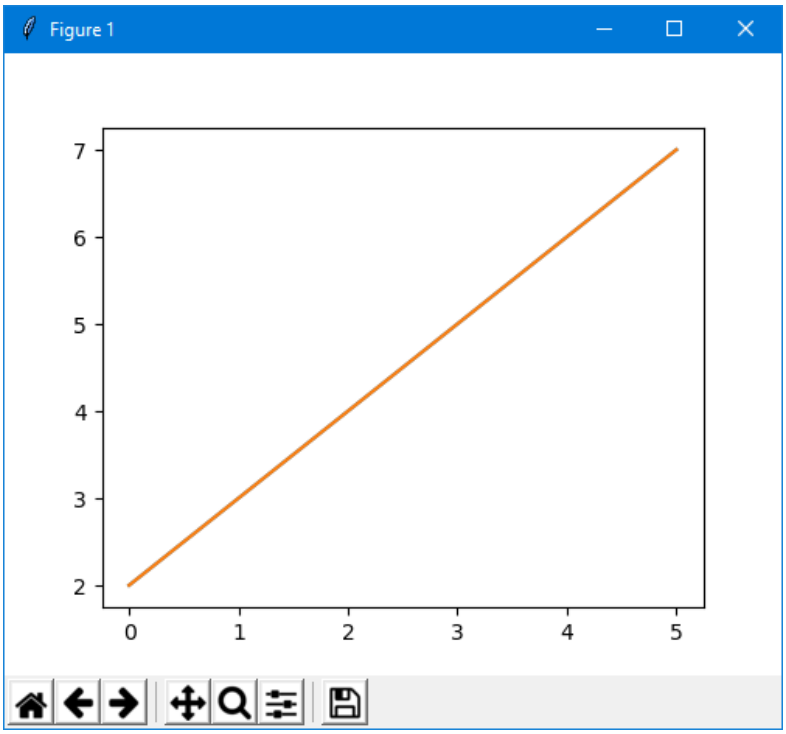
Во всех примерах импортируем модули:

```
import matplotlib as mpl
# де-факто стандарт вызова pyplot в python
import matplotlib.pyplot as plt
```

П1. Список y координат, x координаты числа 0, 1, 2, 3...

Список *y*[2,3,4,5,6,7] координат, *x* координаты образуют последовательность 0, 1, 2, ...:

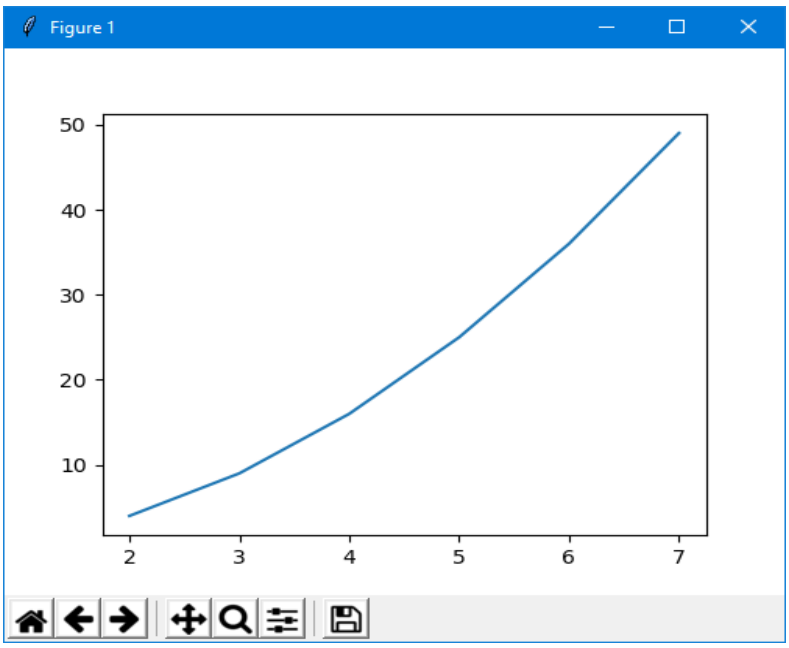
```
plt.plot([2,3,4,5,6,7]);plt.show()
```



П2. Списоки x и y-координат

Список x и y координат - [2,3,4,5,6,7], [4,9,16,25,36,49]);

```
plt.plot([2,3,4,5,6,7],[4,9,16,25,36,49]);plt.show()
```

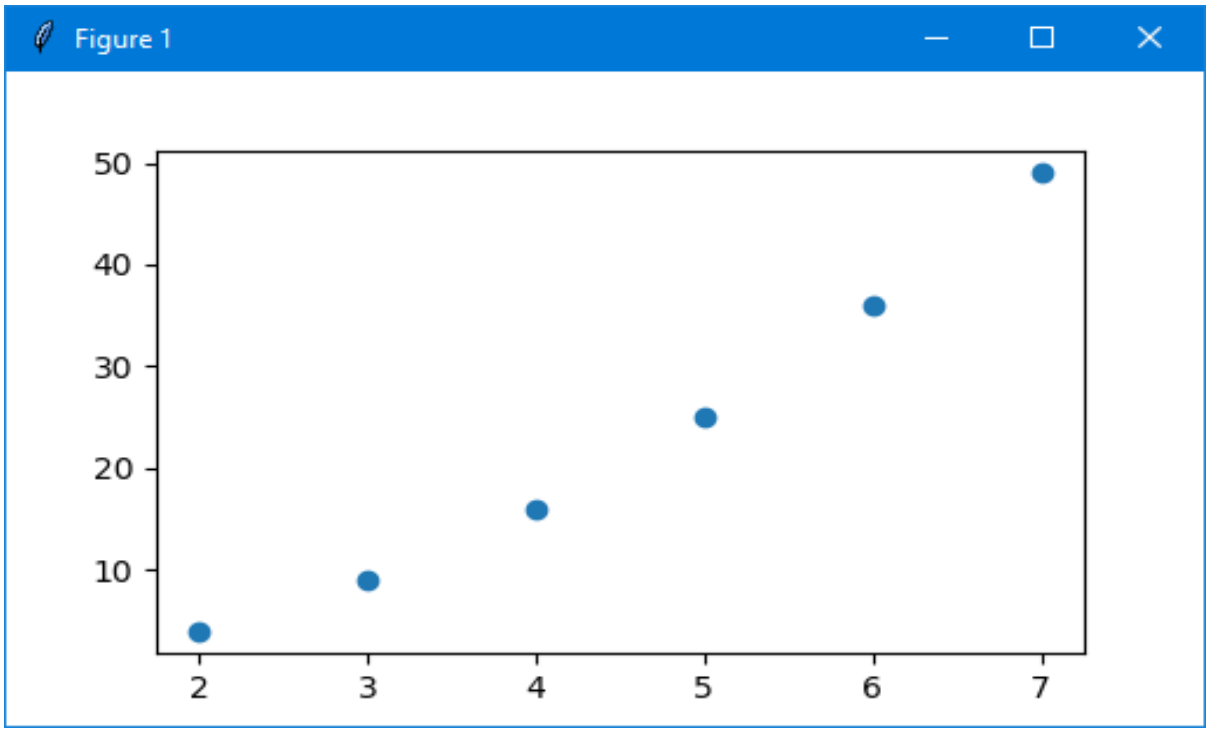


П3. Изображаем точки

Список x и y координат - [2,3,4,5,6,7], [4,9,16,25,36,49]);

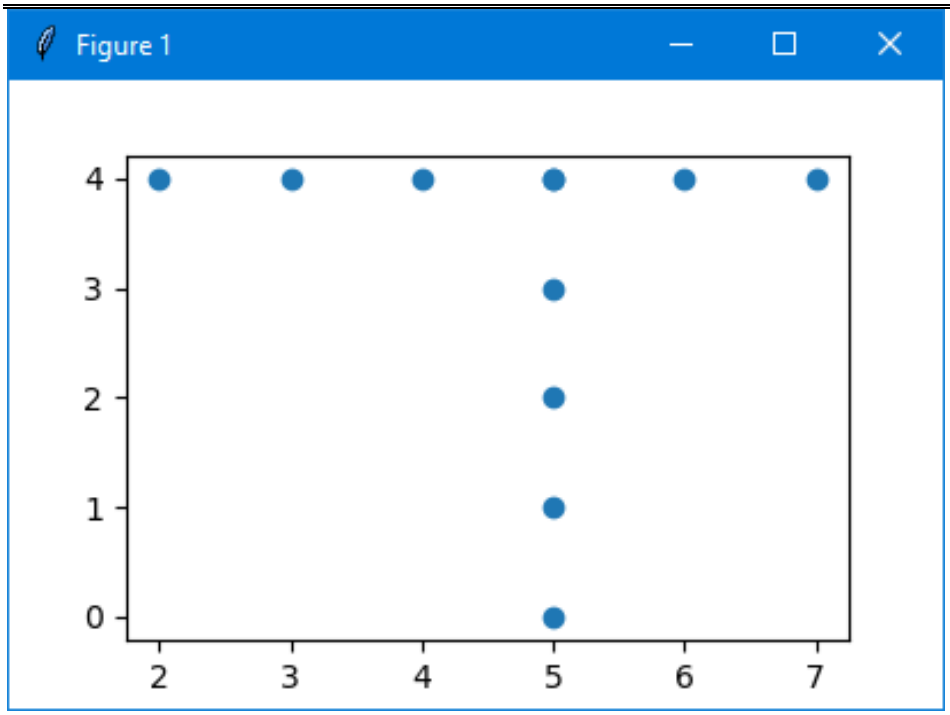
Изображаем только точки

```
plt.scatter([2,3,4,5,6,7],[4,9,16,25,36,49]);plt.show()
```



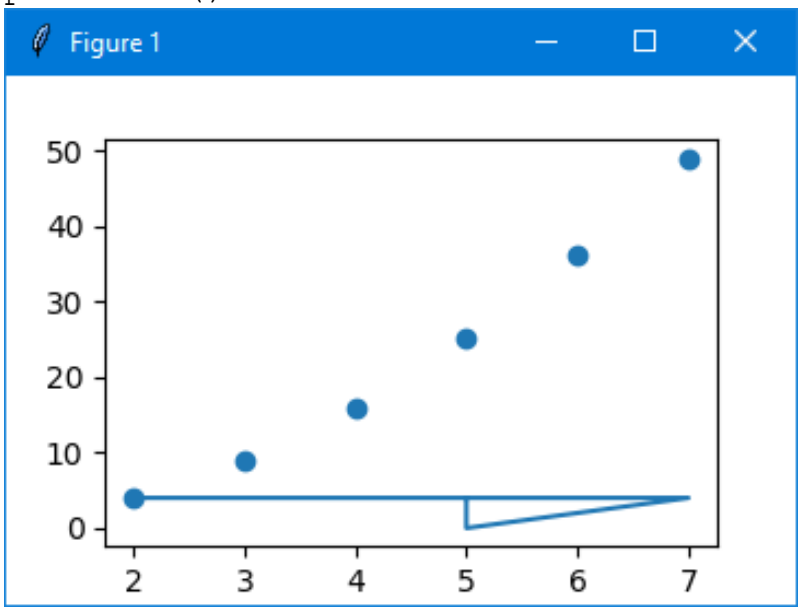
П4. «Произвольные» координаты

plt.scatter([2,3,4,5,6,7,5,5,5,5,5],[4,4,4,4,4,4,0,1,2,3,4,]);plt.show()



П5. Несколько графиков на одном листе

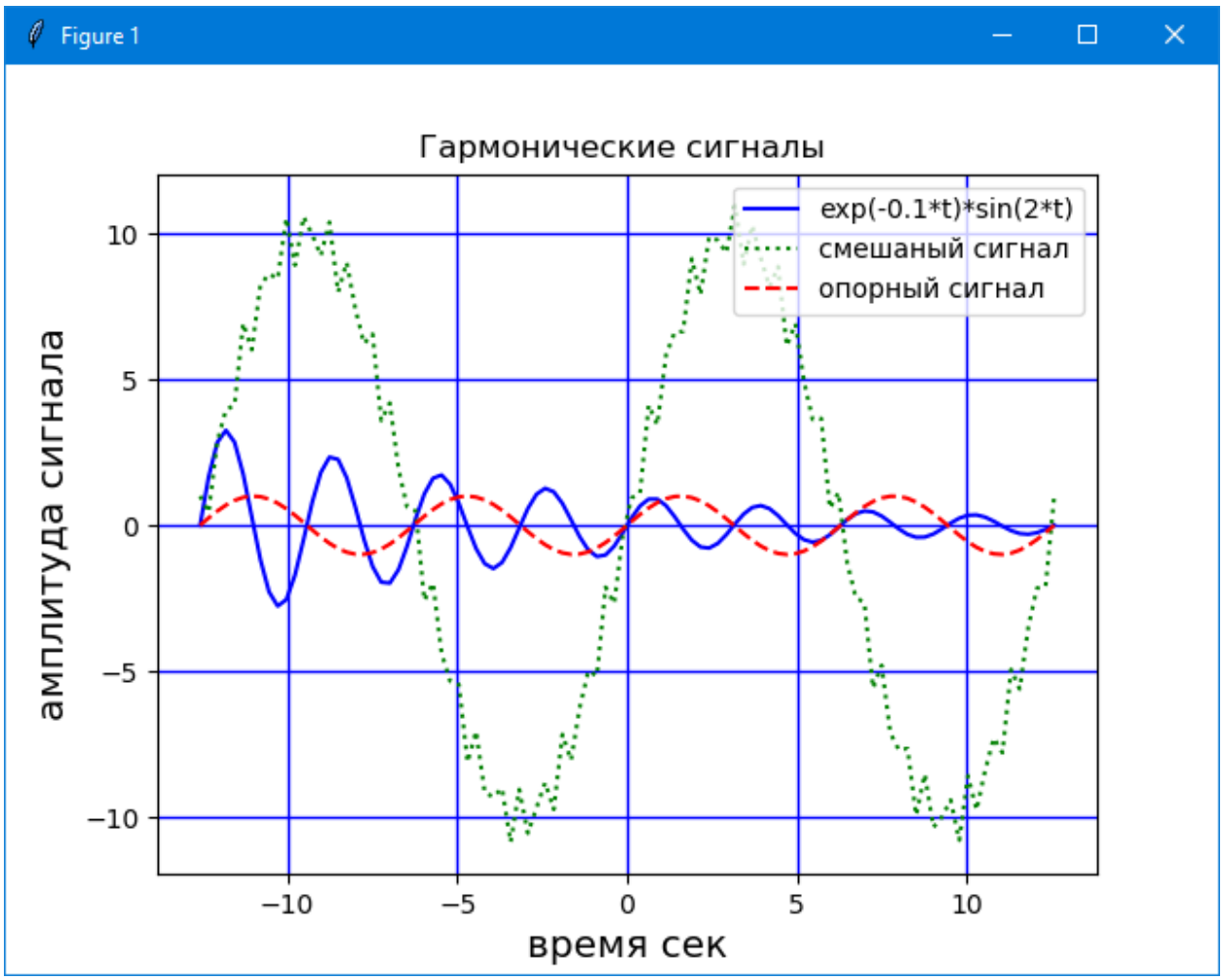
```
plt.scatter([2,3,4,5,6,7],[4,9,16,25,36,49])
plt.plot([2,3,4,5,6,7,5,5,5,5,5],
                                [4,4,4,4,4,4,0,1,2,3,4,])
plt.show()
```



П5. «Украшательство» и много графиков на диаграмме

Рисуем и сохраняем в файле три графика на одной диаграмме в одном масшабе. Смотрите комментарии скрипта.

```
import matplotlib.pyplot as plt
import numpy as np
# независимая переменная
x = np.linspace(-4*np.pi, 4*np.pi, 100)
# название осей
xlab='время сек'
ylab='амплитуда сигнала'
# первый график
y1= np.sin(2*x)*np.exp(-0.1*x)
# легенда
leg1='exp(-0.1*t)*sin(2*t)'
# второй график
y2=10*np.sin(0.5*x)+np.cos(10*x)
leg2='смешаны сигнал'
# третий графие
y3=np.sin(x)
leg3='опорный сигнал'
# Включаем сетку по оси X и оси Y.
# Задаем цвет толщину сетки
plt.grid(color = 'b',linewidth = 1)
# Задаем подписи к осям X и оси Y и размер шрифта
plt.xlabel(xlab, fontsize = 'x-large')
plt.ylabel(ylab, fontsize = 'x-large')
# Задаем заголовок диаграммы
plt.title('Гармонические сигналы ')
# строим графики
plt.plot(x,y1,'b-',label=leg1)
plt.plot(x,y2,'g:',label=leg2)
plt.plot(x,y3,'r--',label=leg3)

# задаем вывод легенды и ее расположение
plt.legend(loc='best')
# Включаем сетку
plt.grid(True)
# Сохраняем построенную диаграмму в файл
# Задаем имя файла и его тип
plt.savefig('signal3.png', format = 'png')
# визуализируем графики
plt.show()
```



Как задаются стили линий, маркеры и цвета линий смотрите в документации или воспользуйтесть «демо скриптом»:

```
import sys
import math
import os
import matplotlib.pyplot as plt
import numpy as np
##sys.exit(0)
import matplotlib as mpl
# Вывод на экран текущей версии библиотеки matplotlib
print ('Current version on matplotlib library is',
                                        mpl.__version__)

x = np.linspace(-2, 2, 10)
y1=1*x
y2=2*x
y3=4*x
y4=8*x

s1= ['-',   'solid line ',  'непрерывная линия']
s2= ['--',  'dashed line ',  'линия из штрихов']
```

```
s3= ['-.',  'dash-dot line ',  'чередование штрихов и
точек']
s4= [':',   'dotted line ',  'линия из точек']

leg1=s1[0]+' '+s1[1]+' '+s1[2]
leg2=s2[0]+' '+s2[1]+' '+s2[2]
leg3=s3[0]+' '+s3[1]+' '+s3[2]
leg4=s4[0]+' '+s4[1]+' '+s4[2]

ax=plt.subplot(111)
#можно поменть размеры окна графика в граф. окне
box=ax.get_position()
ax.set_position([box.x0, box.y0,
                    box.width*1.0 , box.height*1.0])

p1=plt.plot(x,y1,linestyle=s1[0], label=leg1)
p2=plt.plot(x,y2,linestyle=s2[0], label=leg2)
p3=plt.plot(x,y3,linestyle=s3[0], label=leg3)
p4=plt.plot(x,y4,linestyle=s4[0], label=leg4)

plt.title('Стили линий (linestyle=)')
#ax.legend(loc=(1.0,0.5), mode='expand' )
#ax.legend(mode='expand', bbox_to_anchor=(1, 0.05),loc
#                    ='upper center' ) #left best
ax.legend(loc='lower right')
# Включаем сетку
plt.grid()
plt.show()

#sys.exit(0)
# marker
plt.close()
mar=[
'.', 'point marker',
',', 'pixel marker',
'o', 'circle marker',
'v', 'triangle_down marker',
'^', 'triangle_up marker',
'<', 'triangle_left marker',
'>', 'triangle_right marker',
'1', 'tri_down marker',
'2', 'tri_up marker',
'3', 'tri_left marker',
'4', 'tri_right marker',
's', 'square marker',
```

```
'p', 'pentagon marker',
'*', 'star marker',
'h', 'hexagon1 marker',
'H', 'hexagon2 marker',
'+', 'plus marker',
'x', 'x marker',
'D', 'diamond marker',
'd', 'thin_diamond marker',
'|', 'vline marker',
'_', 'hline marker'
]
leg=[]
for i in range(0,len(mar),2):
 leg.append(mar[i]+'  '+mar[i+1])
mpl.rcParams['figure.figsize'] = (8.0, 6.0)
kk=int( len(mar)/2 )
for i in range(kk):
    plt.plot(x, x+i, marker=leg[i][0], label=leg[i])
plt.legend(loc='lower right')
plt.title('Маркеры (marker=)')
plt.show()

#color
col=[
'b', 'blue    синий',
'g', 'green   зеленый',
'r', 'red     красный',
'c', 'cyan    бирюзовый',
'm', 'magenta пурпурный',
'y', 'yellow  желтый',
'k', 'black   черный',
'w', 'white   белый'
]

leg=[]
for i in range(0,len(col),2):
 leg.append(col[i]+'  '+col[i+1])
mpl.rcParams['figure.figsize'] = (5.0, 4.0)
kk=int( len(col)/2 )
for i in range(kk):
    plt.plot(x, x+i, color=leg[i][0], label=leg[i])
plt.legend(loc='lower right')
plt.title('Цвета (color=)')
plt.show()
```

Демо графики:

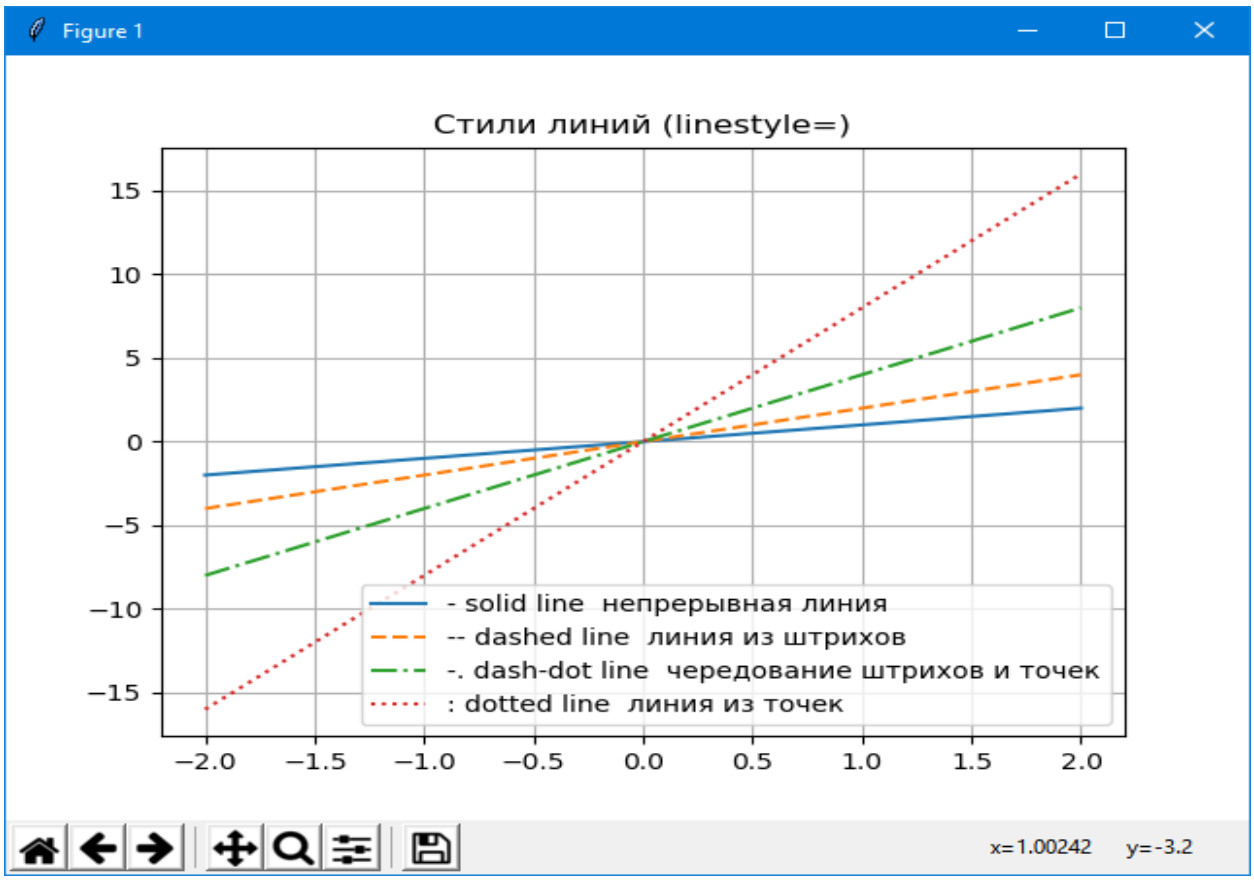


Figure 1
Стили линий (linestyle=)
- solid line непрерывная линия
-- dashed line линия из штрихов
-. dash-dot line чередование штрихов и точек
: dotted line линия из точек
x=1.00242 y=-3.2

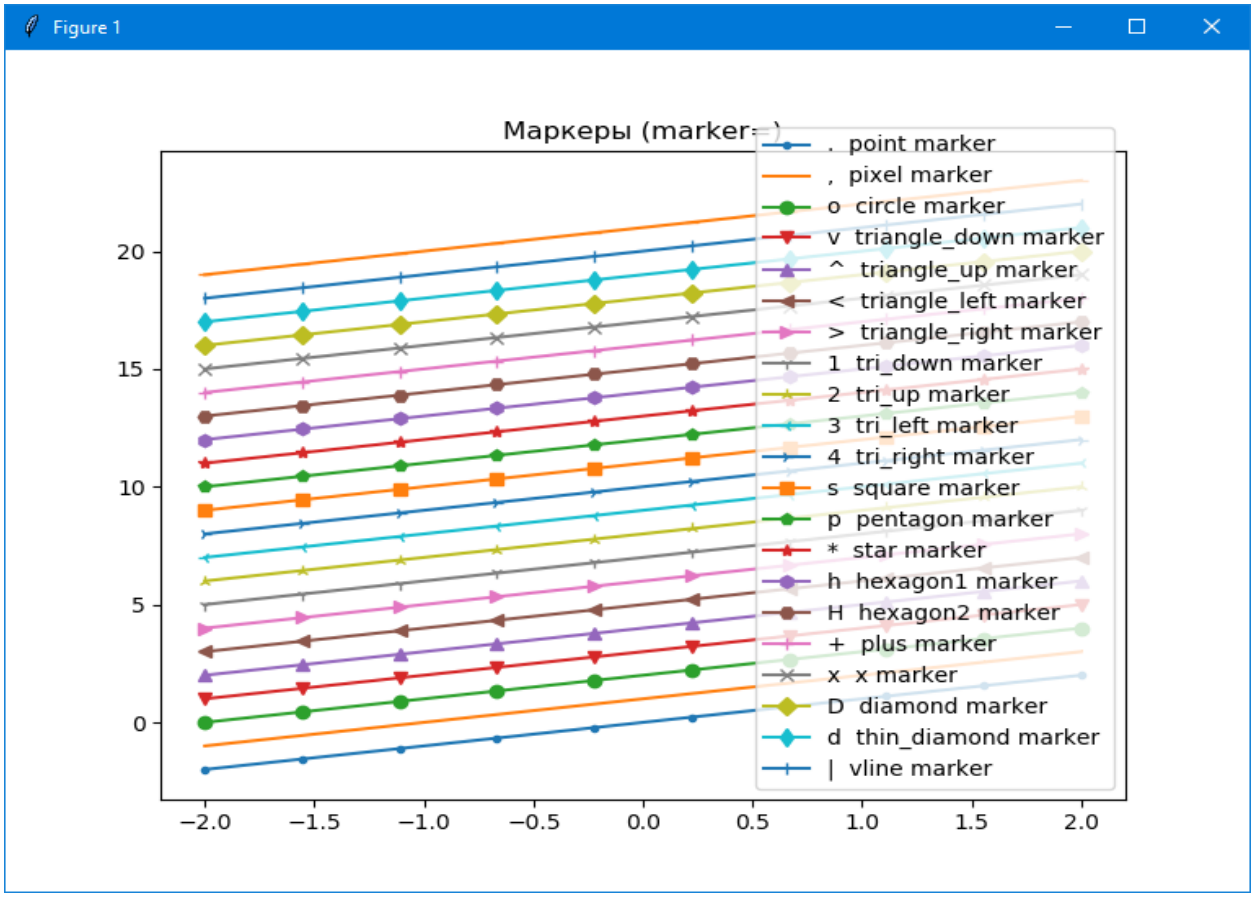
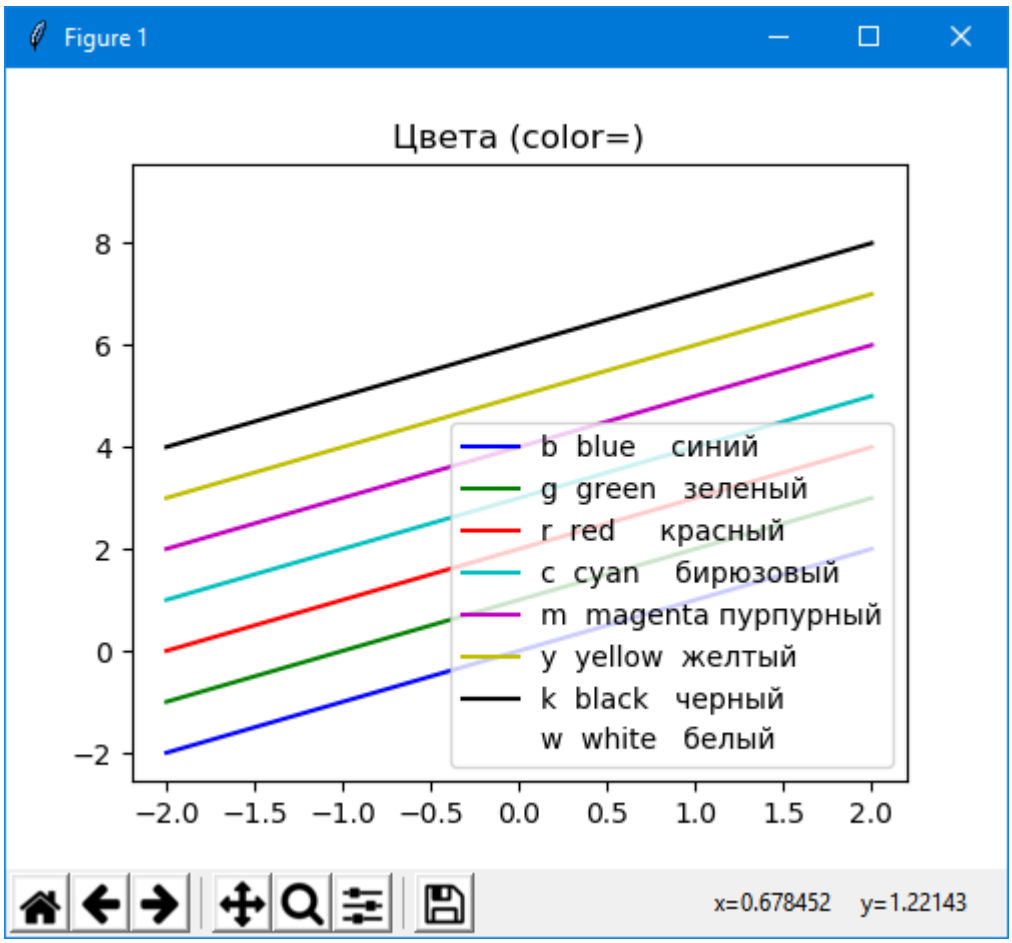


Figure 1
Маркеры (marker=)
. point marker
, pixel marker
o circle marker
v triangle_down marker
^ triangle_up marker
< triangle_left marker
> triangle_right marker
1 tri_down marker
2 tri_up marker
3 tri_left marker
4 tri_right marker
s square marker
p pentagon marker
* star marker
h hexagon1 marker
H hexagon2 marker
+ plus marker
x x marker
D diamond marker
d thin_diamond marker
| vline marker



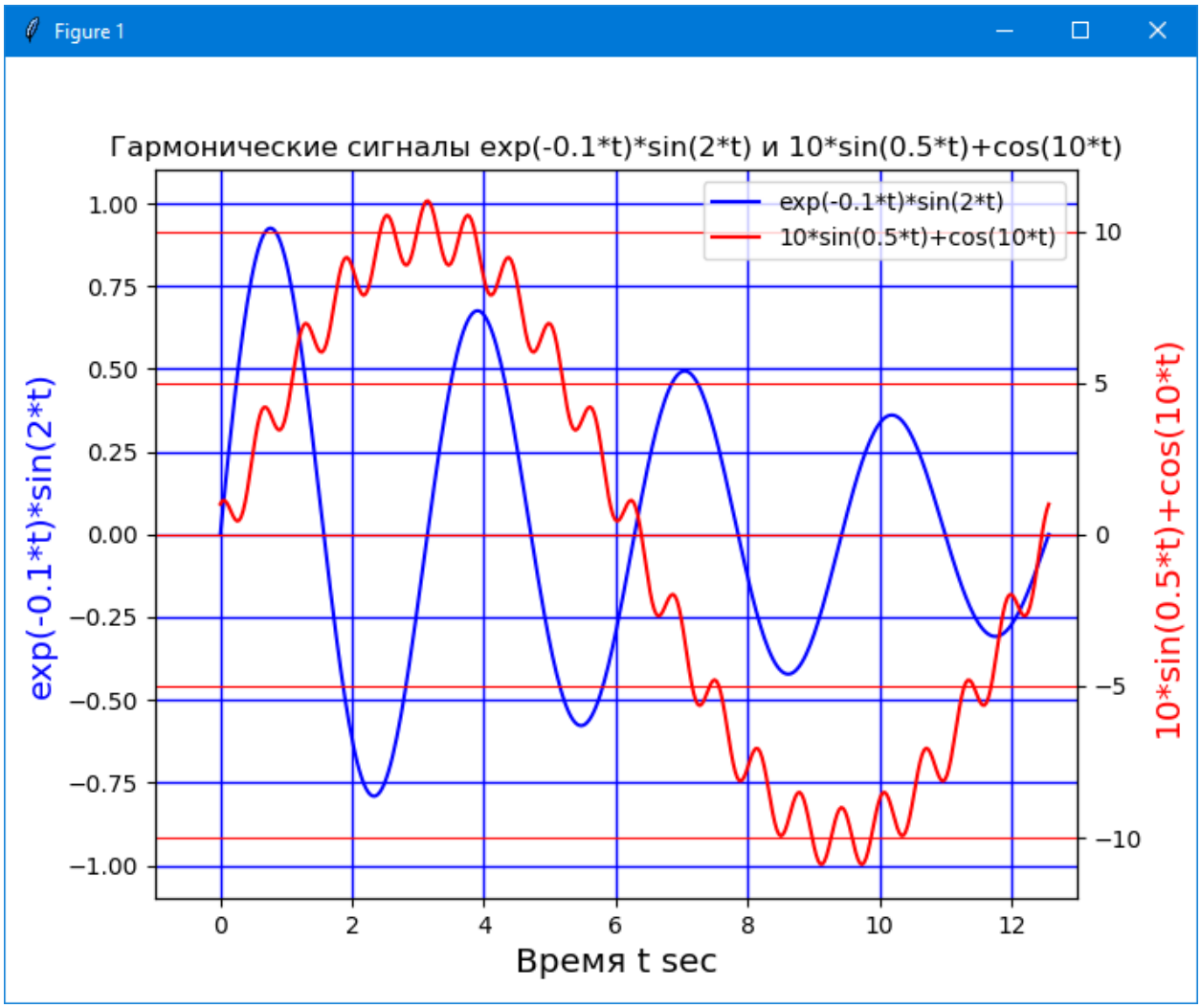
П6. Два графика на диаграмме в индивидуальных масштабах

Обратите внимание на «запятую» после имени переменной **line_01,** = ax_01.plot(X, Y_01, 'b-')

```
# -*- coding: UTF-8 -*-
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
# Значения по оси X время, t
X = np.linspace(0, 4*np.pi, 400)
# Значения по осям Y_01 & Y_02
leg_01='exp(-0.1*t)*sin(2*t)'
Y_01 = np.sin(2*X)*np.exp(-0.1*X)
# "модулированный" сигнал
leg_02="10*sin(0.5*t)+cos(10*t)"
Y_02=10*np.sin(0.5*X)+np.cos(10*X)
# Зададим размеры изображения диаграммы
mpl.rcParams['figure.figsize'] = (8.0, 6.0)
# Строим диаграмму
```

```
# Получаем ссылку на объект типа
matplotlib.axes.AxesSubplot, текущую диаграмму
ax_01 = plt.axes()
#  Задаем исходные данные для первой линии диаграммы
#  (линии степени разложения), внешний вид линии и
#   маркера.
# Функция plot() возвращает ссылку на list,
# первый элемент, которого есть объект класса
#  matplotlib.lines.Line2D
line_01, = ax_01.plot(X, Y_01, 'b-') #, label=leg_01)
# если надо в дальнейшем использовать объект класса
#  matplotlib.lines.Line2D
# то применяют ДВА варианта
# 1) line_01, = ax_01.plot(X, Y_01, 'b-')
#     прием первого элемента списка
# 2) line_01 (без запятой)= … использовать line_01[0]
# Задаем интервалы значений по осям X и
#  основной оси Y
ax_01.axis([-1, 13,   -1.1, 1.1])
# Включаем сетку по оси X и основной оси Y.
#  Задаем цвет сетки
ax_01.grid(color = 'b',linewidth = 1)
# Задаем подписи к осям X и основной оси Y
#  и размер фонта
ax_01.set_xlabel('Время t sec', fontsize = 'x-large')
ax_01.set_ylabel(leg_01, color = 'b',
                                  fontsize = 'x-large')
# Задаем заголовок диаграммы
ax_01.set_title('Гармонические сигналы '+
                                 leg_01+" и "+ leg_02)
#ax_01.legend( loc='upper right')
# Включаем дополнительную ось Y
ax_02 = ax_01.twinx()
# Задаем исходные данные для второй линии диаграммы,
#  внешний вид линии и маркера.
# Функция plot() возвращает ссылку на объект класса
#   matplotlib.lines.Line2D (см выше)
line_02, = ax_02.plot(X, Y_02, 'r-') #, label=leg_02)
# Задаем интервалы значений по осям X
#  и дополнительной оси Y
ax_02.axis([-1, 13, -12, 12])
# Задаем подпись к дополнительной оси Y
ax_02.set_ylabel(leg_02,color='r',
                               fontsize = 'x-large')
```

```
# Задаем исходные данные для легенды и
# место ее размещение
ax_02.legend((line_01, line_02), (leg_01, leg_02),
                                        loc = 'best')
#ax_02.legend(loc='lower left')
# Включаем сетку по дополнительной оси Y.
#  Задаем цвет сетки
ax_02.grid(color = 'r')
# Сохраняем построенную диаграмму в файл
# Задаем имя файла и его тип
#plt.savefig('trigo.png', format = 'png')
# визуализируем
plt.show()
```



П7 Несколько графиков в одном и в разных графических окнах

Работа с `matplotlib` основана на использовании графических окон и осей (оси позволяют задать некоторую графическую область). Все

построения применяются к текущим осям. Это позволяет изображать несколько графиков в одном графическом окне.

По умолчанию создается одно графическое окно **figure(1)** и одна графическая область **subplot(111)** в этом окне.

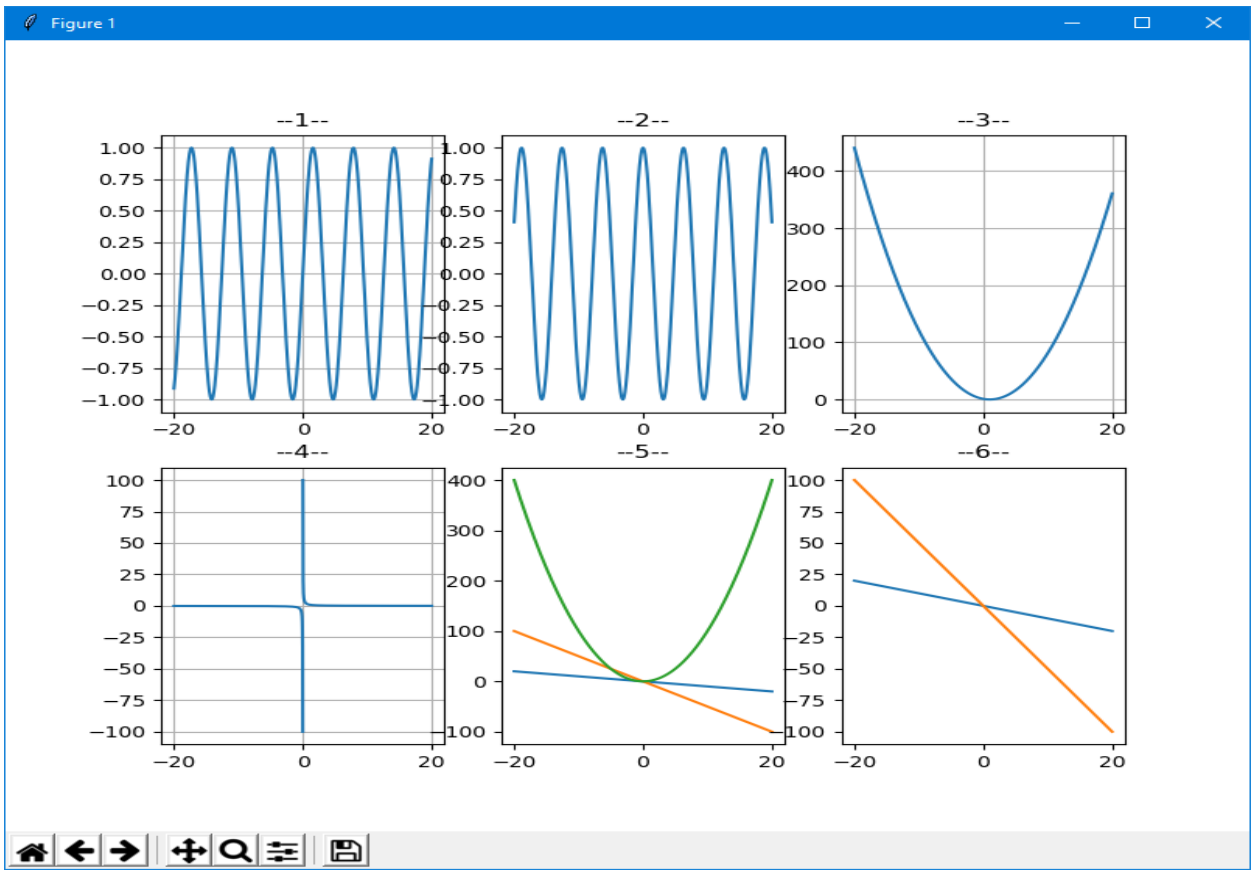
Команда **subplot** позволяет разбить графическое окно на несколько областей. Она имеет три параметра: **nr; nc; np**:

- параметры **nr** и **nc** определяют количество строк и столбцов на которые разбивается графическая область
- параметр **np** определяет номер текущей области (np принимает значения от 1 до nr*nc).

Если **nr*nc<10**, то передавать параметры **nr,nc,np** можно без использовния запятой. Например, допустимы формы **subplot(2,2,1)** и **subplot(221).**

```
import math
import numpy as np
import matplotlib.pyplot as plt

# Импортируем пакет со вспомогательными функциями
import matplotlib.mlab as ml

# Будем рисовать график этой функции
def func (x):
    """
    sinc (x)
    """
    if x == 0:
        return 1.0
    return math.sin (x) / x

# Интервал изменения переменной по оси X
xmin = -20.0
xmax = 20.0
# Шаг между точками
dx = 0.01
# Создадим список координат по оиси X на отрезке [-
xmin; xmax], включая концы
xlist = ml.frange (xmin, xmax, dx)
# Вычислим значение функции в заданных точках
ylist = [func (x) for x in xlist]

# !!! Две строки, три столбца.
# !!! Текущая ячейка - 1
plt.subplot (2, 3, 1)
plt.plot (xlist, np.sin(xlist) ) #ylist)
```

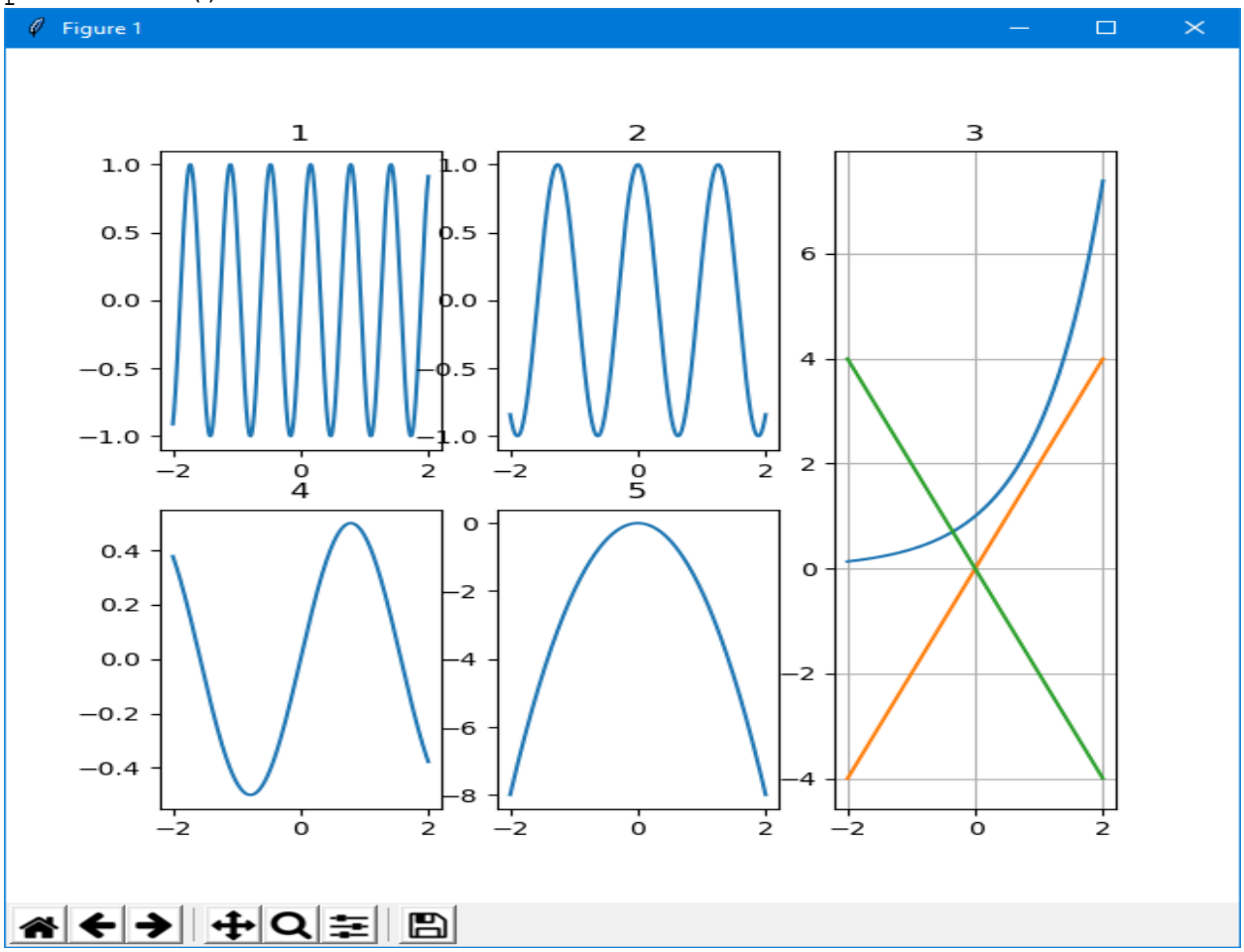
```
plt.grid(True)
plt.title ("--1--")

# !!! Две строки, три столбца.
# !!! Текущая ячейка - 2
plt.subplot (2, 3, 2)
plt.plot (xlist, np.cos(xlist) )
plt.title ("--2--")

# !!! Две строки, три столбца.
# !!! Текущая ячейка - 3
plt.subplot (2, 3, 3)
plt.plot (xlist, (xlist-1)*(xlist-1))
plt.grid(True)
plt.title ("--3--")

# !!! Две строки, три столбца.
# !!! Текущая ячейка - 4
plt.subplot (2, 3, 4)
plt.plot (xlist, 1/xlist)
plt.grid(True)
plt.title ("--4--")

# !!! Две строки, три столбца.
# !!! Текущая ячейка - 5
plt.subplot (2, 3, 5)
plt.plot (xlist, -xlist )
plt.plot (xlist, -5*xlist )
plt.plot (xlist, xlist*xlist)
plt.title ("--5--")

# !!! Две строки, три столбца.
# !!! Текущая ячейка - 6
plt.subplot (2, 3, 6)
plt.plot (xlist, -xlist )
plt.plot (xlist, -5*xlist )
plt.title ("--6--")

# Покажем окно с нарисованным графиком
plt.show()
```



Все эти ячейки чисто условные, поэтому для каждого вызова **subplot()** разбиение может быть свое, что позволяет сделать, например, следующее расположение графиков (обратите внимание на нумерацию ячеек):

```
import math
import numpy as np
import matplotlib.pyplot as plt
# Импортируем пакет со вспомогательными функциями
import matplotlib.mlab as ml
# Интервал изменения переменной по оси X
xmin = -2.0
xmax = 2.0
# Шаг между точками
dx = 0.001
# Создадим список координат по оиси X на отрезке [-
xmin; xmax], включая концы
x = ml.frange (xmin, xmax, dx)
# !!! Две строки, три столбца.
# !!! Текущая ячейка - 1
plt.subplot (2, 3, 1)
plt.plot (x, np.sin(10*x))
plt.title (" 1 ")
# !!! Две строки, три столбца.
# !!! Текущая ячейка - 2
```

```
plt.subplot (2, 3, 2)
plt.plot (x, np.cos(5*x) )
plt.title ("  2  ")
# !!! Две строки, три столбца.
# !!! Текущая ячейка - 4
plt.subplot (2, 3, 4)
plt.plot (x, np.sin(x)*np.cos(x))
plt.title ("  4  ")
# !!! Две строки, три столбца.
# !!! Текущая ячейка - 5
plt.subplot (2, 3, 5)
plt.plot (x, -2*x*x)
plt.title ("  5  ")
# !!! Одна строка, три столбца.
# !!! Текущая ячейка - 3
plt.subplot (1, 3, 3)
plt.plot (x, np.exp(x))
plt.plot (x, 2*x)
plt.plot (x, -2*x)
plt.grid(True)
plt.title ("  3  ")
# Покажем окно с нарисованным графиком
plt.show()
```



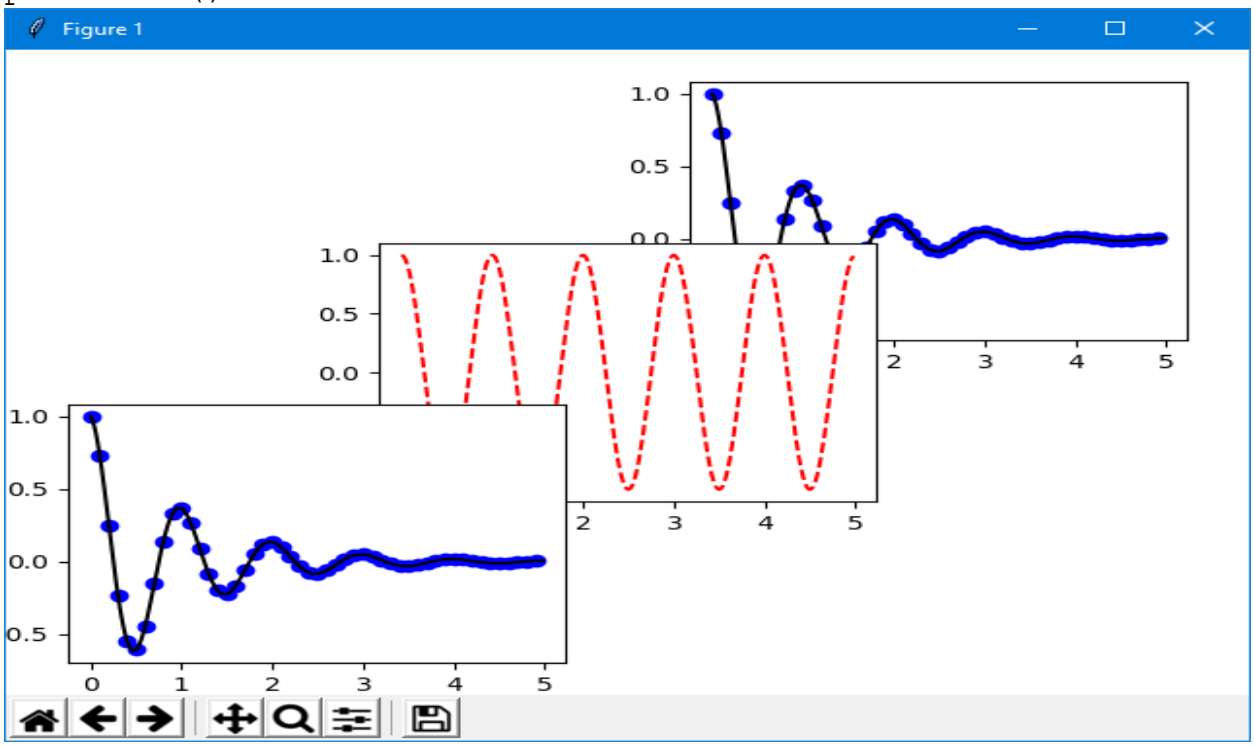
figure() и axes()

Есть возможность расположить оси (подокна) вручную, например не на прямоугольной сетке, а как-нибудь ёлочкой, используйте команду ***axes()***, которая позволяет определить положение осей как

axes([*left, bottom, width, height*]),

где все значения изменяются от 0 до 1. Выглядеть это может так:

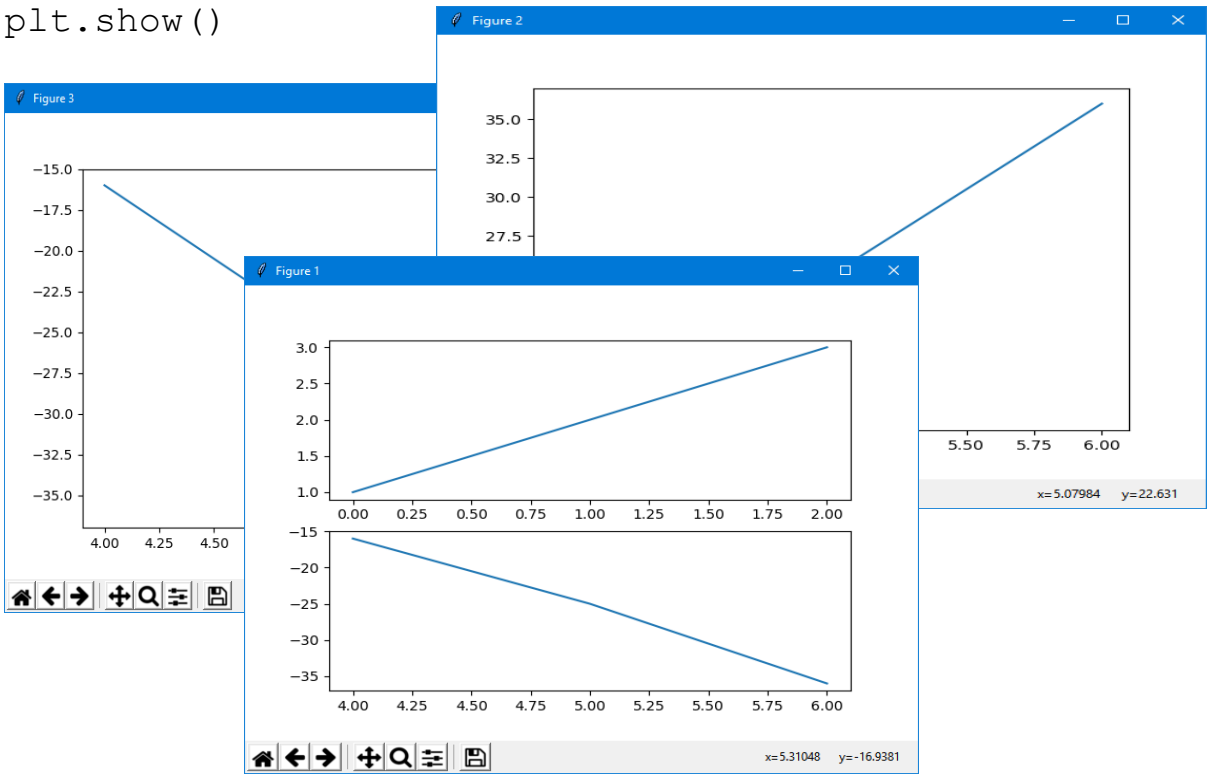
```
import math
import numpy as np
import matplotlib.pyplot as plt
# Импортируем пакет со вспомогательными функциями
import matplotlib.mlab as ml
import numpy as np
import matplotlib.pyplot as plt
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.figure(1)
plt.axes([0.55, 0.55, 0.4, 0.4])
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.axes([0.3, 0.3, 0.4, 0.4])
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.axes([0.05, 0.05, 0.4, 0.4])
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
# Покажем окно с нарисованным графиком
plt.show()
```



Имеется возможность создать несколько окон, вызывая ***figure()*** с увеличивающимся номером окна. Конечно, каждое окно может содержать несколько осей и подокон.

Очистить активное окно можно при помощи ***clf()*** , а активные оси при помощи ***cla()***.

```
import math
import numpy as np
import matplotlib.pyplot as plt
# Импортируем пакет со вспомогательными функциями
import matplotlib.mlab as ml
import numpy as np
import matplotlib.pyplot as plt

plt.figure(1)                    # the first figure
plt.subplot(211)                 # the first subplot in the
first figure
plt.plot([1,2,3])
plt.subplot(212)                 # the second subplot in
the first figure
plt.plot([4,5,6],[-16,-25,-36])
plt.figure(2)                    # a second figure
plt.plot([4,5,6],[16,25,36]) # creates a subplot(111)
                                                  by default
plt.figure(3)
plt.plot([4,5,6],[-16,-25,-36])
plt.show()
```



Установить свой диапазон осей

```
plt.xlim(right=xmax)  #xmax is your value
plt.xlim(left=xmin)   #xmin is your value
plt.ylim(top=ymax)    #ymax is your value
plt.ylim(bottom=ymin) #ymin is your value
```

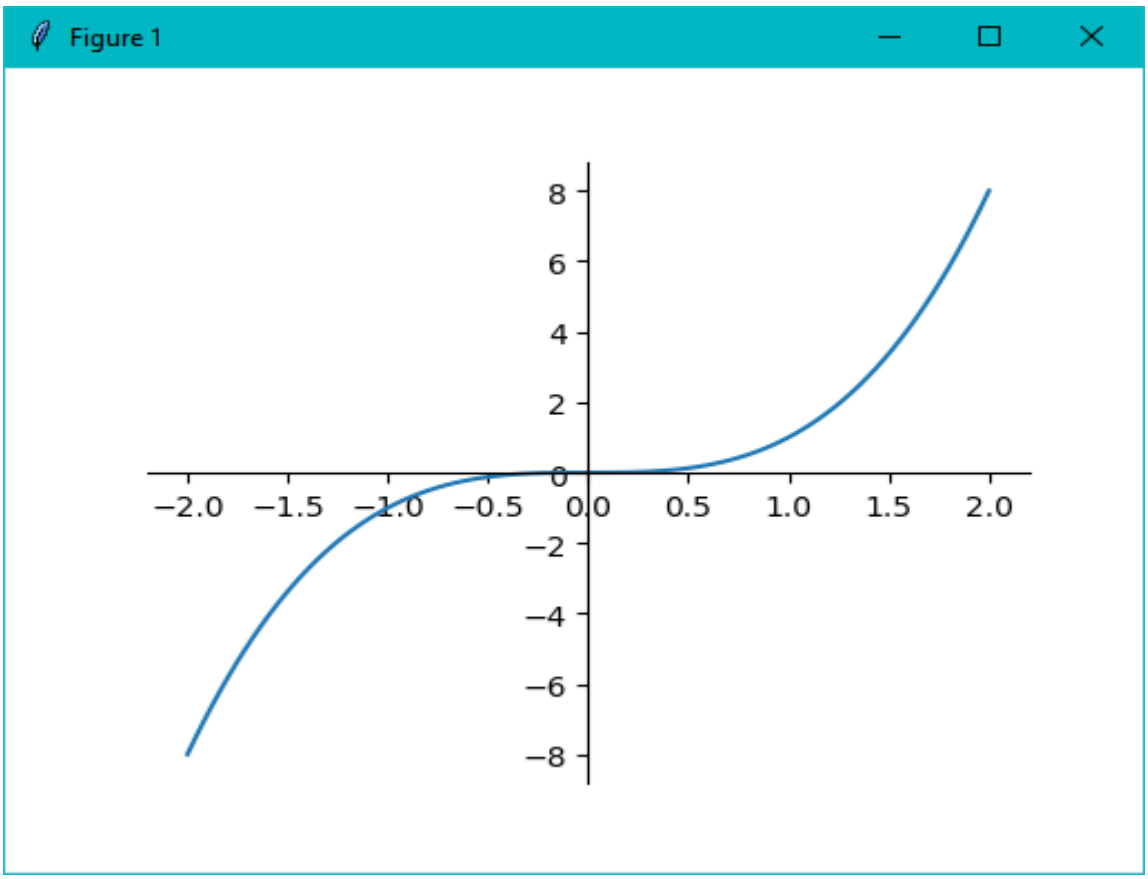
Или так:

```
plt.ylim(ymin, ymax)
plt.xlim(xmin, xmax)
```

Перенос координатных осей в центр графика

Если необходимо пренести координатные оси в «центр рисунка», можно поступить так:

```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(-2,2,100)
Y = X**3
plt.plot(X, Y)
ax = plt.gca()
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.show()
```

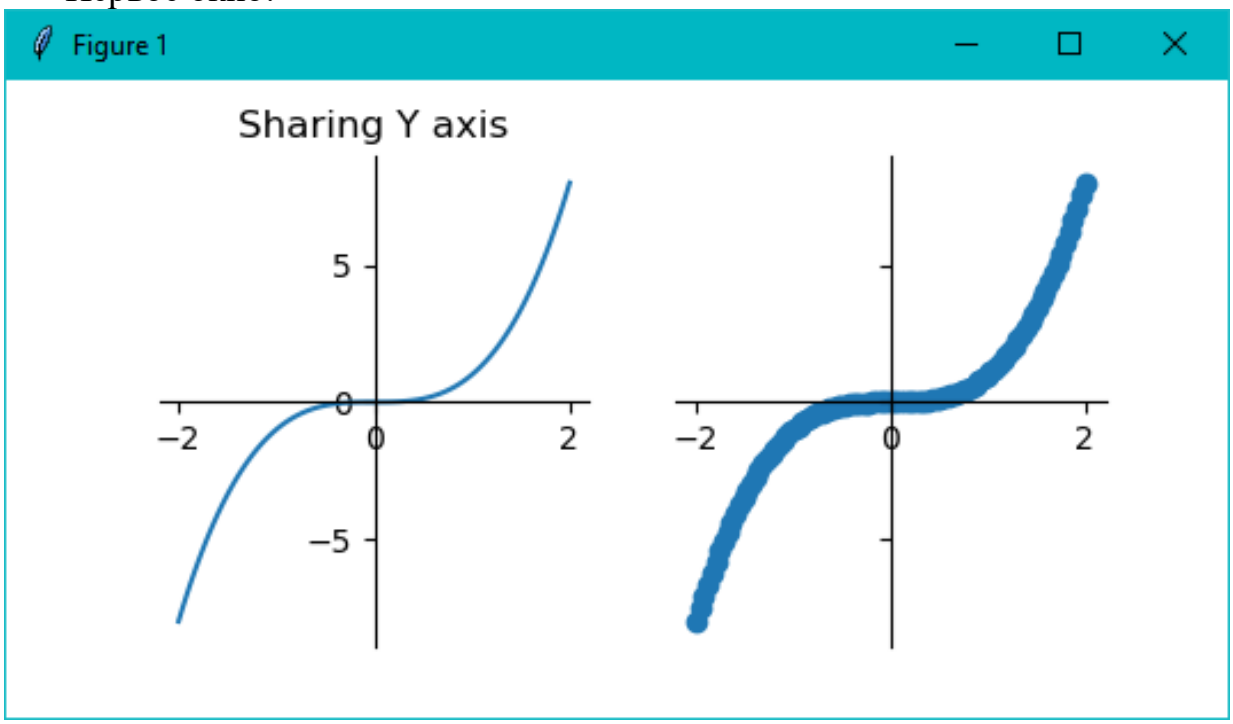


Пример переноса осей для **subplot:**

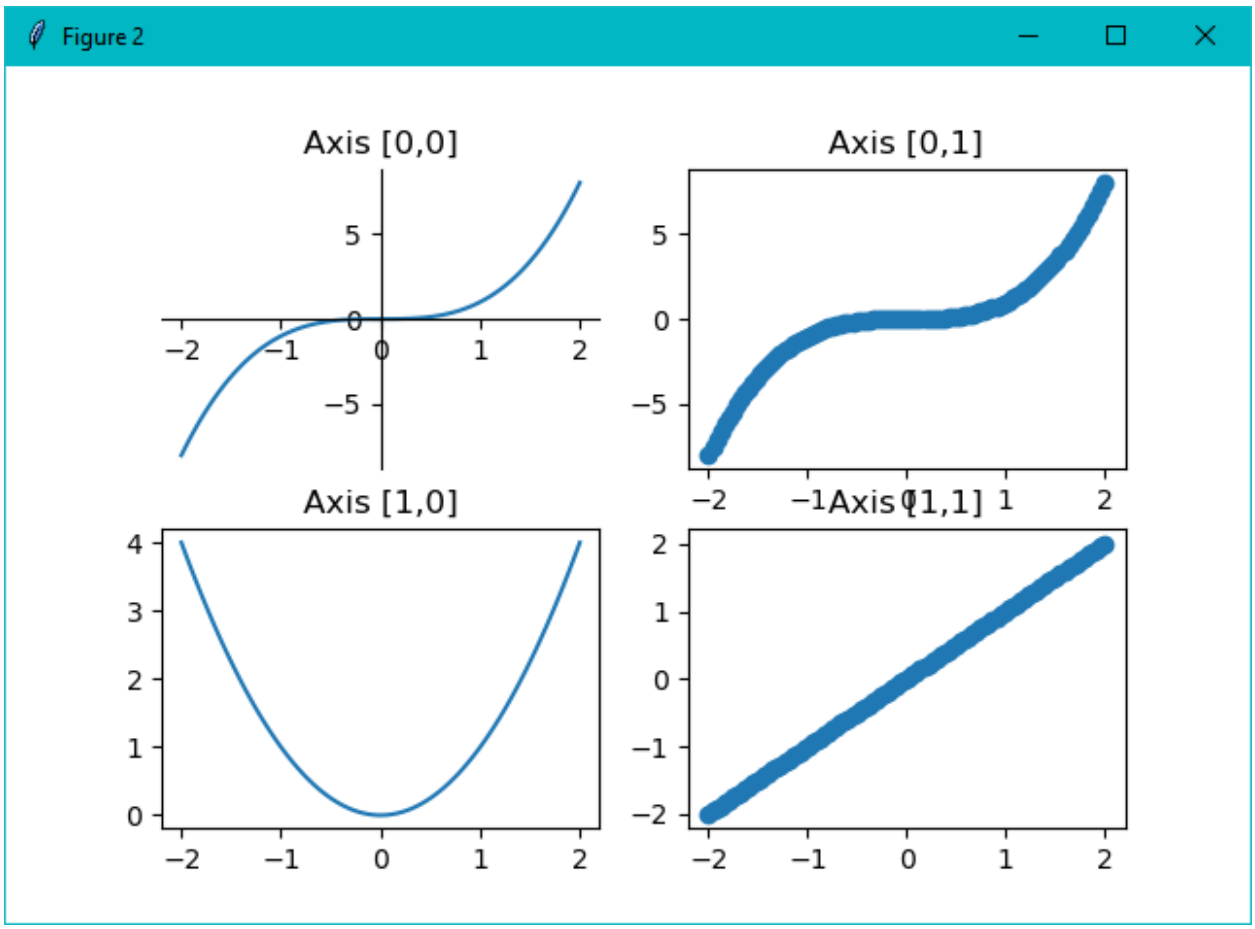
```
#Пример получения осей для subplots:
# Two subplots, unpack the axes array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(X, Y)
ax1.set_title('Sharing Y axis')
ax1.spines['left'].set_position('center')
ax1.spines['bottom'].set_position('center')
ax1.spines['top'].set_visible(False)
ax1.spines['right'].set_visible(False)

ax2.scatter(X, Y)
ax2.spines['left'].set_position('center')
ax2.spines['bottom'].set_position('center')
ax2.spines['top'].set_visible(False)
ax2.spines['right'].set_visible(False)

f, axarr = plt.subplots(2, 2) # Four axes,
                              returned as a 2-d array

axarr[0, 0].plot(X, Y)
axarr[0, 0].spines['left'].set_position('center')
axarr[0, 0].spines['bottom'].set_position('center')
axarr[0, 0].spines['top'].set_visible(False)
```

```
axarr[0, 0].spines['right'].set_visible(False)
axarr[0, 0].set_title('Axis [0,0]')

axarr[0, 1].scatter(X, Y)
axarr[0, 1].set_title('Axis [0,1]')
axarr[1, 0].plot(X, X ** 2)
axarr[1, 0].set_title('Axis [1,0]')
axarr[1, 1].scatter(X, X ** 1)
axarr[1, 1].set_title('Axis [1,1]')

plt.show()
```

Первое окно:



Второе окно:

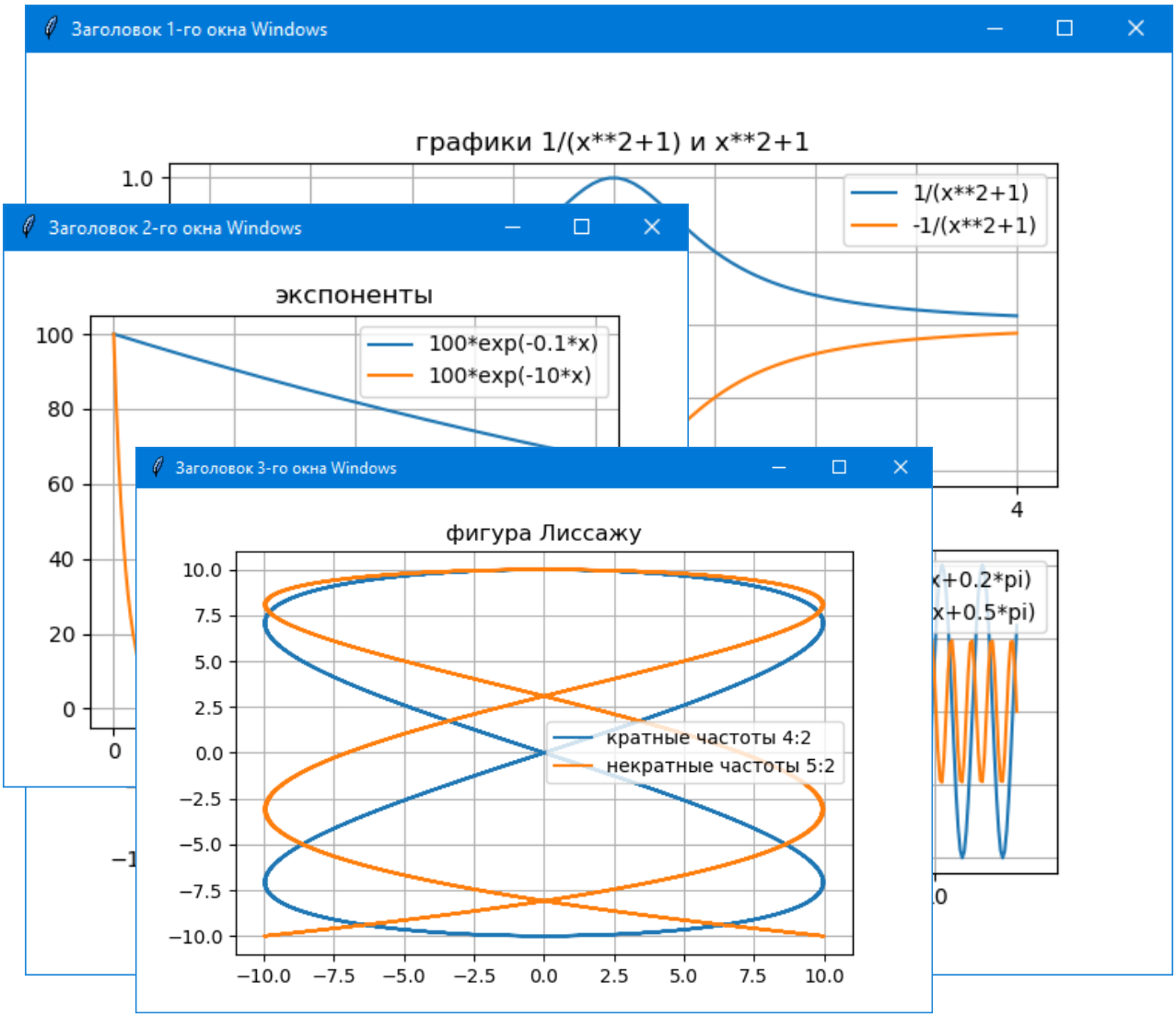


П8 Надписи на окнах диаграмм и окнах Windows, изменение размера окна

Приведем пример, аналогичный рассмотренному выше. Для всех окон и диаграмм заданы титульные (тульбарные) надписи:

```
import math
import numpy as np
import matplotlib.pyplot as plt
# Импортируем пакет со вспомогательными функциями
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt

x1=np.linspace(-4,4,100)
y1=1/(x1**2+1)
y11=-1/(x1**2+1)
# Зададим размеры изображения диаграммы
mpl.rcParams['figure.figsize'] = (8.0, 6.0)
plt.figure("Заголовок 1-го окна Windows")     # первое
                                              # окно windows
plt.subplot(211)        # первые графики в этом окне
plt.plot(x1,y1,label='1/(x**2+1)')
plt.plot(x1,y11,label='-1/(x**2+1)')
```

```
plt.title("графики 1/(x**2+1) и x**2+1" )
plt.grid(True)
plt.legend()
plt.subplot(212)   # вторые графики в этом окне
x2=np.linspace(-4*np.pi,4*np.pi,400)
plt.plot(x2,10*np.sin(5*x2+0.2*np.pi),label='10*sin(5*x
+0.2*pi)')
plt.plot(x2,5*np.cos(10*x2+0.5*np.pi),label="5*cos(10*x
+0.5*pi)")
plt.title("тригонометрические функции" )
plt.grid(True)
plt.legend(loc='best')

plt.figure("Заголовок 2-го окна Windows")  # a second
                                                figure
# или так
#fig = plt.figure()
#fig.canvas.set_window_title("Заголовок 2-го окна
                                               Windows")
x3=np.linspace(0,4,400)
plt.plot(x3, 100*np.exp(-0.1*x3), label='100*exp(-
0.1*x)')
plt.plot(x3, 100*np.exp(-10*x3), label='100*exp(-
10*x)')
plt.title("экспоненты" )
plt.grid(True)
plt.legend(loc='best')
fig3=plt.figure()
fig3.canvas.set_window_title("Заголовок 3-го окна
                                               Windows")
plt.grid(True)
t=np.linspace(0,8*np.pi,500)
plt.plot(10*np.sin(4*t),10*np.cos(2*t),
                         label="кратные частоты 4:2")
plt.plot(10*np.sin(5*t),10*np.cos(2*t),
                        label="некратные частоты 5:2")
plt.title("фигура Лиссажу" )
plt.grid(True)
plt.grid(True)
plt.legend(loc='best')

plt.show()
```



П9. Параметрический график

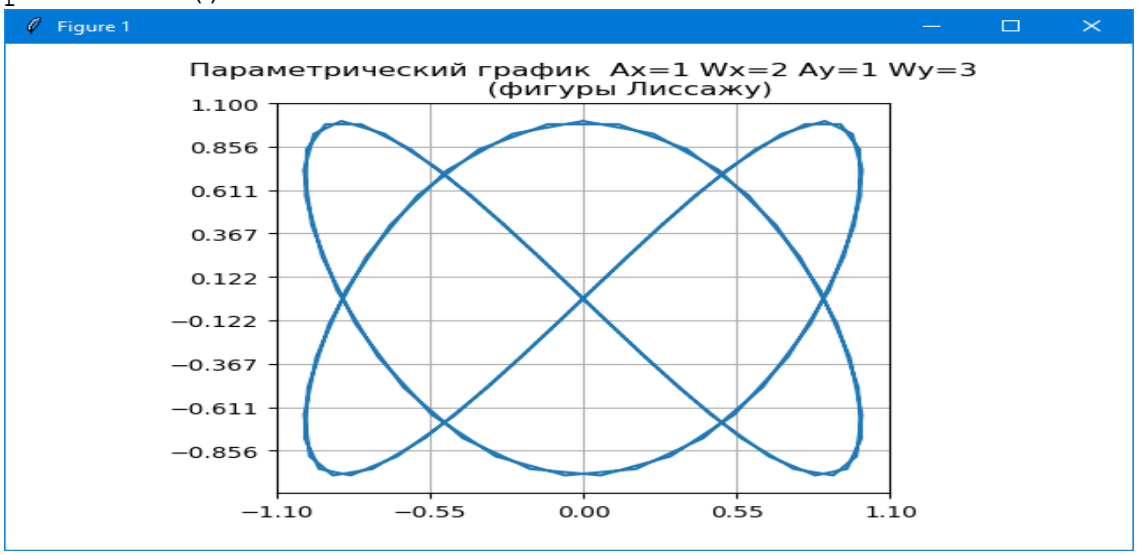
Массив *x* не обязан быть монотонно возрастающим. Можно строить любую параметрическую линию `x=x(t), y=y(t)`.

Разрешающие способности монитора по разным осям могут быть разными. Поэтому чтобы окружности выглядели как окружности, а не как эллипсы, (а квадраты как квадраты, а не как прямоугольники), нужно установить параметр - aspect ratio (обычно равный 1).

Построим `x=Ax*cos(Wx*t), y=Ay*sin(Wy*t)`.

```
import matplotlib.pyplot as plt
import numpy as np
Ax=1; Ay=1; Wx=2; Wy=3
# независимая переменная
t=np.linspace(0,4*np.pi,100)
# установим параметр - aspect(ratio)
axx=plt.subplot(111)
axx.set_aspect(1)
```

```
# так можно изменить шаг сетки на графике
plt.xticks([i for i in   \
              np.linspace(-1.1*Ax, 1.1*Ax, 5)])
plt.yticks([i for i in \
             np.linspace(-1.1*Ay, 1.1*Ay, 10)])
# Задаем заголовок диаграммы
plt.title("Параметрический график " \
          " Ax={0} Wx={1} Ay={2} Wy={3}\n" \
          "           (фигуры Лиссажу)"  \
               .format(Ax, Wx, Ay, Wy) )
plt.plot(Ax*np.sin(Wx*t),Ay*np.cos(Wy*t))
plt.grid(True)
# визуализируем графики
plt.show()
```



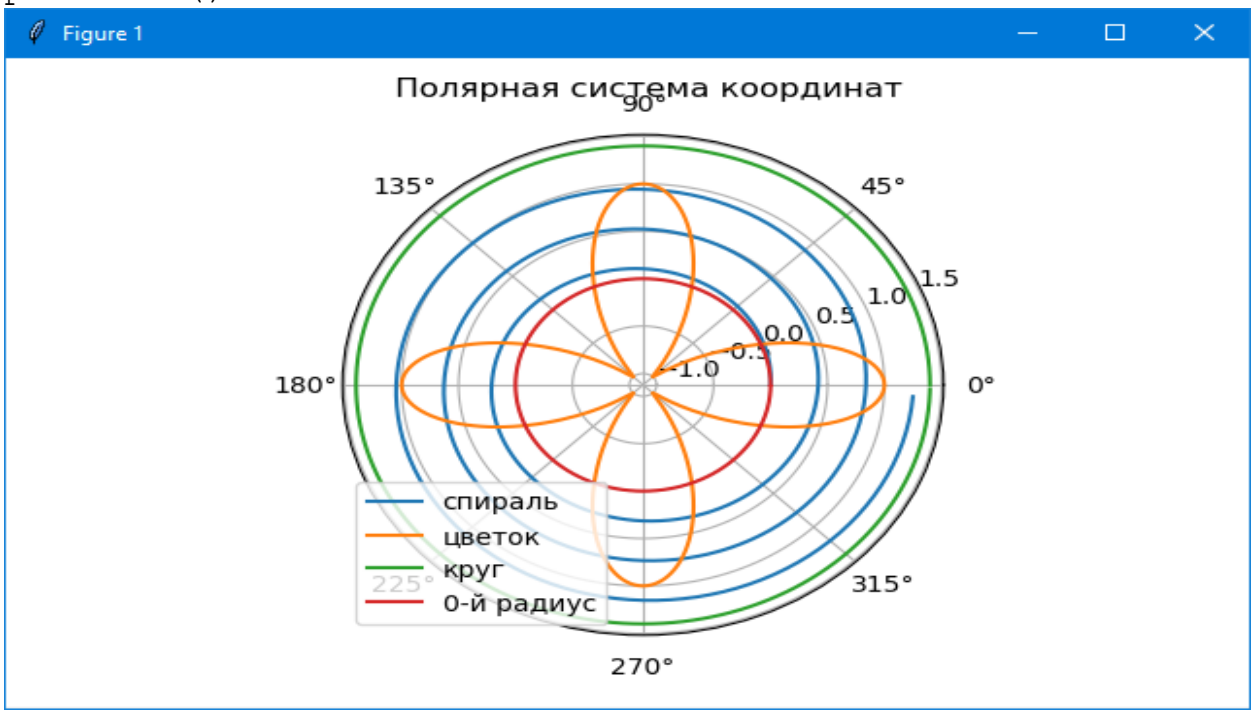
П10. Полярные координаты

Кроме наиболее часто используемой декартовой системы координат, довольно широко применяется и полярная система координат, удобная в различных радиальных задачах, координаты точек в ней задается с помощью радиус-вектора **ρ**, идущего из начала координат и угла **θ**. Угол может быть задан в радианах или градусах, `matplotlib` использует градусы.

Четыре графика в полярных координатах:

```
import matplotlib.pyplot as plt
import numpy as np
theta = np.arange(0., 2., 1./180.)*np.pi
#plt.title('Поляная система координат')
plt.polar(3*theta, theta/5, label="спираль");
plt.polar(theta, 0+np.cos(4*theta), label="цветок");
plt.polar(theta, [1.4]*len(theta), label="круг");
plt.polar(theta, 0*theta, label="0-й радиус");
plt.title(" Полярная система координат")
lt.grid(True)
```

```
plt.legend(loc='lower left')
# визуализируем графики
plt.show()
```



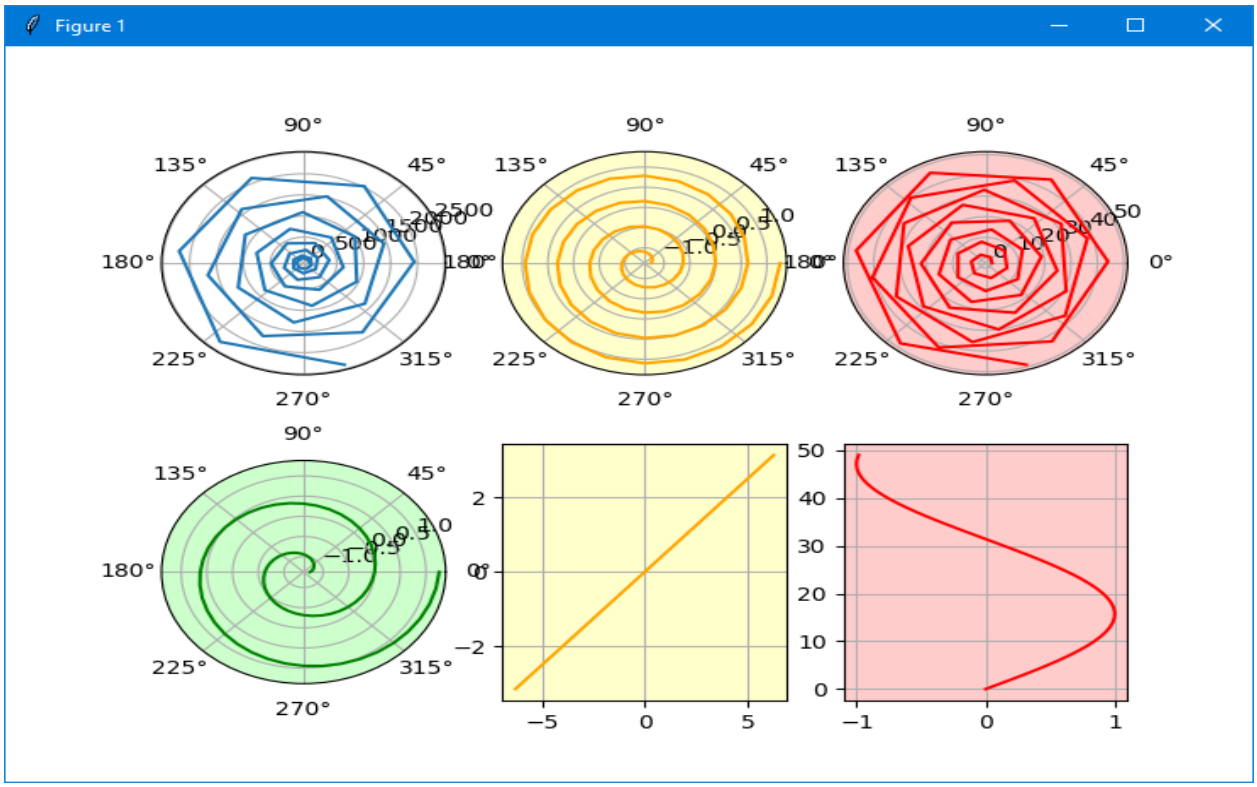
Для построения в полярных координат используется функция `polar()`. В аргументах первыми идут углы, потом радиусы. В примере используется один и тот же массив для углов и радиус-векторов и рисуется четыре разные кривые.

Первая образует спираль, поскольку с каждым новым углом меняется и радиус, вторая рисует цветок в соответствии с уравнением розы, третья и четвертая - окружности ввиду неизменности радиуса для всех углов (в данном случае он равен 1.4 и 0).

Еще один пример:

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(8., 6.))
x = np.arange(50)
y = x**2 + 2.5

lag = 0.05*np.pi
r = np.arange(0.0, 5.0, 0.1)
#phi = r*np.pi
phi=np.arange(-2*np.pi, 2*np.pi + lag, lag)
r=phi*0.2
# x и y
ax1 = fig.add_subplot(231, projection='polar')
ax1.plot(x, y)
ax1.grid(True)
```

```
ax2 = fig.add_subplot(232, projection='polar',
facecolor ='#FFFFCC')
ax2.plot(phi*2., r, 'orange')       #
facecolor='#FFFFCC') # axisbg
ax2.grid(True)

ax3 = fig.add_subplot(233, projection='polar',
facecolor='#FFCCCC')
ax3.plot(x, x, 'r')
ax3.grid(True)

ax4 = fig.add_subplot(234, projection='polar',
facecolor='#CCFFCC')
ax4.plot(phi, 0.2*phi, 'g') #(-1.5*phi, r, 'g')
ax4.grid(True)

ax5 = fig.add_subplot(235, facecolor='#FFFFCC')
ax5.plot(phi, phi*0.5, 'orange')
ax5.grid(True)

ax6 = fig.add_subplot(236, facecolor='#FFCCCC')
ax6.plot(np.sin(0.1*x), x, 'r')
ax6.grid(True)
###plt.legend(loc='lower left')
plt.show()
```



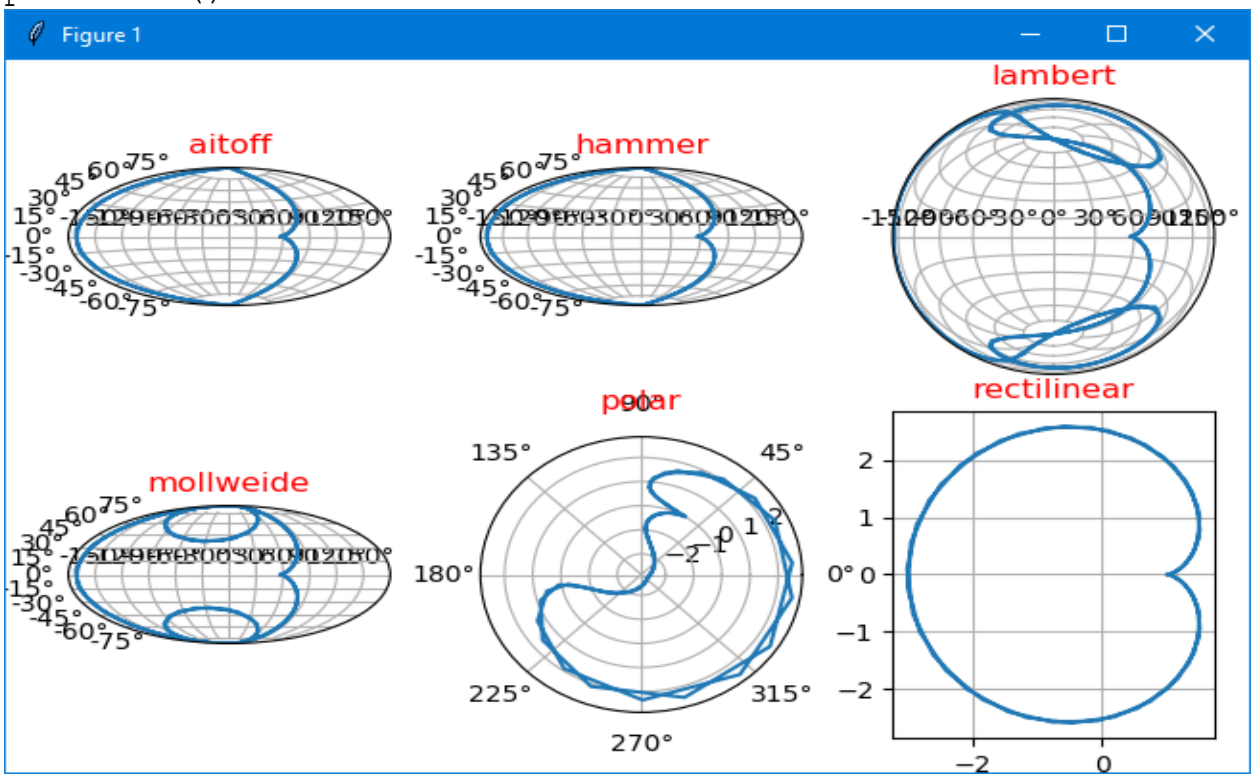
matplotlib позволяет рисовать не только в полярной системе координат, но и в других. За это отвечает параметр `projection`, который в случае равенства значению `'polar'` аналогичен значению параметра `polar=True`.

В matplotlib поддерживаются следующие проекции:

`'aitoff'`, `'hammer'`, `'lambert'`, `'mollweide'`, `'polar'`, `'rectilinear'`.

Чаще всего используются два последних варианта, остальные являются экзотическими для рутинных задач в научной графике.

```
import matplotlib.pyplot as plt
import numpy as np
a = 1.
x = np.arange(-2*np.pi, 2*np.pi, 0.2)
y = np.sin(x) * np.cos(x)
# Уравнение кардиойды
xz = a*(2*np.cos(x) - np.cos(2*x))
yz = a*(2*np.sin(x) - np.sin(2*x))
label = ['aitoff', 'hammer', 'lambert', 'mollweide',
'polar', 'rectilinear']
fig = plt.figure(figsize=(10,8))

for i in range(len(label)):
    ax = fig.add_subplot(231+i, projection=label[i])
    ax.plot(xz, yz)
    ax.set_title(label[i], color='r')
    ax.grid(True)
plt.tight_layout()
plt.show()
```



У полярной системы координат есть специальные функции, которые позволяют настраивать внешний вид рисунка.

Для радиуса R:

1. `ax.set_rlabel_position(phi)` - перемещает ось ординат (радиус) по кругу на угол phi (в ГРАДУСАХ) от положения нуля;
2. `ax.set_rmax(R)` - позволяет ограничить область изменения радиуса R на рисунке;
3. `ax.set_rmin(R)` - позволяет ограничить область изменения радиуса R на рисунке;
4. `ax.set_rlim()` - позволяет ограничить область изменения радиуса R на рисунке;
5. `ax.set_rscale()` - позволяет сделать шкалу радиусов логарифмической;
6. `ax.set_rgrid()` - позволяет настроить для оси радиуса вспомогательную сетку, положения делений, формат подписей к ним и т.д.

Для угла **phi** (в matplotlib он называется **theta**):

1. `ax.set_theta_zero_location(loc)` - перемещает положение нуля на определённое положение. `Loc` принимает значения `'N', 'NW', 'W', 'SW', 'S', 'SE', 'E'` или `'NE'`. По умолчанию положение нуля находится в положении "восток" или "3 часа";
2. `ax.set_theta_offset(phi)` - перемещает положение нуля на угол phi (в радианах) . По умолчанию положение нуля находится в положении "восток" или "3 часа";
3. `ax.set_theta_direction(loc)` - определяет направление обхода. `Loc` может быть либо -1 или по часовой стрелке и 1 или против часовой стрелки;
4. `ax.set_thetagrids()` - позволяет настроить для оси углов вспомогательную сетку, положения делений, формат подписей к ним и т.д.

П11. График рассеяния

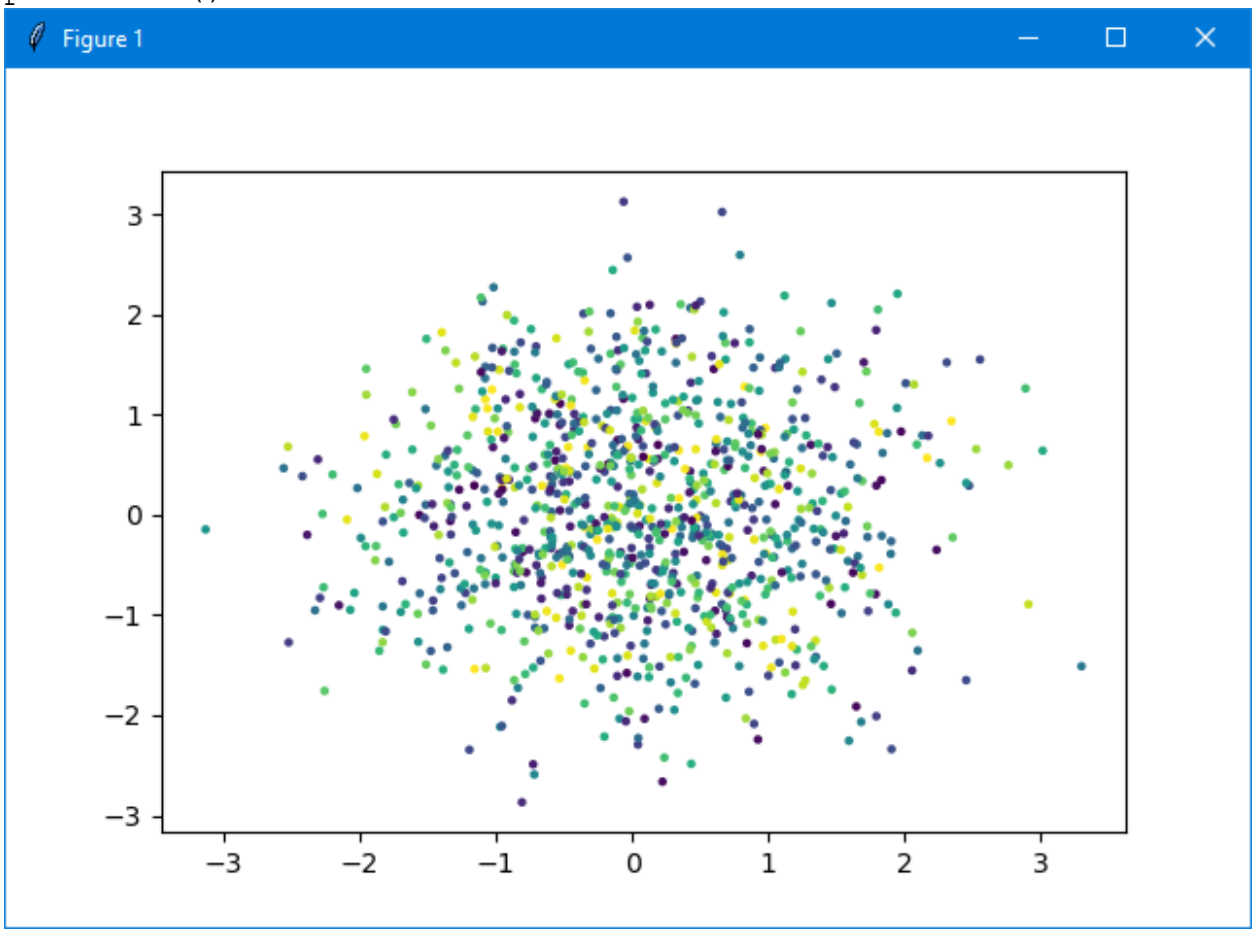
Такой тип графиков позволяет изображать одновременно два множества данных, которые не образуют кривой, а именно двухмерное множество точек. Каждая точка имеет две координаты. График рассеяния часто используется для определения связи между двумя величинами и позволяет определить более точные пределы измерений.

В модуле `matplotlib.pyplot` имеется своя функция, для графика рассеяния (`scatter plot`) это функция `scatter()`.

Она принимает две последовательности и изображает их на плоскости в виде маркеров, по умолчанию они круглые и синие. Но естественно, с ними можно поработать с помощью `keywords` той же функции:

- **`s`** задает размер маркеров и может быть как одним числом для всех, так и представлять массив значений
- **`c`** задает цвет маркеров, также либо один для всех, либо множество
- **`marker`** определяет тип маркера.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randn(1000)
y = np.random.randn(1000)

size = 5
colors = np.random.rand(1000)

plt.scatter(x, y, s=size, c=colors)
plt.show()
```



П12. Настройки в стиле LaTeX

Вот пример настройки почти всего, что можно настроить.

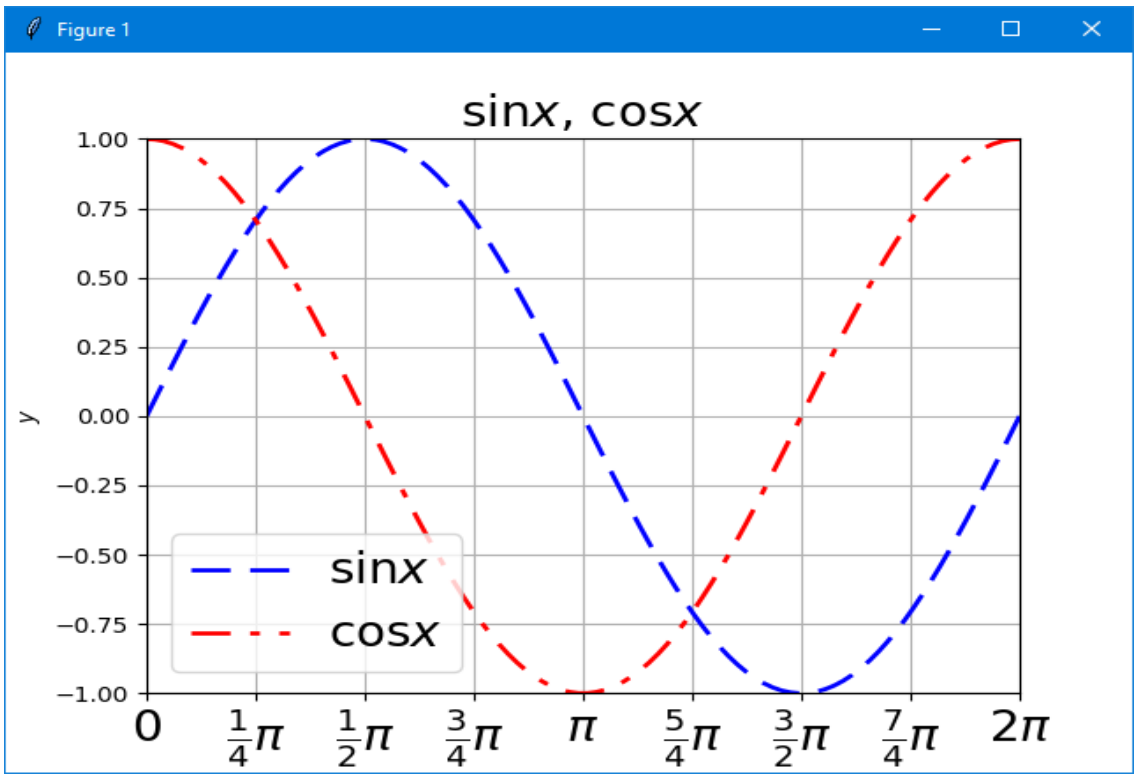
Можно задать последовательность засечек на оси *x* (и *y*) и подписи к ним (в них, как и в других текстах, можно использовать *LATEX*-овские обозначения).

Задать подписи осей *x* и *y* и заголовок графика.

Во всех текстовых элементах можно задать размер шрифта. Можно задать толщину линий и штрихи (так, на графике косинуса рисуется штрих длины 8, потом участок длины 4 не рисуется, потом участок длины 2 рисуется, потом участок длины 4 опять не рисуется, и так по циклу; поскольку толщина линии равна 2, эти короткие штрихи длины 2 фактически выглядят как точки).

Можно задать подписи к кривым (legend); где разместить эти подписи тоже можно регулировать

```
import matplotlib.pyplot as plt
import numpy as np
plt.axis([0,2*np.pi,-1,1])

plt.xticks(np.linspace(0,2*np.pi,9),
       ('0',r'$\frac{1}{4}\pi$',r'$\frac{1}{2}\pi$',

r'$\frac{3}{4}\pi$',r'$\pi$',r'$\frac{5}{4}\pi$',

r'$\frac{3}{2}\pi$',r'$\frac{7}{4}\pi$',r'$2\pi$'),
      fontsize=20)

plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'$\sin x$, $\cos x$',fontsize=20)

x=np.linspace(0,2*np.pi,100)
plt.plot(x,np.sin(x),linewidth=2,
                     color='b',dashes=[8,4],
                                label=r'$\sin x$')
plt.plot(x,np.cos(x),linewidth=2,
                    color='r',dashes=[8,4,2,4],
                                label=r'$\cos x$')
plt.legend(fontsize=20)
plt.grid(True)
# визуализируем графики
plt.show()
```

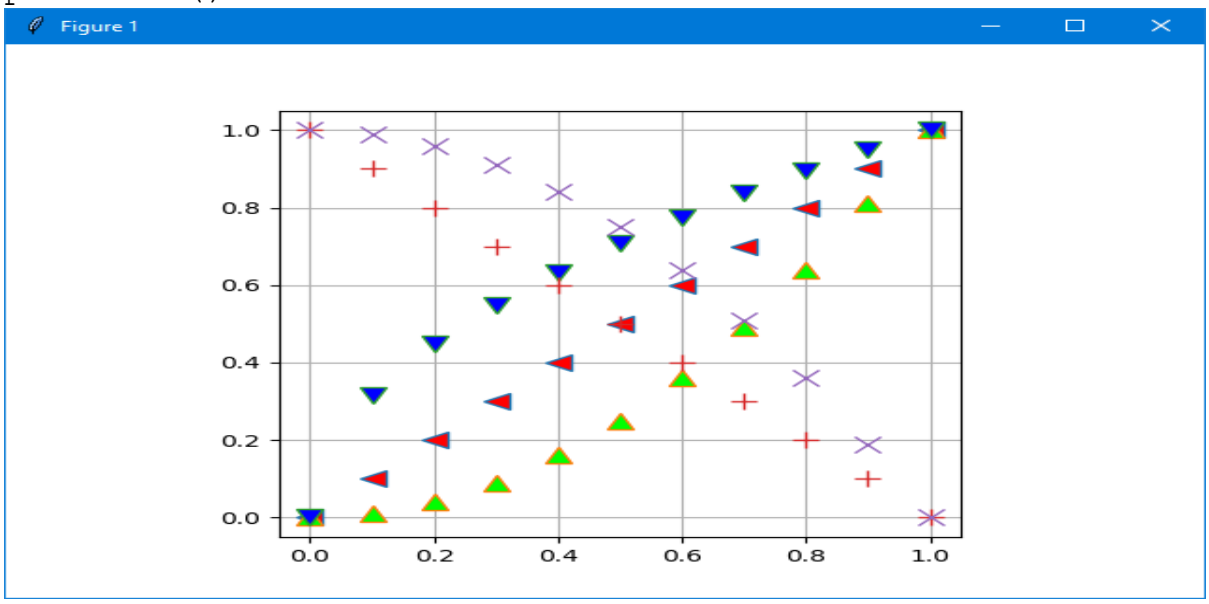


П13. Модифицированные маркеры

Если `linestyle=''`, то точки не соединяются линиями. Сами точки рисуются маркерами разных типов. Тип определяется строкой из одного символа, который чем-то похож на нужный маркер. В добавок к стандартным маркерам, можно определить самодельные.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(0,1,11)
# установим параметр - aspect(ratio)
axx=plt.subplot(111)
axx.set_aspect(1)
plt.axis([-0.05,1.05,-0.05,1.05])
plt.plot(x,x,linestyle='',marker='<',markersize=10,
        markerfacecolor='#FF0000')
plt.plot(x,x**2,linestyle='',marker='^',markersize=10,
        markerfacecolor='#00FF00')
plt.plot(x,x**(1/2),linestyle='',marker='v',markersize=
10,
        markerfacecolor='#0000FF')
plt.plot(x,1-x,linestyle='',marker='+',markersize=10,
        markerfacecolor='#0F0F00')
plt.plot(x,1-
x**2,linestyle='',marker='x',markersize=10,
     markerfacecolor='#0F000F')
```

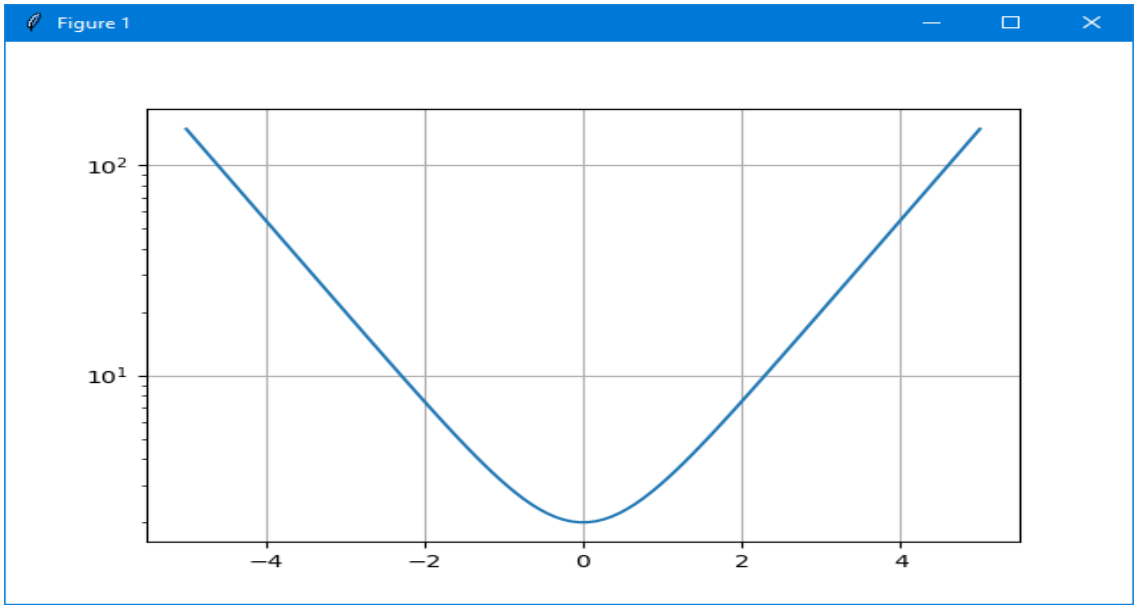
```
plt.grid(True)
# визуализируем графики
plt.show()
```



П14. Логарифмический масштаб

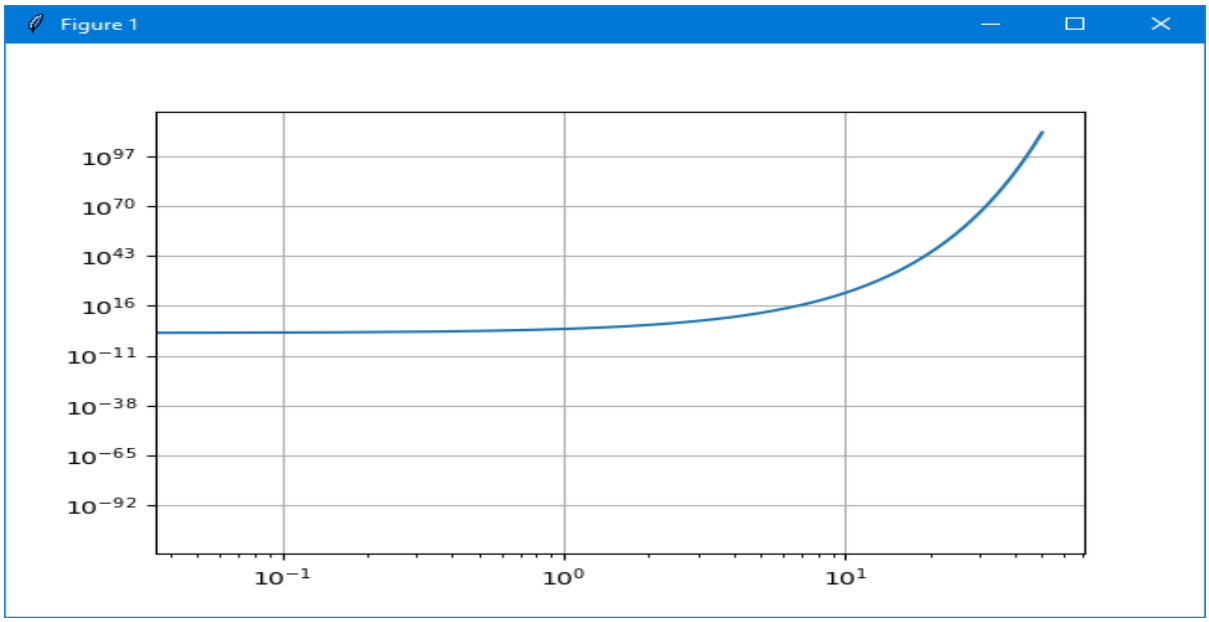
Если **y** меняется на много порядков, то удобно использовать логарифмический масштаб по **y**

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
plt.yscale('log')
plt.plot(x,np.exp(x)+np.exp(-x))
plt.grid(True)
# визуализируем графики
plt.show()
```



Можно задать логарифмический масштаб по обоим осям.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-50,50,1000)
plt.xscale('log')
plt.yscale('log')
plt.plot(x,np.exp(5*x+3))
plt.grid(True)
# визуализируем графики
plt.show()
```



П13. Экпериментальные данные

Допустим, имеется теоретическая кривая (резонанс без фона).

```
xt=np.linspace(-4,4,101)
yt=1/(xt**2+1)
```

Поскольку реальных экспериментальных данных под рукой нет, мы их сгенерируем. Пусть они согласуются с теорией, и все статистические ошибки равны 0.1.

```
xe=np.linspace(-3,3,21)
yerr=0.1*np.ones(21)
ye=1/(xe**2+1)+yerr*np.random.normal(size=21)
```

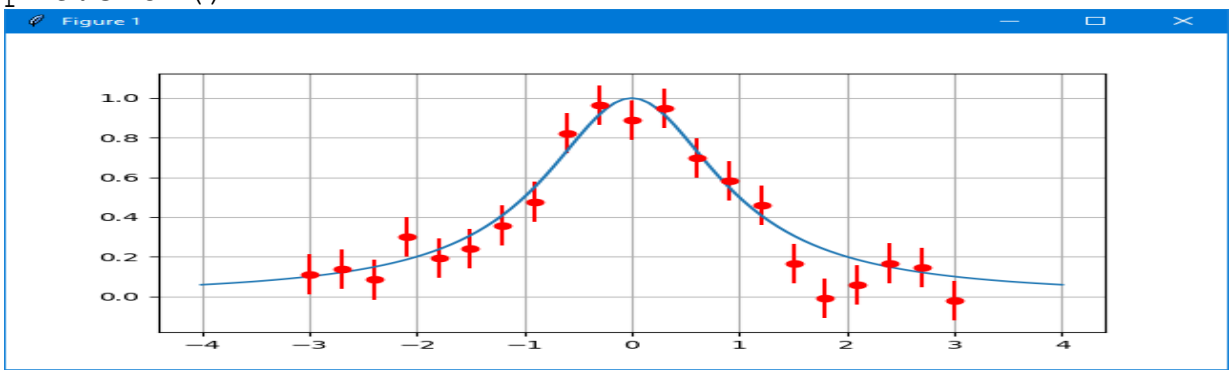
Экспериментальные точки с усами и теоретическая кривая на одном графике.

```
plt.plot(xt,yt)
plt.errorbar(xe,ye,fmt='ro',yerr=yerr)
```

Весь скрипт:

```
import math
```

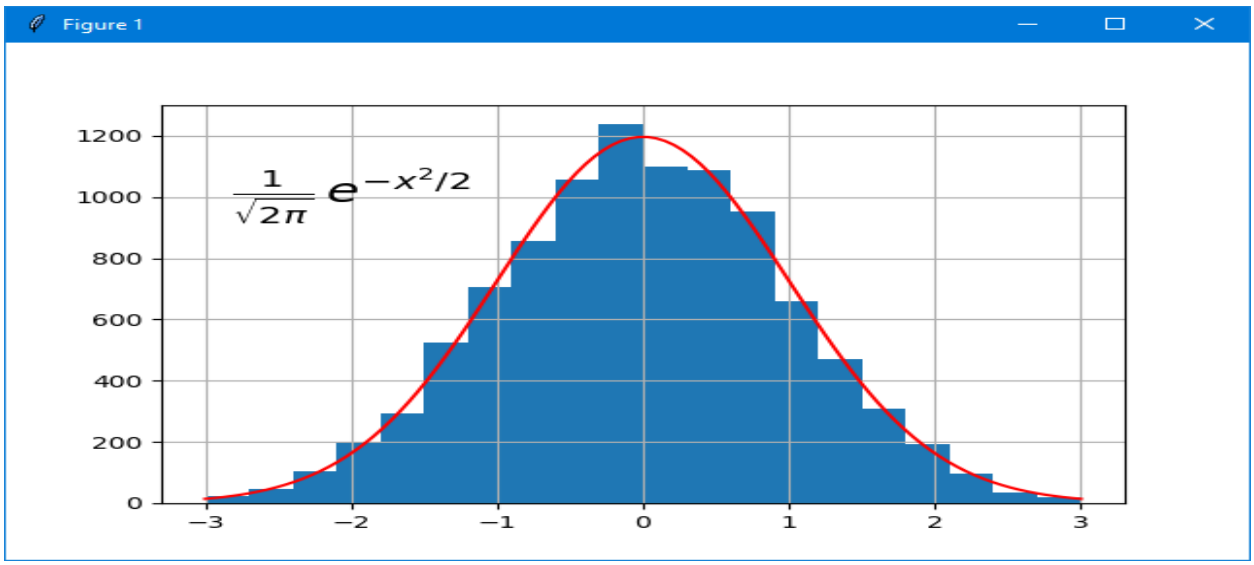
```
import numpy as np
import matplotlib.pyplot as plt
xt=np.linspace(-4,4,101)
yt=1/(xt**2+1)
xe=np.linspace(-3,3,21)
yerr=0.1*np.ones(21)
ye=1/(xe**2+1)+yerr*np.random.normal(size=21)
plt.plot(xt,yt)
plt.errorbar(xe,ye,fmt='ro',yerr=yerr)
plt.grid(True)
# визуализируем графики
plt.show()
```



П15. Гистограмма

Сгенерируем N случайных чисел с нормальным (гауссовым) распределением (среднее 0, среднеквадратичное отклонение 1), и раскидаем их по 20 бинам от −3 до 3 (точки за пределами этого интервала отбрасываются). Для сравнения, вместе с гистограммой нарисуем Гауссову кривую в том же масштабе. И даже напишем формулу Гаусса.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as ml
N=10000
r=np.random.normal(size=N)
n,bins,patches=plt.hist(r,range=(-3,3),bins=20)
x=np.linspace(-3,3,100)
plt.plot(x,N/np.sqrt(2*np.pi)*0.3*
                              np.exp(-0.5*x**2),'r')
plt.text(-2,1000,r'$\frac{1}{\sqrt{2\pi}}
                                      \,e^{-x^2/2}$',
     fontsize=20,horizontalalignment='center',
     verticalalignment='center')
plt.grid(True)
# визуализируем графики
plt.show()
```

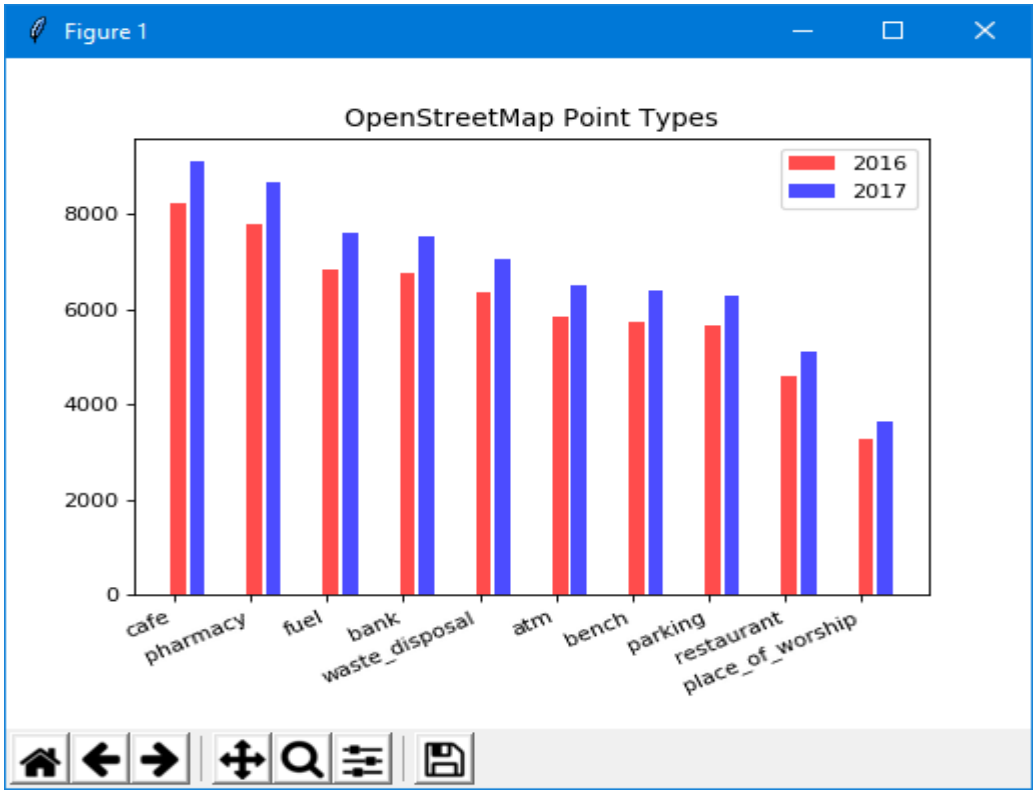


Рассмотрим несколько примеров (https://eax.me/python-matplotlib/) построения специальных диаграммы.

В качестве примера построим диаграмму, отображающую, сколько точек на карте к какому типу (заправка, кафе и так далее) относятся. Чтобы было чуть интереснее, сделаем вид, что в прошлом году точек каждого вида было на 10% меньше, и попытаемся отразить это изменение:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime as dt
import csv
data_names = ['cafe', 'pharmacy', 'fuel', 'bank',
'waste_disposal',
              'atm', 'bench', 'parking', 'restaurant',
              'place_of_worship']
data_values = [9124, 8652, 7592, 7515, 7041, 6487,
6374, 6277,
               5092, 3629]
dpi = 80
fig = plt.figure(dpi = dpi, figsize = (512 / dpi, 384 /
dpi) )
mpl.rcParams.update({'font.size': 10})
plt.title('OpenStreetMap Point Types')
#ax = plt.axes()
#ax.yaxis.grid(True, zorder = 1)
xs = range(len(data_names))
plt.bar([x + 0.05 for x in xs], [ d * 0.9 for d in
                                    data_values],
        width = 0.2, color = 'red', alpha = 0.7,
                                   label = '2016',
        zorder = 2)
```

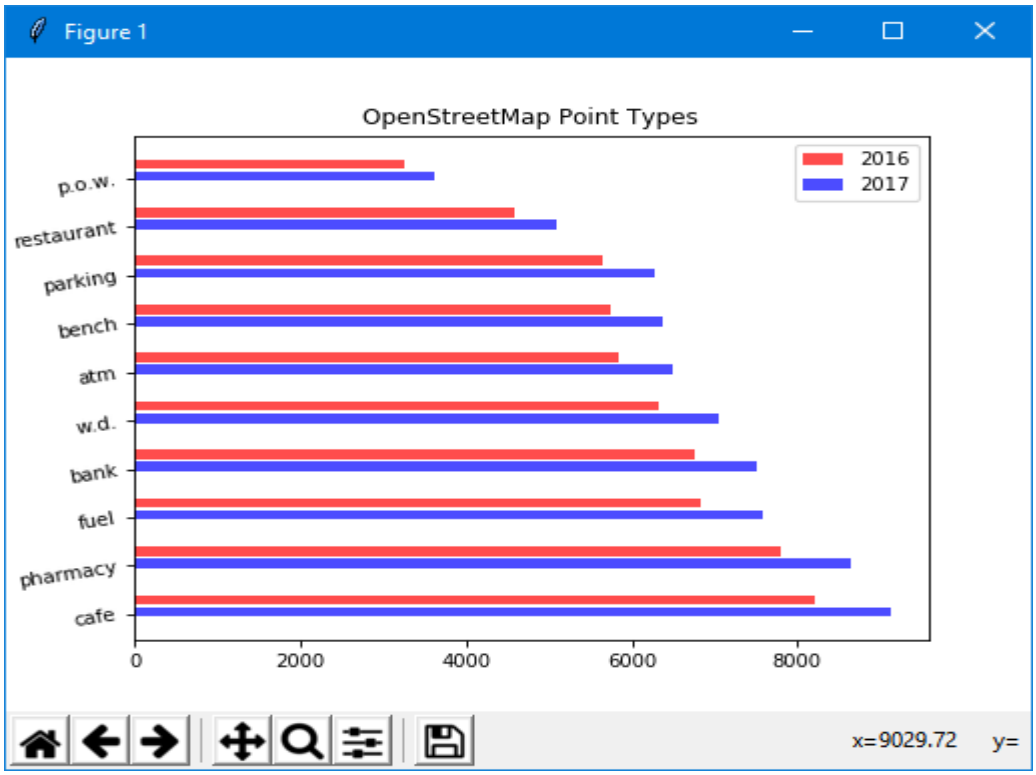
```
plt.bar([x + 0.3 for x in xs], data_values,
        width = 0.2, color = 'blue', alpha = 0.7,
                                        label = '2017',
        zorder = 2)
plt.xticks(xs, data_names)
fig.autofmt_xdate(rotation = 25)
plt.legend(loc='upper right')
#fig.savefig('bars.png')
plt.show()
```



Те же данные можно отобразить, расположив столбики горизонтально:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime as dt
import csv

data_names = ['cafe', 'pharmacy', 'fuel', 'bank',
'w.d.', 'atm','bench', 'parking', 'restaurant',
                                          'p.o.w.']
data_values = [9124, 8652, 7592, 7515, 7041, 6487,
6374, 6277,  5092, 3629]

dpi = 80
fig = plt.figure(dpi = dpi, figsize =
                              (512 / dpi, 384 / dpi) )
```

```
mpl.rcParams.update({'font.size': 9})
plt.title('OpenStreetMap Point Types')
#ax = plt.axes()
#ax.xaxis.grid(True, zorder = 1)
xs = range(len(data_names))
plt.barh([x + 0.3 for x in xs], [ d * 0.9 for d in
data_values],
         height = 0.2, color = 'red', alpha = 0.7,
label = '2016',
         zorder = 2)
plt.barh([x + 0.05 for x in xs], data_values,
         height = 0.2, color = 'blue', alpha = 0.7,
label = '2017',
         zorder = 2)
plt.yticks(xs, data_names, rotation = 10)

plt.legend(loc='upper right')
plt.show()
```



П16. Круговая диаграмма

Для примера визуализируем распределение кафе по различным городам (пример взят с https://eax.me/python-matplotlib/):

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

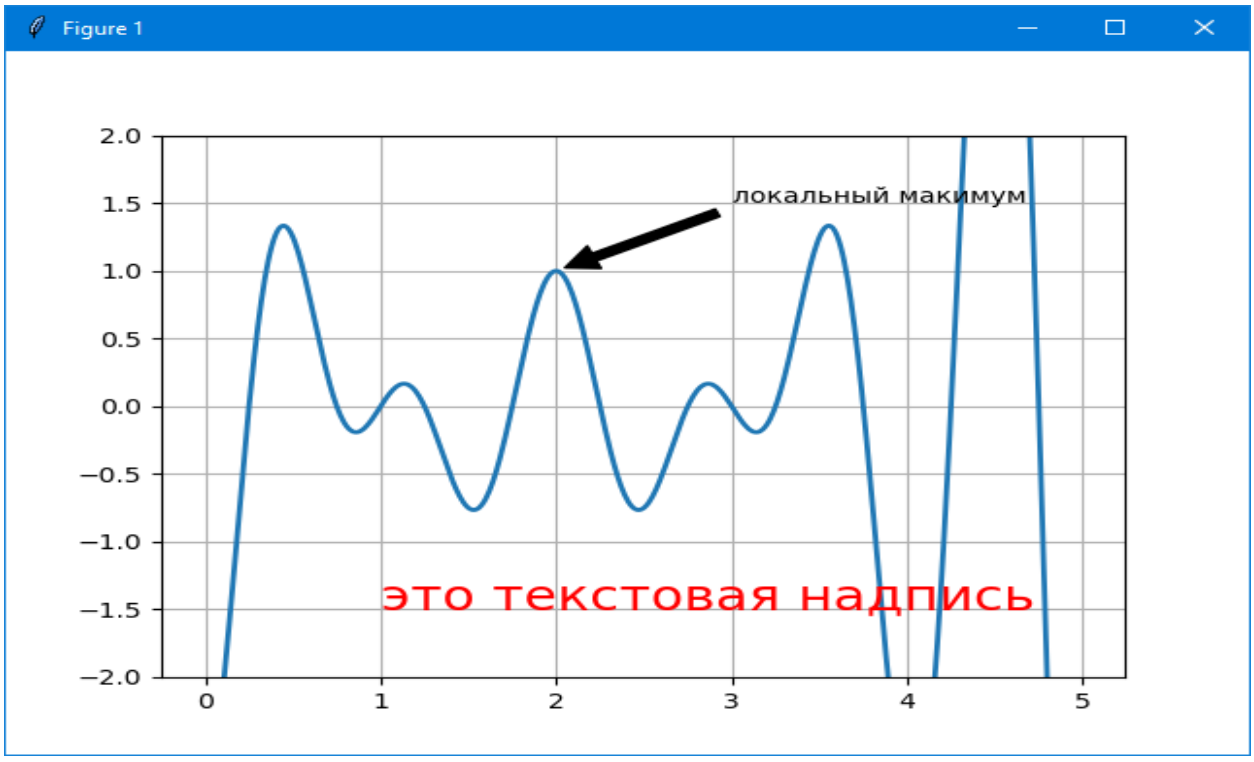
```
import datetime as dt
import csv

data_names = ['Москва', 'Санкт-Петербург', 'Сочи',
'Архангельск',
              'Владимир', 'Краснодар', 'Курск',
'Воронеж',
              'Ставрополь', 'Мурманск']
data_values = [1076, 979, 222, 189, 137, 134, 124, 124,
91, 79]

dpi = 80
fig = plt.figure(dpi = dpi, figsize = (512 / dpi, 384 /
dpi) )
mpl.rcParams.update({'font.size': 9})

plt.title('Распределение кафе по городам России (%)')

xs = range(len(data_names))

plt.pie(
    data_values, autopct='%.1f', radius = 1.1,
    explode = [0.15] + [0 for _ in
range(len(data_names) - 1)] )
plt.legend(
    bbox_to_anchor = (-0.16, 0.45, 0.25, 0.25),
    loc = 'lower left', labels = data_names )
plt.show()
```

Огромное количество примеров можно найти на https://matplotlib.org/gallery.html

П17. Текст и надписи

Текст и дополнительные надписи (аннатации) размещаются на диаграмме:

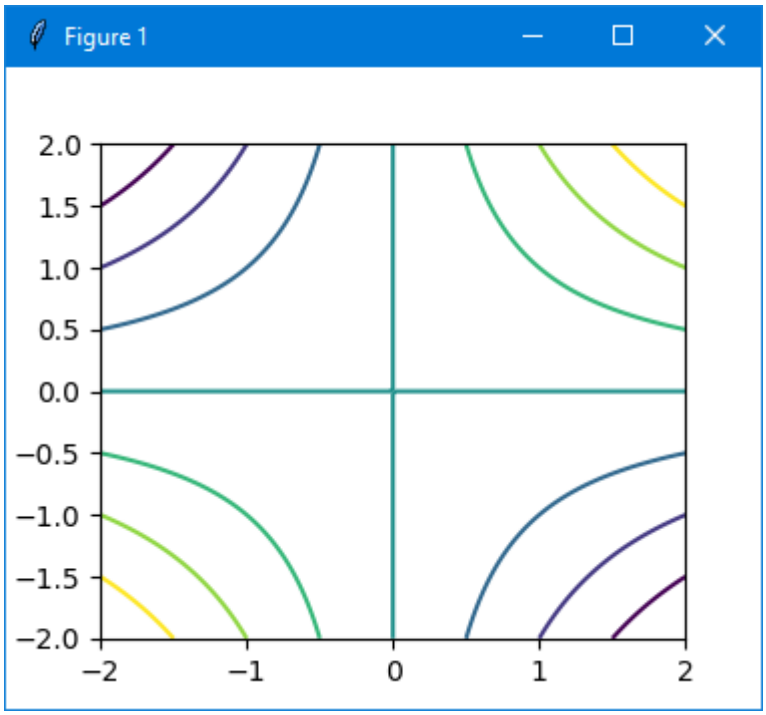
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = (1-(t-2)*(t-2))  * np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)
plt.text(1,-1.5,'это текстовая надпись', size=20,
color='r')
plt.grid(True)
plt.annotate('локальный макимум', xy=(2, 1),
                              xytext=(3, 1.5),
                arrowprops=
                dict(facecolor='black', shrink=0.05), )
plt.ylim(-2,2)
plt.show()
```



П18. Контурные графики

Пусть необходимо хотим изучить поверхность **z=xy.** Построим её «горизонтали»
(взять с http://www.inp.nsk.su/~grozin/python/python6.html):

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
x=np.linspace(-1,1,50)
y=x
z=np.outer(x,y)
plt.contour(x,y,z)
plt.axes().set_aspect(1)
plt.show()
```

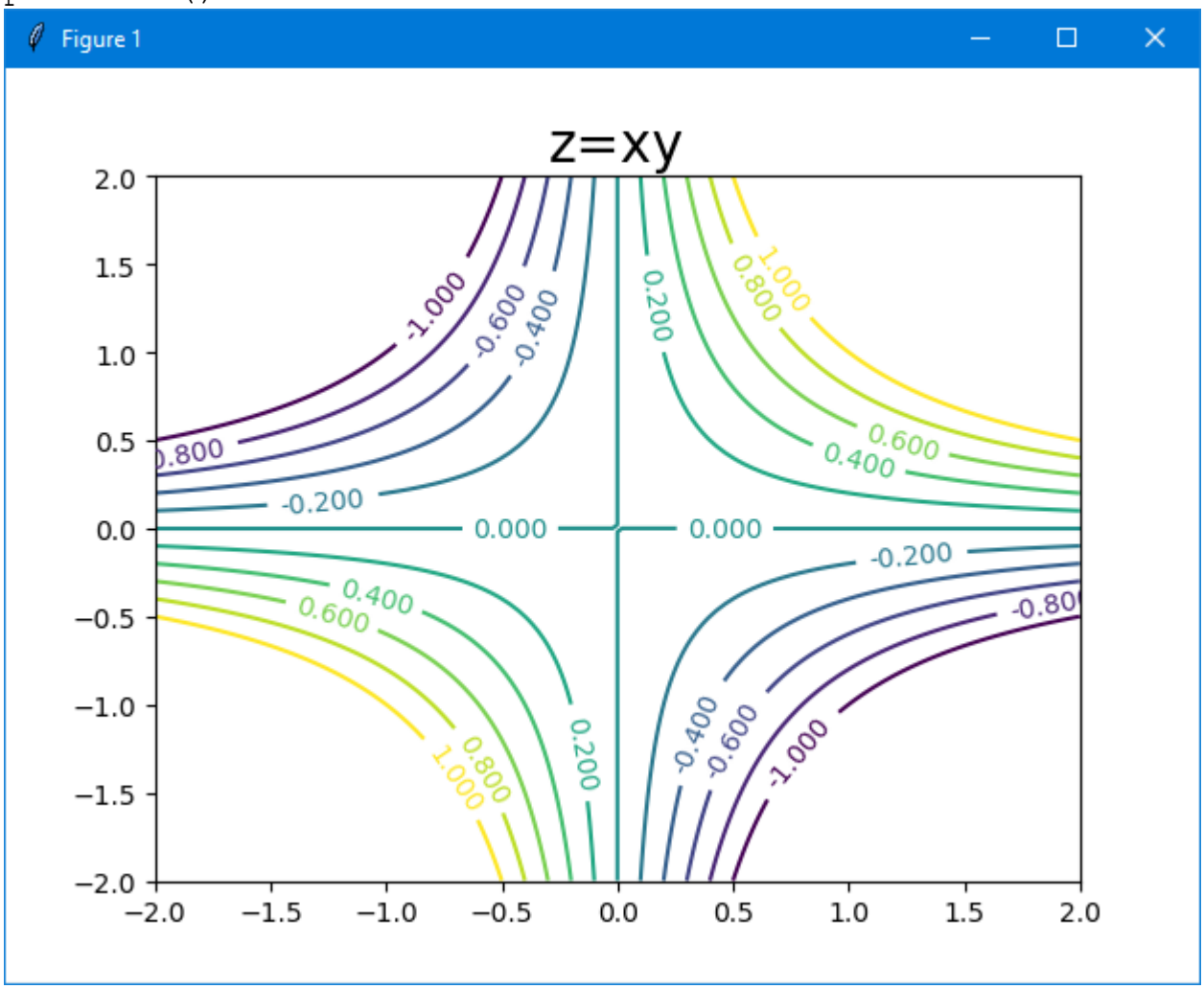


Роль функции **np.outer**(x,y) пакета numpy очевидна из примера:

```
>>> x=np.array([1,2,3])
>>> y=np.array([10,20,30])
>>> x*y
array([10, 40, 90])
>>> np.outer(x,y)
array([[10, 20, 30],
       [20, 40, 60],
       [30, 60, 90]])
```

Горизонтали можно раскрасить и подписать, а так же увеличить их число:

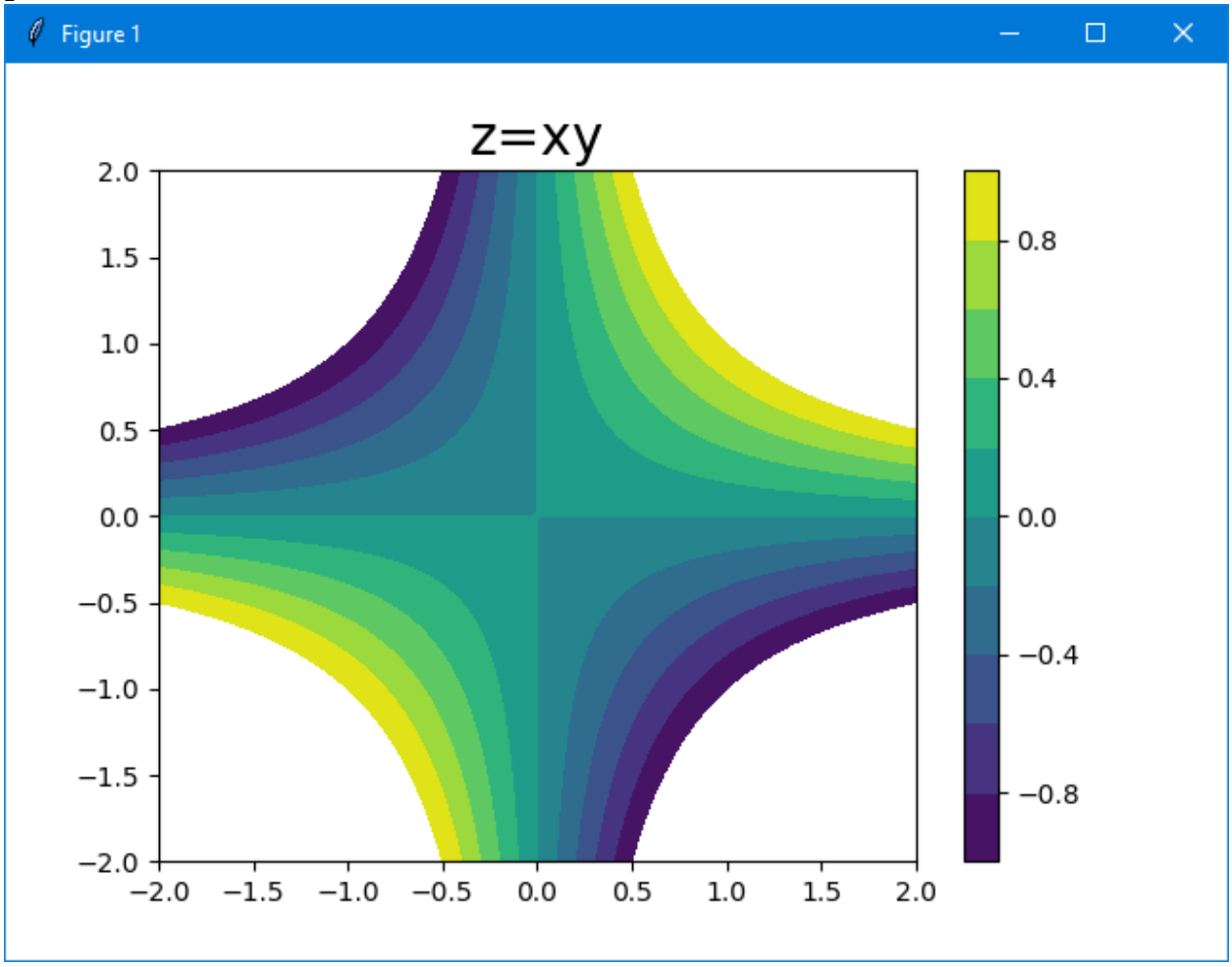
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
x=np.linspace(-2,2,100)
y=x
z=np.outer(x,y)
plt.title('z=xy',fontsize=20)
curves=plt.contour(x,y,z,np.linspace(-1,1,11))
plt.clabel(curves)
plt.show()
```



Высоту (значение Z) можно задать цветом, как на физических географических картах. `colorbar` показывает соответствие цветов и значений *z*.

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
x=np.linspace(-2,2,100)
y=x
```

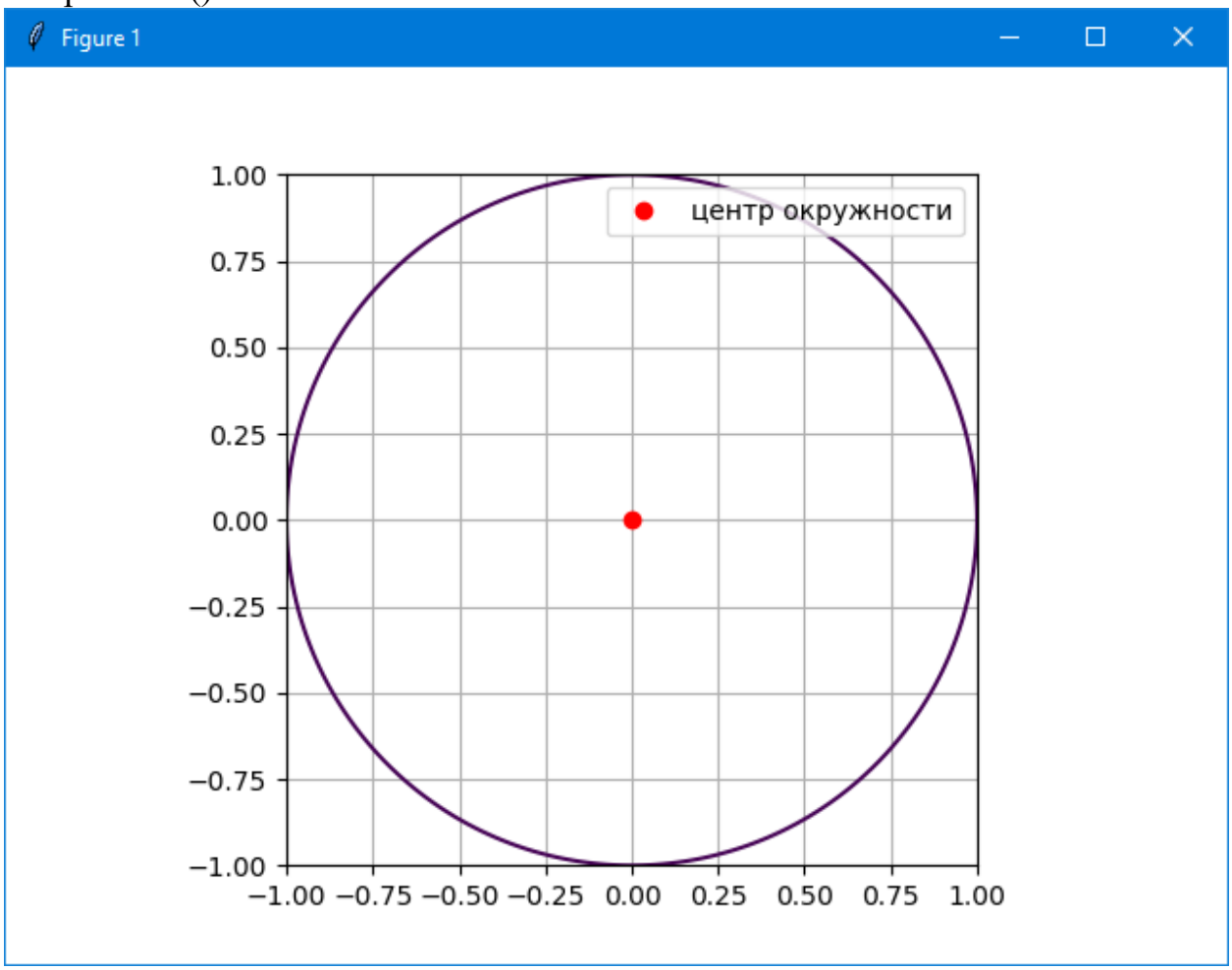
```
z=np.outer(x,y)
plt.title('z=xy',fontsize=20)
plt.contourf(x,y,z,np.linspace(-1,1,11))
plt.colorbar()
#plt.axes().set_aspect(1)
plt.show()
```



Так можно «рисовать» графики функций на плоскости, заданные «неявно» - F(x,y)=0:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1.0, 1.0, 100)
y = np.linspace(-1.0, 1.0, 100)
X, Y = np.meshgrid(x,y)
F = X**2 + Y**2 - 1  #0.6
plt.contour(X,Y,F,[0])
plt.plot([0],[0],'ro', label="центр окружности")
plt.gca().set_aspect('equal') #, чтобы рисунок выглядело кругом

# Включаем сетку
plt.grid(True)
```

```
plt.legend(loc='best')
plt.show()
```

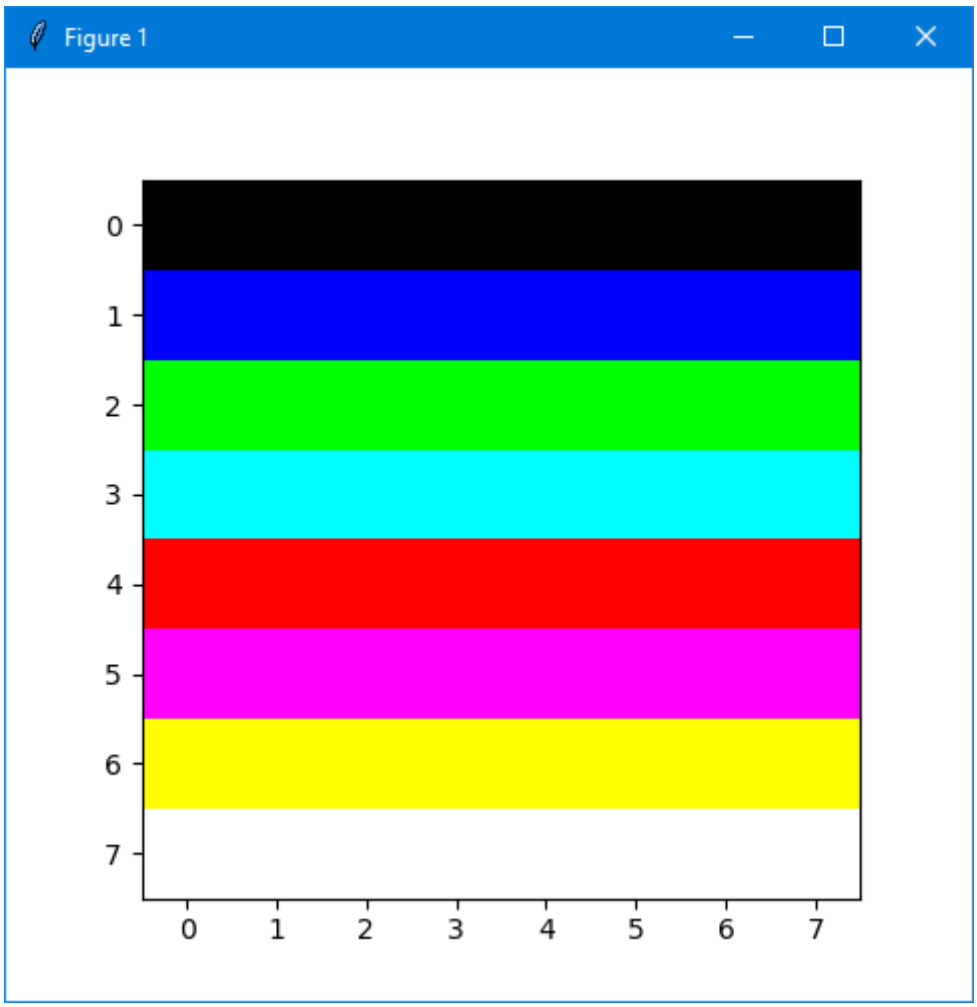


П19. Images (пиксельные картинки)

Картинка задаётся массивом `z`: `z[i,j]` - это цвет пикселя `i,j`, массив из 3 элементов (`r g b`, числа от 0 до 1). Для наглядности в примере формируются «натуральные» цвета в список col:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

n=8
u=np.linspace(0,1,n)
x,y=np.meshgrid(u,u)
z=np.zeros((n,n,3))

col=[]
for r in range(0,2):
    for g in range(0,2):
        for b in range(0,2):
```

```
            print(r,g,b)
            col.append([r,g,b])
#print (col)

for i in range(n):
    for j in range(n):     #  R G B
        z[i,j]=col[i]

plt.imshow(z)
plt.show()
```



П20. Трехмерный график

Все классы для работы с трехмерными графиками находятся в пакете `mpl_toolkits.mplot3d`, из которого нужно будет их импортировать.

Для того, чтобы нарисовать трехмерный график, в первую очередь надо создать трехмерные оси.

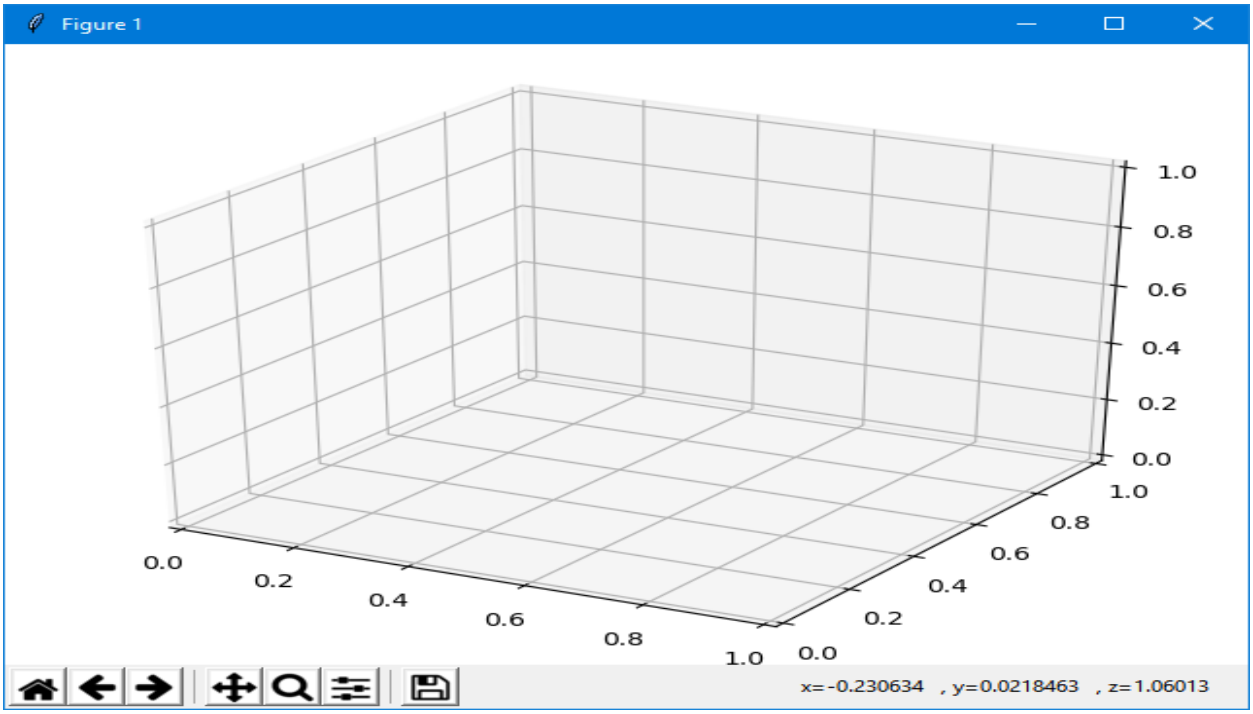
Чтобы их создать, нужно создать экземпляр класса *mpl_toolkits.mplot3d.Axes3D*. Его конструктор ожидает, как минимум, один параметр - экземпляр класса *matplotlib.figure.Figure*. Этот объект создается

вызовом *pylab.figure()*. У конструктора класса *matplotlib.figure.Figure* есть еще и другие необязательные параметры (по материалам сайта http://jenyay.net/Programming/Python3d).

Нарисуем пустые оси:

```
import numpy as np
import pylab
from mpl_toolkits.mplot3d import Axes3D

fig = pylab.figure()
Axes3D(fig)
pylab.show()
```

В результате увидим следующее окно:



Полученные оси можно вращать мышкой.

П21. Трёхмерная линия

Линия в пространстве задаётся параметрически: $x=x(t)$, , $y=y(t)$, $z=z(t)$. Например так:

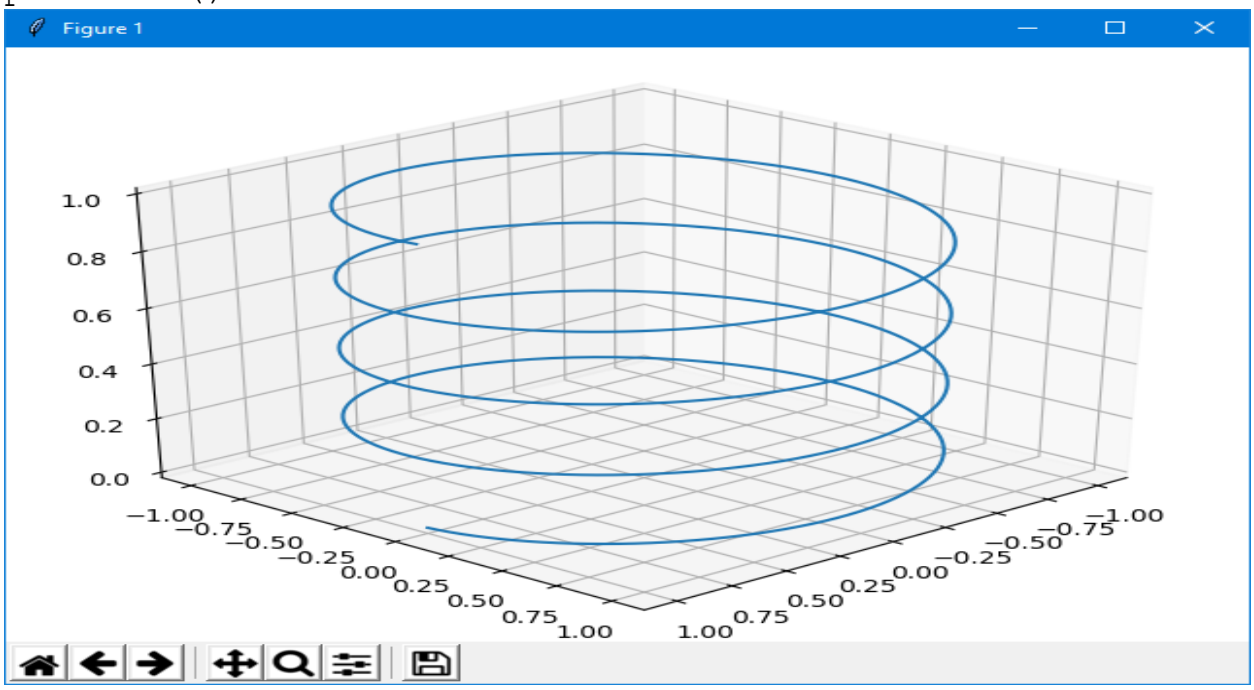
```
t=np.linspace(0, 4*np.pi,1000)
x=np.cos(2*t)
y=np.sin(2*t)
z=t /(4*np.pi)
```

Для построения и визуализации используется объект класса Axes3D из пакета `mpl_toolkits.mplot3d`:

```
import pylab
import mpl_toolkits.mplot3d as A3D
```

figure() - это текущий рисунок, создаём в нём объект ax, потом используем его методы для визуализации

```
import pylab
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
##from mpl_toolkits.mplot3d import Axes3D
import mpl_toolkits.mplot3d as A3D

t=np.linspace(0, 4*np.pi,1000)
x=np.cos(2*t)
y=np.sin(2*t)
z=t /(4*np.pi)

#Тут нужен объект класса Axes3D из пакета
mpl_toolkits.mplot3d.
# figure() - это текущий рисунок, создаём в нём объект
ax,
# потом используем его методы.
fig=pylab.figure()
ax=A3D.Axes3D(fig)
ax.elev,ax.azim=30,45 # задать, с какой стороны
смотрим.

ax.plot3D(x,y,z)
plt.show()
```



Построим три прямые по осям координат:

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
axes = axes3d.Axes3D(fig)

t = np.arange(-1, 1, 0.01)
x, y = np.meshgrid(t, t)

axes.plot3D(t, 0*t, 0*t, color='k' )
axes.plot3D(0*t, t, 0*t, color='k' )
axes.plot3D(0*t, 0*t, t, color='k' )

plt.show()
```

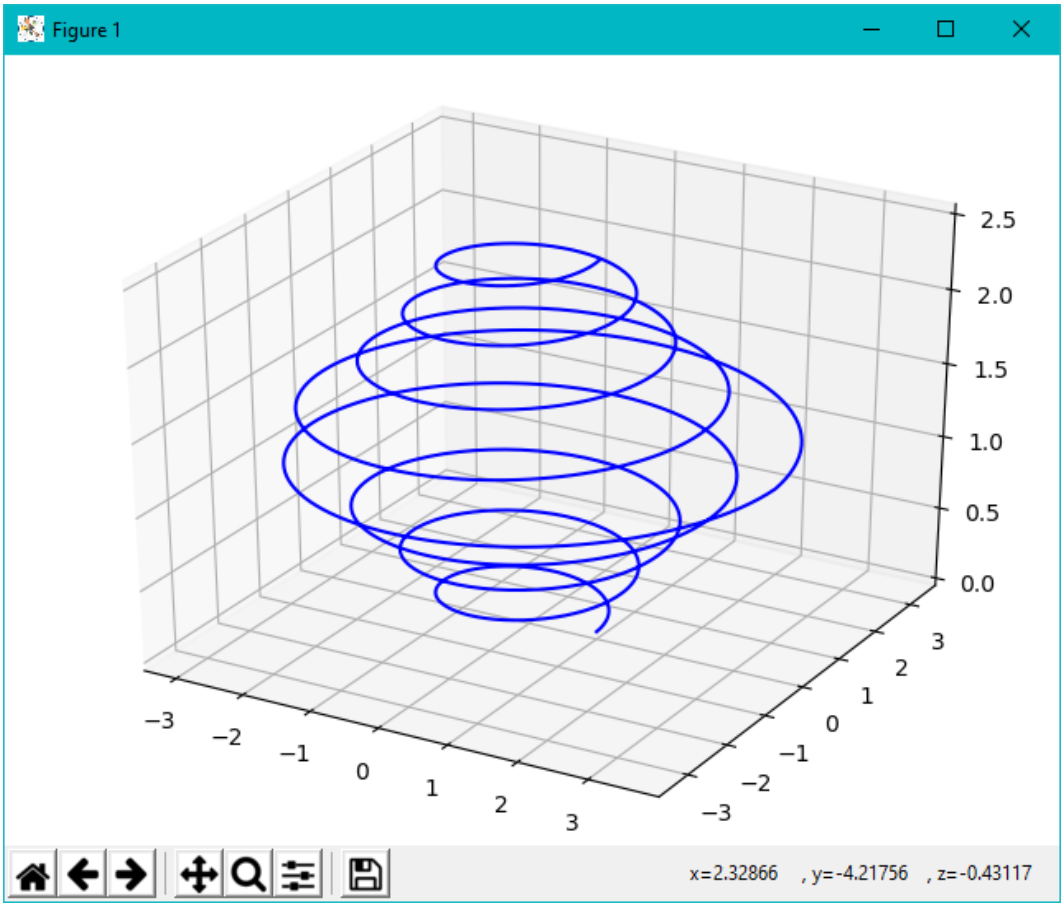
Построим линию, заданную уравнениями:

$$x(t)=\cos(2\cdot t)\cdot e^{\frac{a\cdot t}{10}} \qquad y(t)=\sin(2\cdot t)\cdot e^{\frac{a\cdot t}{10}} \qquad z(t)=\frac{t}{10}$$

$$t\in[0,8\cdot\pi] \qquad a=\begin{cases}-1, & npu \quad t\in[0,4\cdot\pi)\\ +1, & npu \quad t\in[4\cdot\pi,\ 8\cdot\pi]\end{cases}$$

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d as A3D

tt1=0; tt2=4*np.pi; tt3=8*np.pi; n1=500; n2=500
#-------
t1=np.linspace(tt1, tt2, n1)
x1=np.cos(2*t1) *np.exp(0.1*t1)
y1=np.sin(2*t1) *np.exp(0.1*t1)
z1=0.1*t1
#------
ttt=t1[500-1]
t2=np.linspace(tt2, tt3, n2)
x2=np.cos(2*(t2)) *np.exp(-0.1*(t2-ttt)+0.1*ttt) #+
x1[500-2]
y2=np.sin(2*(t2)) *np.exp(-0.1*(t2-ttt)+0.1*ttt) #+
y1[500-2]
z2=0.1*t2
#-----
#print("x=  ", x1[500-1], x2[0], t1[500-1], t2[0])
```

```
#print("y=  ", y1[500-1], y2[0], t1[500-1], t2[0])
#-----
x=np.concatenate((x1[:-1],x2))
y=np.concatenate((y1[:-1],y2))
z=np.concatenate((z1[:-1],z2))
#----
fig=plt.figure()
ax=A3D.Axes3D(fig)
ax.plot3D(x,y,z,'b-')
plt.show()
```



П22. Поверхности

Все поверхности задаются параметрические: ***x=x(u,v), , y=y(u,v), z=z(u,v).***

Если мы хотим задать поверхность «явно» *z=z(x,y)*, то удобно создать массивы *x=u* и *y=v* функцией **`meshgrid`**.

Для всех примеров будем использовать следующую функцию, от двух координат.

$$f(x,y)=\frac{\sin(x)\sin(y)}{xy}$$

Для начала нужно подготовить данные для рисования. Нам понадобятся *три двумерные матрицы*:

- матрицы *X* и *Y* будут хранить координаты сетки точек, в которых будет вычисляться приведенная выше функция,
- матрица *Z* будет хранить значения этой функции в соответствующей точке.

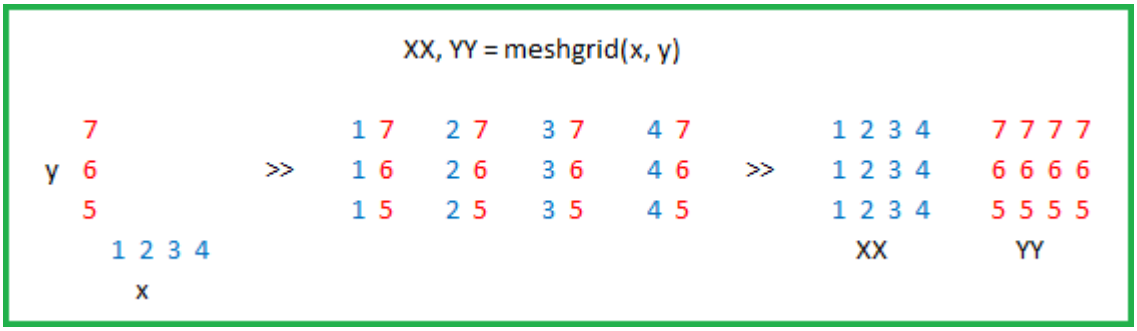
Если мы хотим нарисовать трехмерный график на эквидистантной сетке (на сетке, у которой расстояние между точками одинаковое), то для создания матриц, которые будут хранить координаты, будем использовать функцию **`meshgrid()`** из библиотеки `numpy`.

Эта функция создает двумерные матрицы сеток по одномерным массивам. Работа этой функции очень наглядно показана ниже:

```
>>> X, Y = numpy.meshgrid([1,2,3], [4,5,6,7])
>>> X
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
>>> Y
array([[4, 4, 4],
       [5, 5, 5],
       [6, 6, 6],
       [7, 7, 7]])
```

Теперь по индексу узла сетки можно узнать реальные координаты: X[0][0] = 1, Y[0][0] = 4 и т.п.

Фактически функция **`[X, Y] = meshgrid(x, y)`** задает сетку на плоскости x-y в виде двумерных массивов X, Y, которые определяются одномерными массивами x и y. Строки массива X являются копиями вектора x, а столбцы - копиями вектора y. Формирование таких массивов упрощает вычисление функций двух переменных, позволяя применять операции над массивами. Ниже приведем рисунок поясняющий построение массивов X и Y:



Чтобы отделить подготовку данных от самого рисования, создание сетки и расчет функции выделим в отдельную функцию:

```
def makeData ():
    # Строим сетку в интервале от -10 до 10
    # с шагом 0.1 по обоим координатам
    x = numpy.arange (-10, 10, 0.1)
    y = numpy.arange (-10, 10, 0.1)
    # Создаем двумерную матрицу-сетку
    xgrid, ygrid = numpy.meshgrid(x, y)
    # В узлах рассчитываем значение функции
    zgrid = numpy.sin (xgrid) * numpy.sin (ygrid) /
                                          (xgrid * ygrid)
    return xgrid, ygrid, zgrid
```

Эта функция возвращает три двумерные матрицы: x, y, z. Координаты x и y лежат в интервале от -10 до 10 с шагом 0.1.

Теперь возвращаемся непосредственно к рисованию. Чтобы отобразить наши данные, достаточно вызвать метод *plot_surface()* экземпляра класса *Axes3D*, в который передадим полученные с помощью функции *makeData()* двумерные матрицы.

Теперь наш пример выглядит следующим образом:

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
import numpy

def makeData ():
    x = numpy.arange (-10, 10, 0.1)
    y = numpy.arange (-10, 10, 0.1)
    xgrid, ygrid = numpy.meshgrid(x, y)
    zgrid = numpy.sin (xgrid) * numpy.sin (ygrid) /
                                          (xgrid * ygrid)
    return xgrid, ygrid, zgrid

x, y, z = makeData()
fig = pylab.figure()
axes = Axes3D(fig)
axes.plot_surface(x, y, z)
pylab.show()
```

В новых версиях matplotlib вместо

```
axes = Axes3D(fig)
```

рекомендуется кодировать:

```
fig = pylab.figure()
axes = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(axes)
```

Так, например, можно «подписать» оси координат:

```
axes.set_xlabel("-- x -->")
axes.set_ylabel("-- y -->")
```

```
axes.set_zlabel("-- z -->")
```

Если мы запустим этот скрипт, то появится окно:

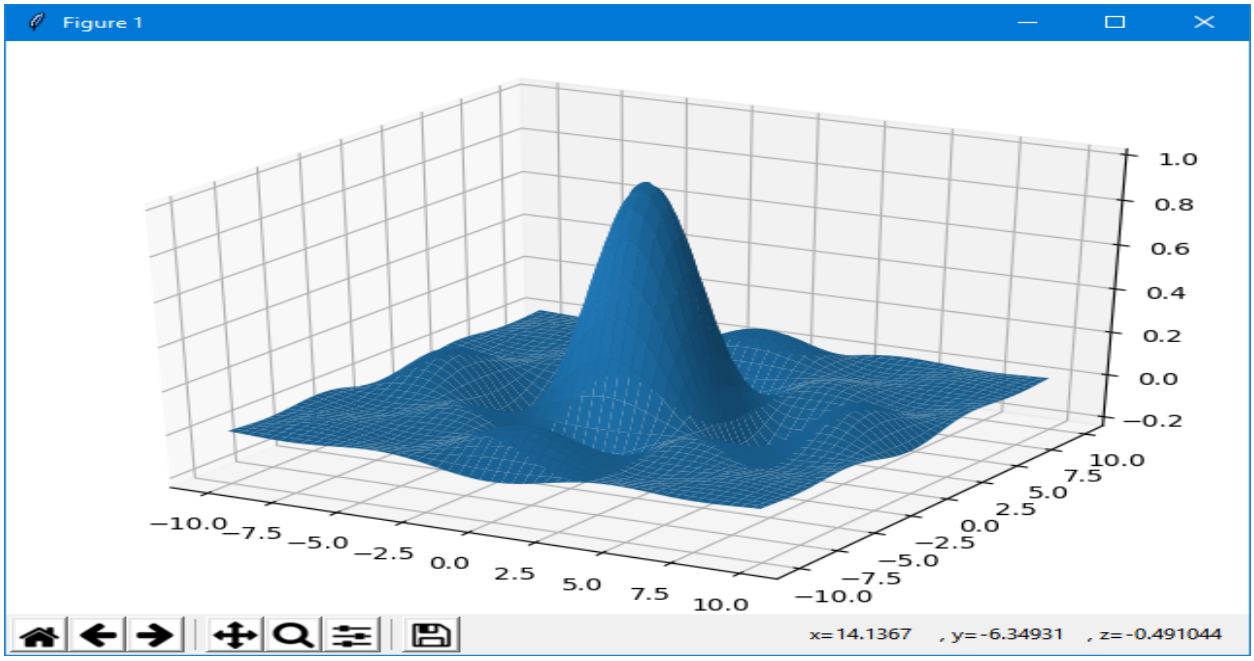


Рисунок можно вращать с помощью мышки. Matplotlib не использует графический ускоритель, поэтому вращение происходит довольно медленно, хотя скорость зависит от количества точек на поверхности.

Рассмотрим остальные параметры метода *Axes3D.plot_surface(X, Y, Z, *args, **kwargs)*, их не так много:

- *X*, *Y* и *Z* - эти параметры задают сетку и значение функции в узлах.
- *rstride* и *cstride* задают шаг вывода графика. Чем меньше шаг, тем точнее отображается функция, но тем дольше происходит рисование.
- *color* - задает цвет графика
- *cmap* - задает градиент цветов, чтобы цвет ячейки графика зависел от значения функции в этой области.

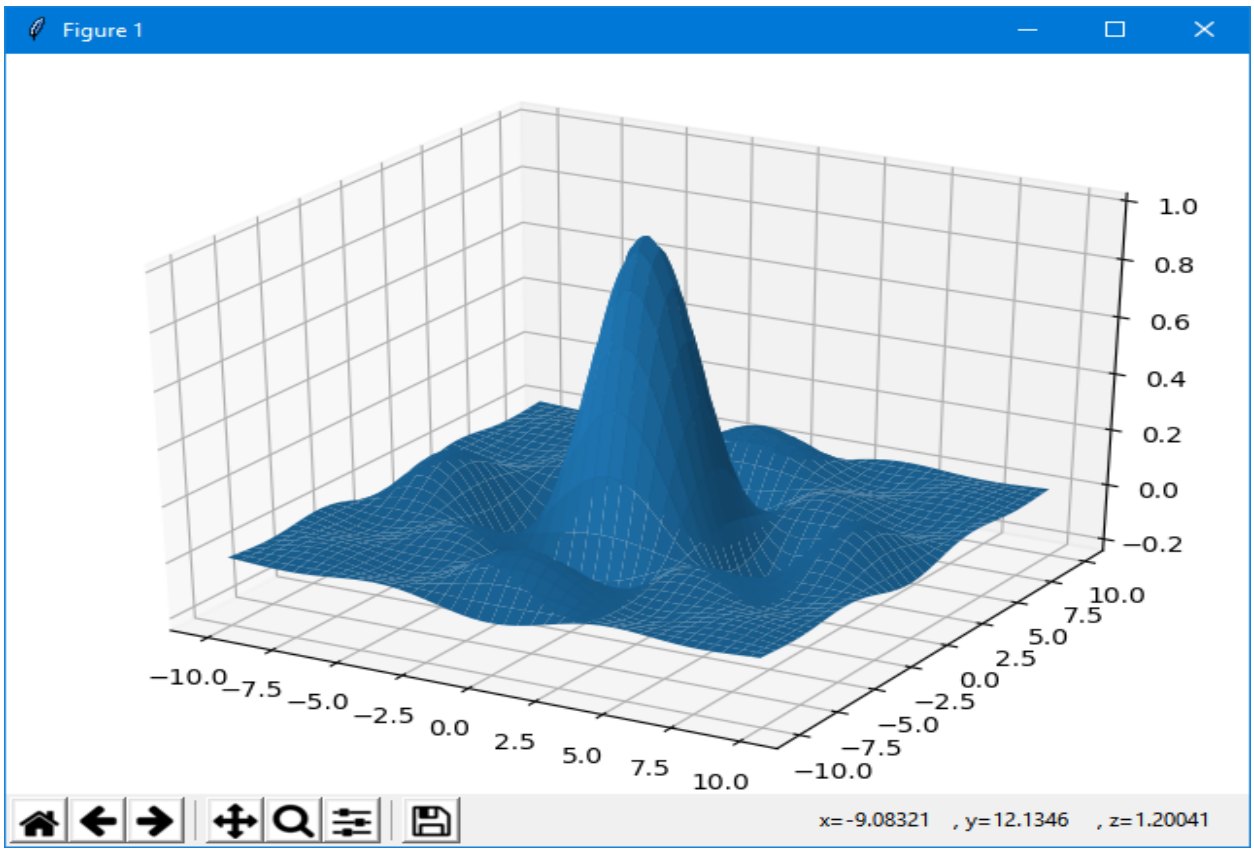
Рассмотрим эти параметры.

Шаг сетки

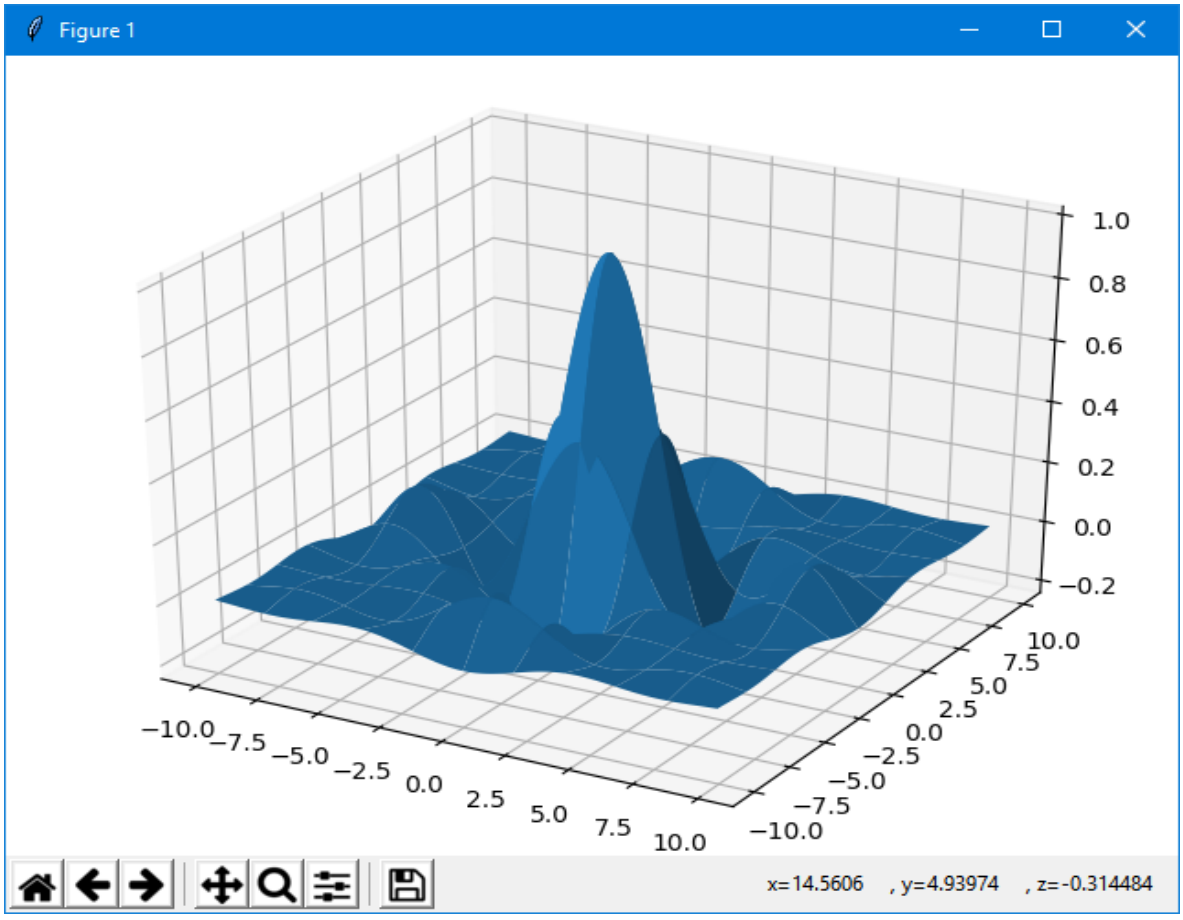
Сначала посмотрим как влияют на внешний вид параметры *rstride* и *cstride*. Изменим в предыдущем примере строку с использованием метода *plot_surface()* следующим образом:

```
axes.plot_surface(x, y, z, rstride=5, cstride=5)
```

В результате мы получим более мелкую сетку на графике:



Если таким же образом установить значения этих параметров в 20, то сетка будет наоборот более крупная:



Изменение цвета

Теперь изменим цвет поверхности с помощью параметра *color*. Этот параметр представляет собой строку, которая описывает цвет. Строка цвета может задаваться разными способами:

Цвет можно определить английским словом для соответствующего цвета или одной буквой. Таких цветов не много:

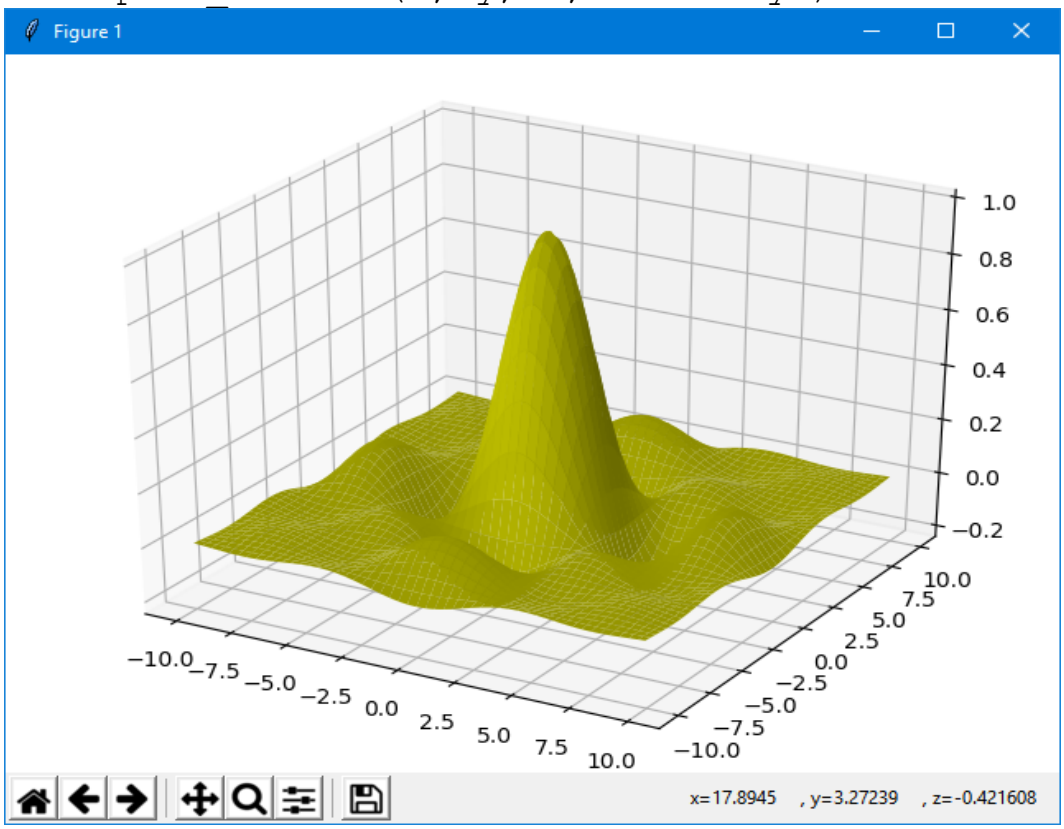
- 'b' или 'blue'
- 'g' или 'green'
- 'r' или 'red'
- 'c' или 'cyan'
- 'm' или 'magenta'
- 'y' или 'yellow'
- 'k' или 'black'
- 'w' или 'white'

Для примера сделаем поверхность желтой:

```
axes.plot_surface(x, y, z, color='yellow')
```

или

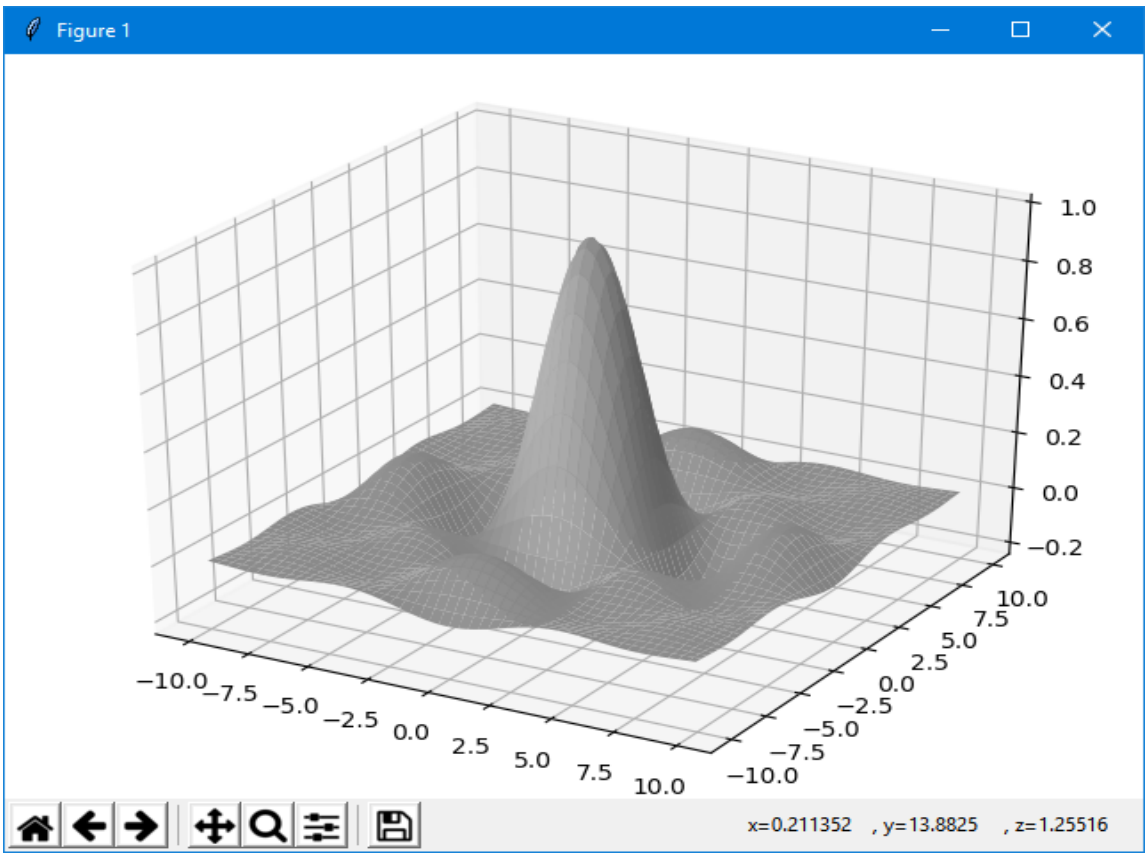
```
axes.plot_surface(x, y, z, color='y')
```



Если нам нужен серый цвет, то его яркость можем задать с помощью строки, содержащей число в интервале от 0.0 до 1.0 (0 - белый, 1 - черный). Например, можно написать следующую строку:

```
axes.plot_surface(x, y, z, color='0.7')
```

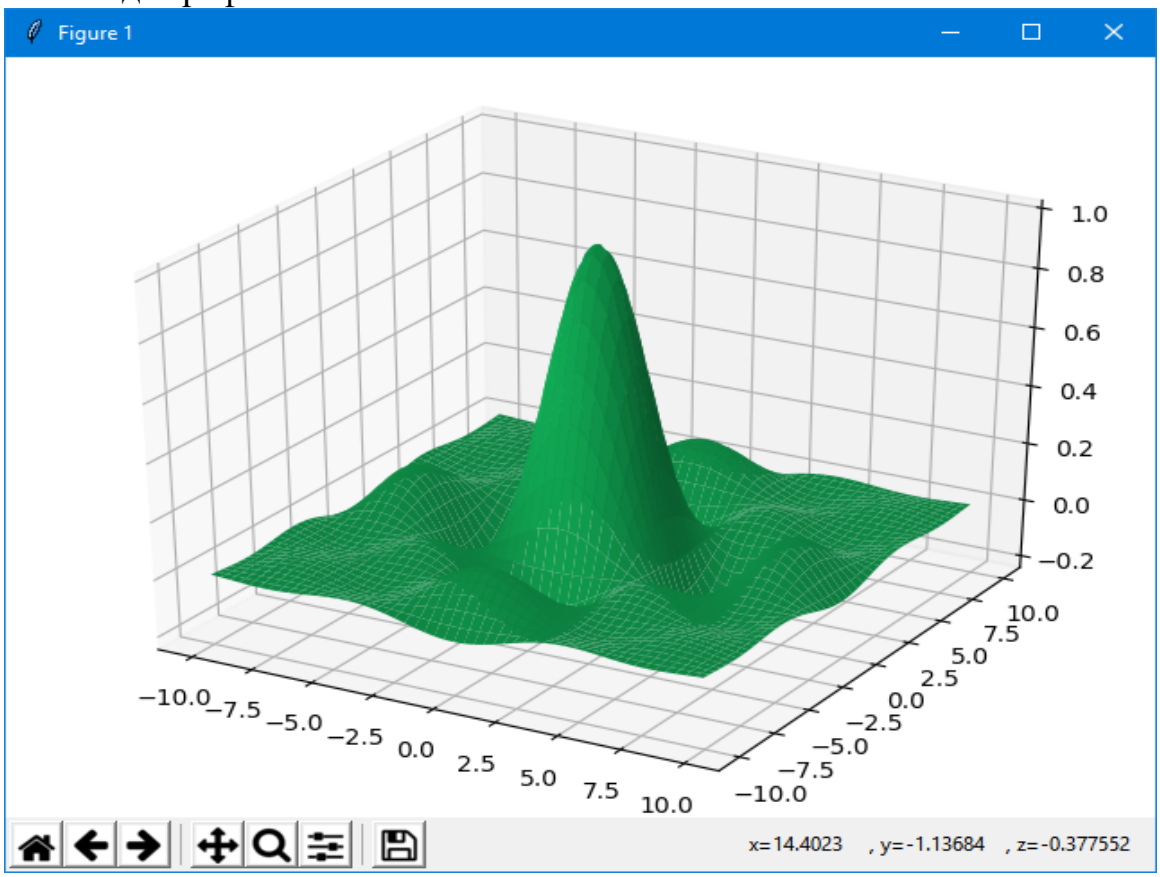
В этом случае мы увидим такой вот серый график:



Кроме того мы можем задавать цвет так как это принято в HTML после символа решетки ('#'). Например, можем задать цвет следующим образом:

```
axes.plot_surface(x, y, z, color='#11aa55')
```

Тогда график позеленеет:



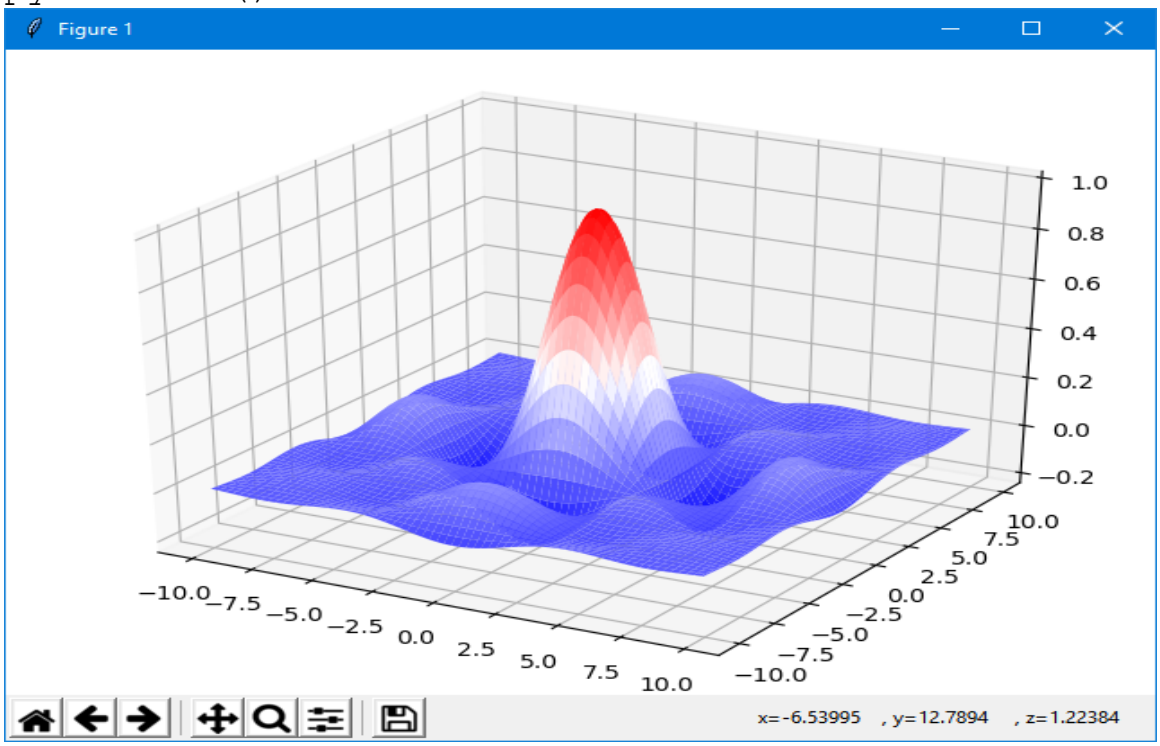
Использование цветовых карт (colormap)

Цветовые карты используются, если нужно указать в какие цвета должны окрашиваться участки трехмерной поверхности в зависимости от значения Z в этой области (задание цветового градиента).

Чтобы при выводе графика использовался градиент, в качестве значения параметра *cmap* (от слова colormap, цветовая карта) нужно передать экземпляр класса *matplotlib.colors.Colormap* или производного от него.

Следующий пример использует класс *LinearSegmentedColormap*, производный от *Colormap*, чтобы создать градиент перехода от синего цвета к красному через белый.

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LinearSegmentedColormap
import numpy
def makeData ():
    x = numpy.arange (-10, 10, 0.1)
    y = numpy.arange (-10, 10, 0.1)
    xgrid, ygrid = numpy.meshgrid(x, y)
    zgrid = numpy.sin (xgrid) * numpy.sin (ygrid) /
            (xgrid * ygrid)
    return xgrid, ygrid, zgrid
x, y, z = makeData()
fig = pylab.figure()
axes = Axes3D(fig)
axes.plot_surface(x, y, z, rstride=3, cstride=3,  \
cmap = LinearSegmentedColormap.from_list ("red_blue",
                                  ['b', 'w', 'r'], 256))
pylab.show()
```

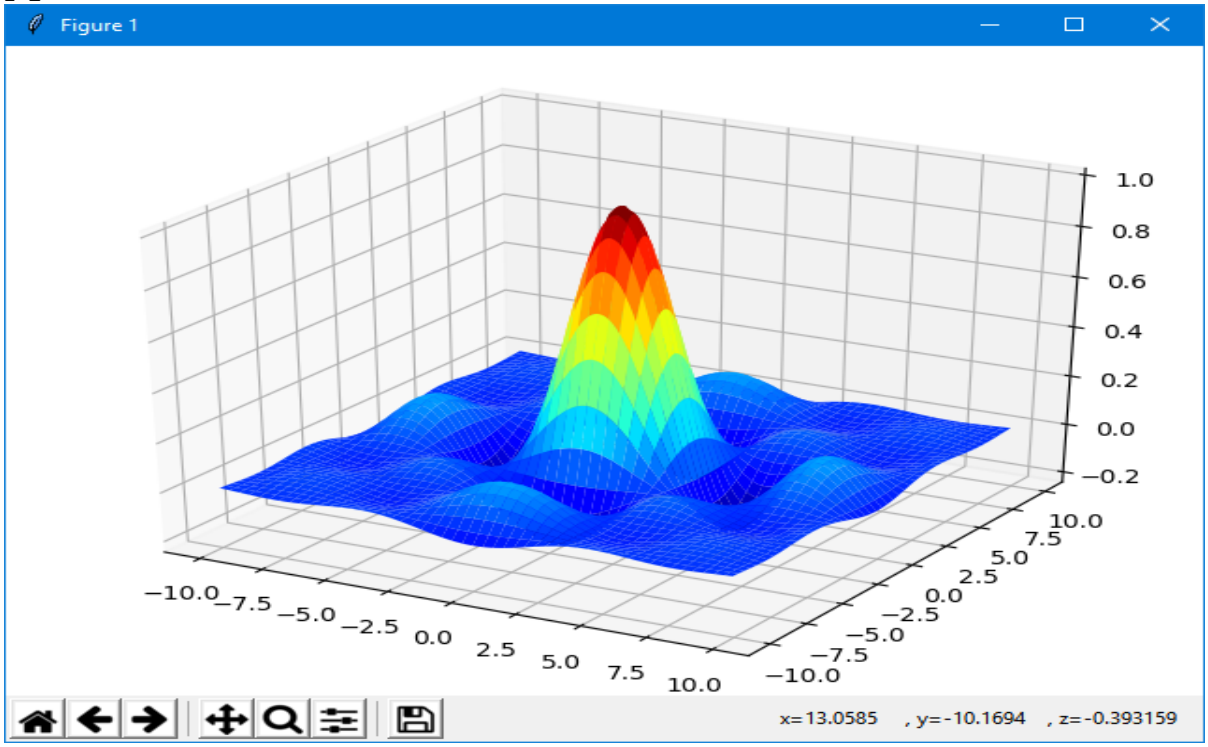


Здесь используется статический метод `from_list()`, который принимает три параметра:

- Имя создаваемой карты
- Список цветов, начиная с цвета для минимального значения на графике (голубой - 'b'), через промежуточные цвета (у нас это белый - 'w') к цвету для максимального значения функции (красный - 'r').
- Количество цветовых переходов. Чем это число больше, тем более плавный градиент, но тем больше памяти он занимает.

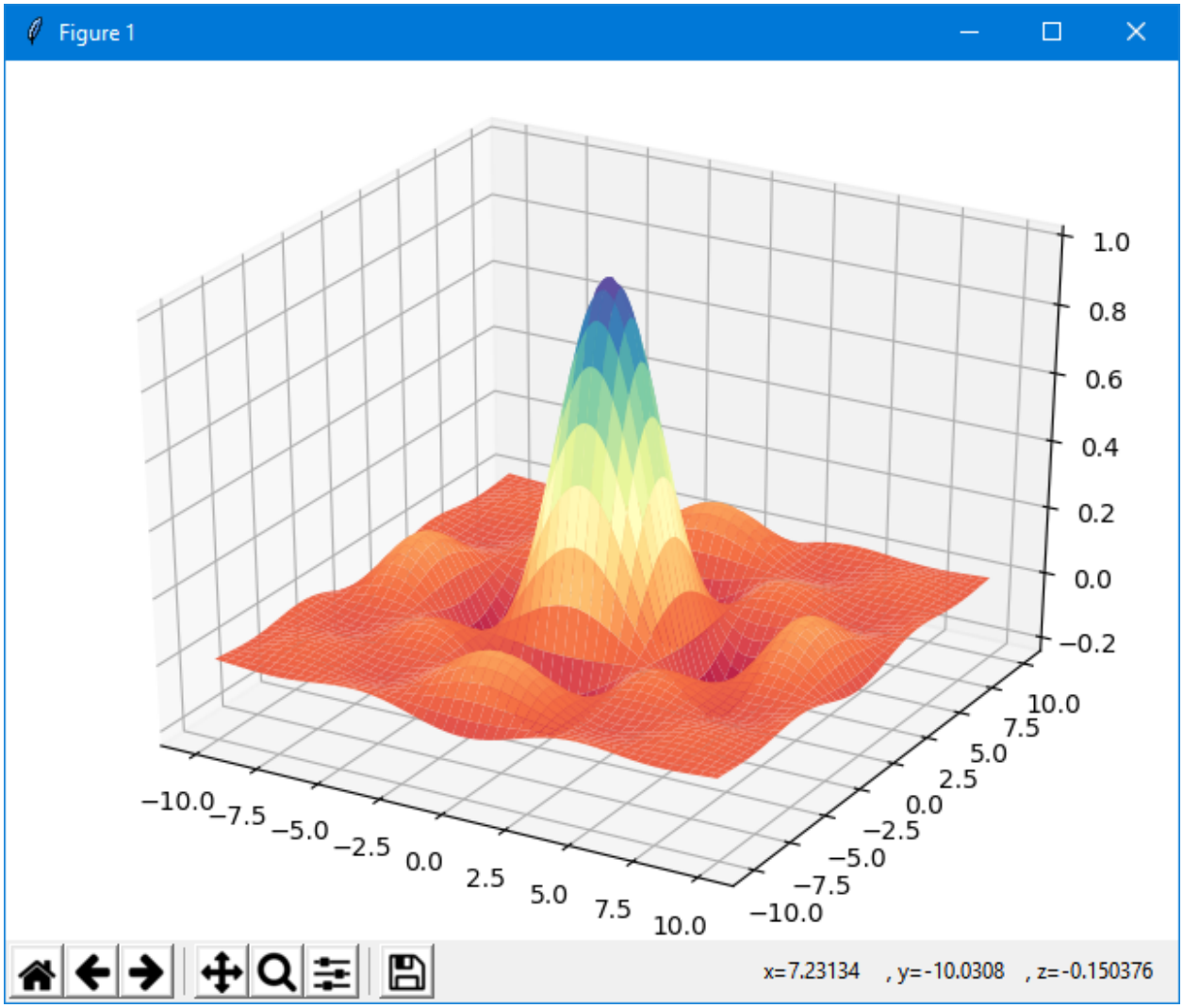
Карта ***cm.jet*** - это, наверное, самая часто используемая карта в примерах.

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import cm
import numpy
def makeData():
    x = numpy.arange (-10, 10, 0.1)
    y = numpy.arange (-10, 10, 0.1)
    xgrid, ygrid = numpy.meshgrid(x, y)
    zgrid = numpy.sin (xgrid) * numpy.sin (ygrid) /
(xgrid * ygrid)
    return xgrid, ygrid, zgrid
x, y, z = makeData()
fig = pylab.figure()
axes = Axes3D(fig)
axes.plot_surface(x, y, z, rstride=4, cstride=4, cmap =
cm.jet)
pylab.show()
```



Обратите внимание, что в качестве аргумента *cmap* мы передаем не строку, а имя переменной из модуля *cm*, причем это уже созданный экземпляр класса, а не имя класса. Так, например, *cm.jet* - это экземпляр класса *matplotlib.colors.LinearSegmentedColormap*.

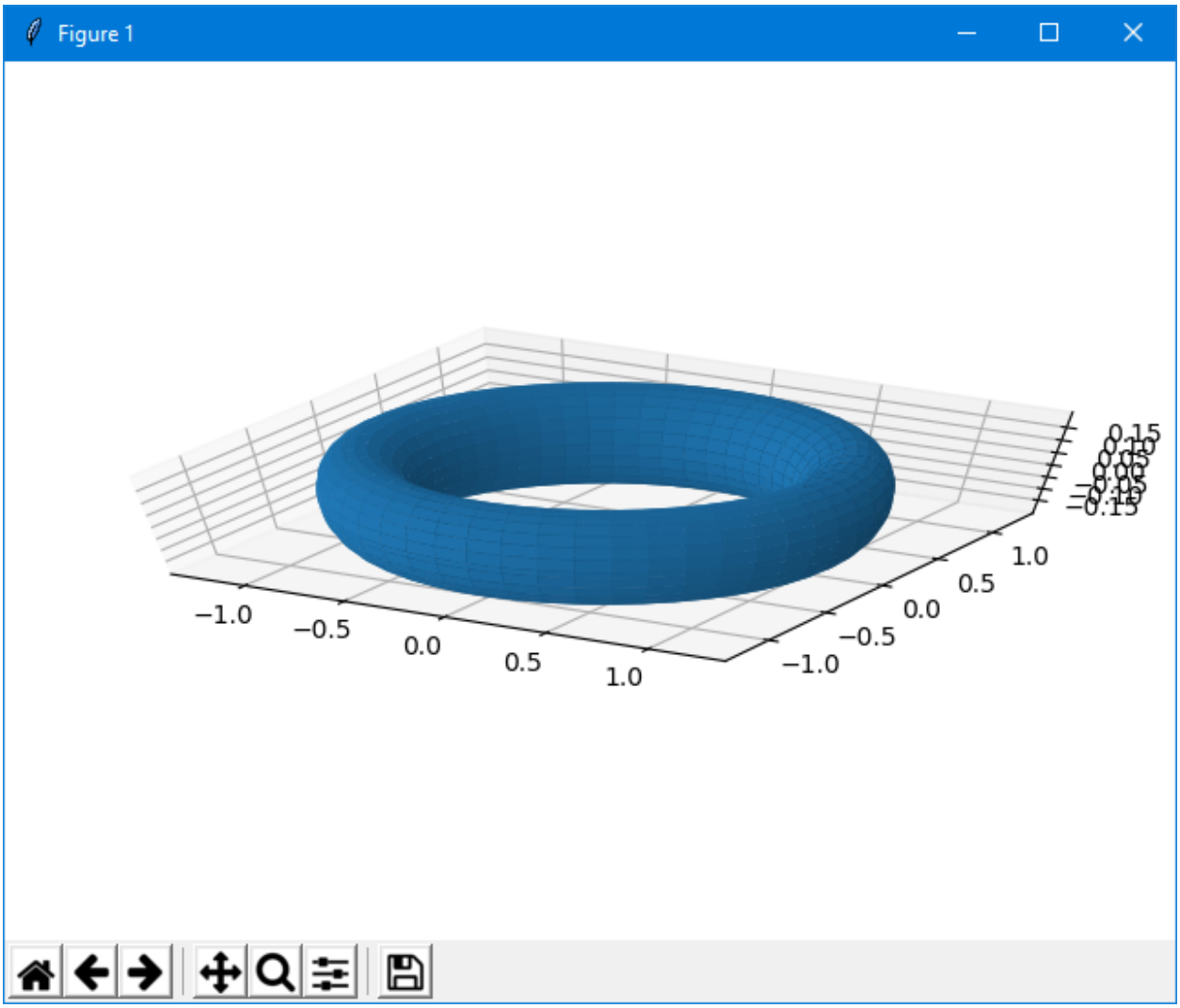
Для примера цветовая карта *cm.Spectral* выглядит следующим образом:



П23. Параметрические поверхности с параметрами ϑ φ

```
import pylab
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import cm
import numpy as np

t=np.linspace(0,2*np.pi,50)
th,ph=np.meshgrid(t,t)
r=0.2
```

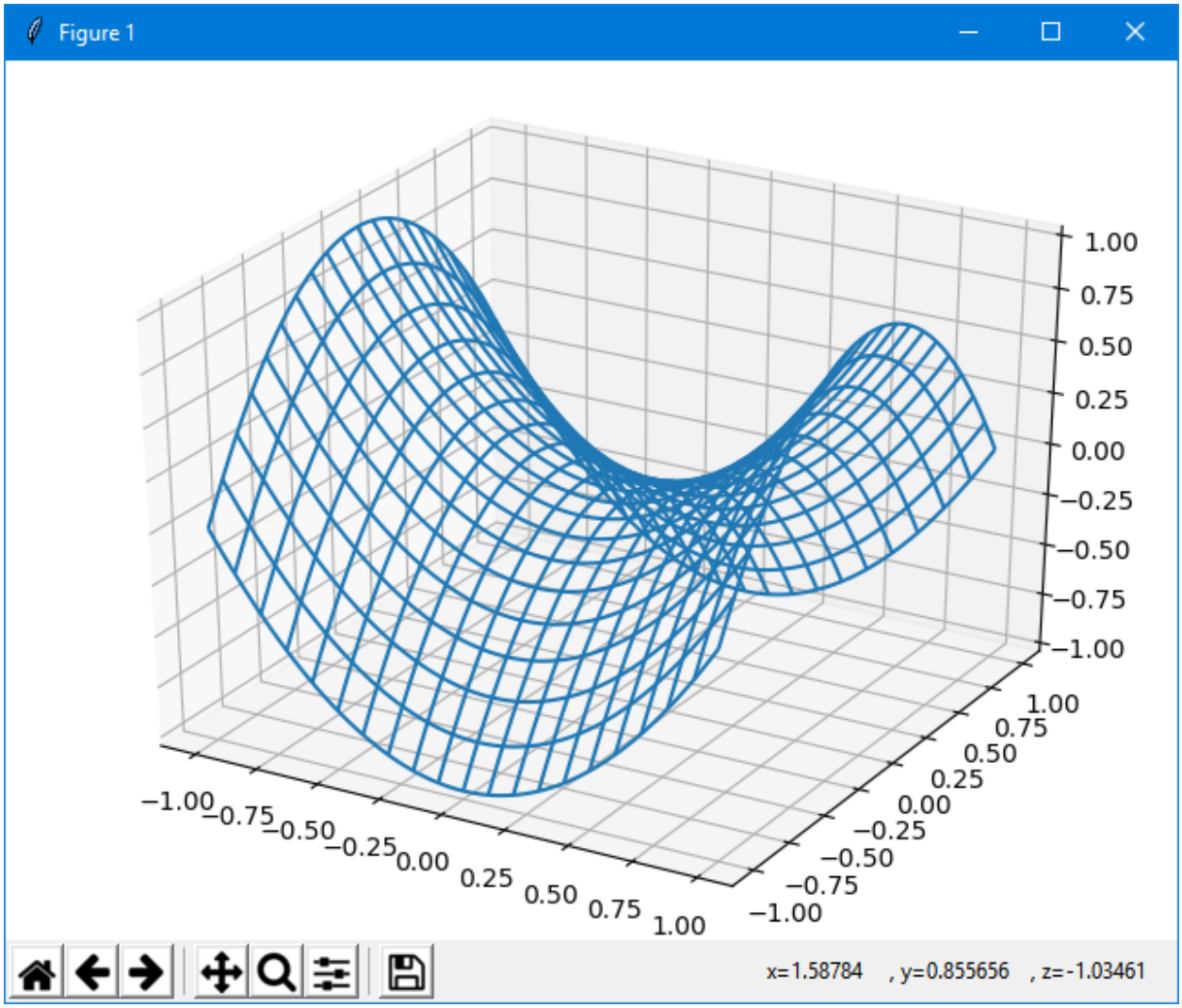
```
x,y,z=(1+r*np.cos(ph))*np.cos(th),(1+r*np.cos(ph))*np.s
in(th),r*np.sin(ph)
fig=pylab.figure()
ax=Axes3D(fig)
ax.elev=60
ax.set_aspect(0.3)
ax.plot_surface(x,y,z,rstride=2,cstride=1)
pylab.show()
```



П24. Построение графика «сетки» функции двух переменных

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
ax = axes3d.Axes3D(plt.figure())
i = np.arange(-1, 1, 0.01)
X, Y = np.meshgrid(i, i)
Z = X**2-Y**2
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
```

```
plt.show()
```



Для примера построит три плоскости паралельные координатным плоскостям

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
axes = axes3d.Axes3D(fig)
t = np.arange(-1, 1, 0.01)
x, y = np.meshgrid(t, t)
axes.plot_wireframe(0*x, y, x, rstride=5, cstride=5,
                                                color='r')
axes.plot_wireframe(x, 0*y, y, rstride=5, cstride=5,
                                                  color='b' )
axes.plot_wireframe(x, y, 0*x, rstride=5, cstride=5,
                                                  color='g')
plt.show()
```

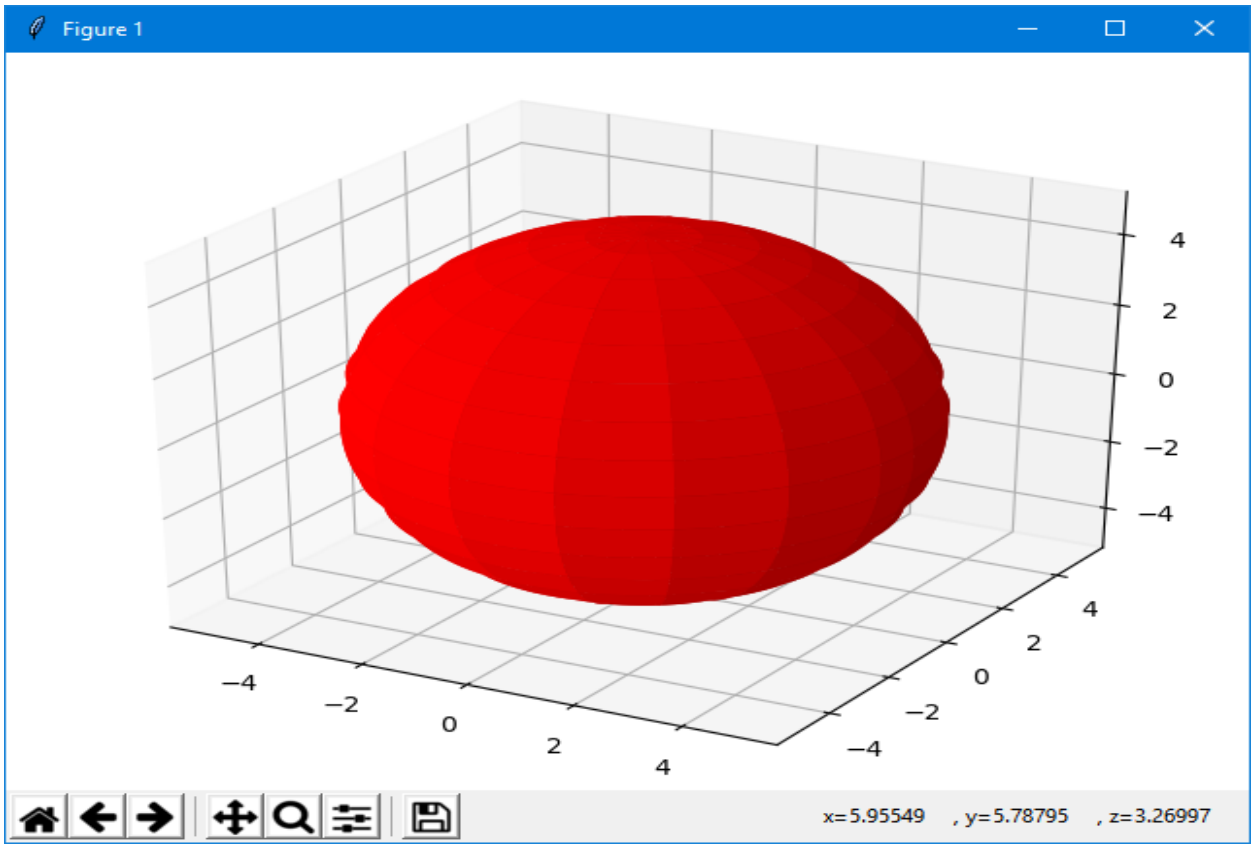
П25 Построение графика функции двух переменных $x=x(u,v)$, $y=y(u,v)$, $z=z(u,v)$.

Поверхность задается параметрически: $x=x(u,v)$, , $y=y(u,v)$, $z=z(u,v)$.

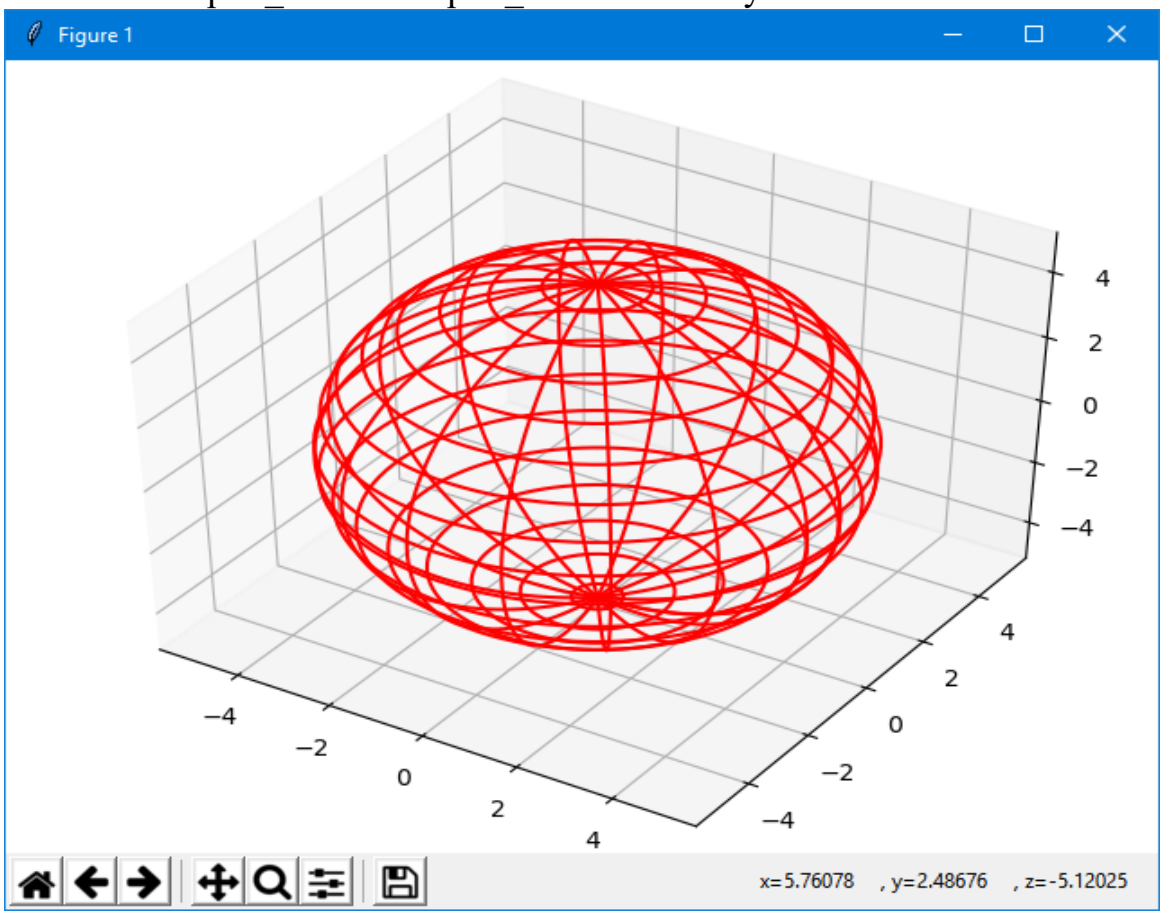
Классическое уравнение сферы, заданное параметрически:

$$x(u,v)=\cos(u)\cdot\cos(v)$$
$$y(u,v)=\sin(u)\cdot\cos(v)$$
$$z(u,v)=\sin(v)$$
$$u\in[-\pi,+\pi]$$
$$v\in[-2\pi,+2\pi]$$

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
ax = axes3d.Axes3D(plt.figure())
u = np.linspace(0, 2*np.pi, 100)
v = np.linspace(0, np.pi, 100)
x =5*np.outer(np.cos(u), np.sin(v))
y =5*np.outer(np.sin(u), np.sin(v))
z =5*np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(x, y, z,  rstride=6, cstride=6,
                                          color='r')
plt.show()
```



Заменив plot_surface на plot_wireframe получим:



В программе использовалась функция модуля ***numpy.outer***, которая вычисляет внешнее (прямое или тензорное) произведение двух векторов:

numpy.outer(a, b, out=None)

Для двух векторов *a* длинной *N* и *b* длинной *M* внешнее произведение определяется следующим правилом:

```
[[a0*b0, a0*b1, ..., a0*bN],
 [a1*b0, a1*b1, ..., a1*bN],
 ...
 [aM*b1, aM*b1, ..., aM*bN]]
```

Данное правило может быть обобщено на массивы с большей размерностью. Но данная функция сжимает все многомерные массивы до одной оси.

Параметры: **a, b** - числа, массивы NumPy или подобные массивам объекты. Одномерные массивы, необязательно одинаковой длины. Двумерные и многомерные массивы сжимаются до одной оси. **out** - массив NumPy, необязательный параметр.

Массив в который можно поместить результат функции. Данный массив должен соответствовать форме и типу данных результирующего массива функции, а так же обязательно быть *C*-смежным, т.е. хранить данные в строчном *C* стиле. Указание данного параметра, позволяет избежать лишней

операции присваивания тем самым немного ускоряя работу вашего кода. Полезный параметр если вы очень часто обращаетесь к функции в цикле.

Возвращает: результат - двумерный массив NumPy являющимся внешним произведением двух векторов..

Функция **ones()** возвращает новый массив указанной формы и типа, заполненный единицами. А **np.ones(np.size(u))** - возвращает новый массив, размера **u** так же заполненный единицами.

В выше приведенном примере строилась так называемая «плотная» сетка 3-мерного координатного пространства на значениях сетки координат x, y, z соответственно. Для этого использовалалась функция прямого или тензорного произведение двух векторов значений u и v.

Можно поступить «по другому» - построить массив плотных координатных сеток 3-мерного координатного пространства для указанных в виде диапазонов одномерных массивов координатных векторов u и v. Затем «просто» применить формулы, задающие требуемые зависимости ***x=x(u,v), , y=y(u,v), z=z(u,v).***

Для построения массива плотных координатных сеток используем функцию **numpy.mgrid:**

```
numpy.mgrid[index object] =
        <numpy.lib.index_tricks.nd_grid object>
```

Функция **mgrid()** возвращает массив плотных координатных сеток N-мерного координатного пространства для указанных в виде диапазонов одномерных массивов координатных векторов.

Параметры: index object - объект индексации

Под объектом индексации понимается список из двух или трех элементов, например *[0:5]* или *[0:5:10j]*.

Если элементов в списке всего два, то это интерпретируется как полуоткрытый интервал *[start, ... , stop)*, в котором все элементы отличаются на 1, а значение *stop* в сам интервал не входит.

В качестве третьего элемента указывается мнимая часть комплексного числа, которое указывает на количество равномерно разнесенных элементов внутри закрытого интервала *[start, ... , stop]*, при этом значение *stop* попадает в интервал.

Возвращает: результат - массив NumPy, являющимся массивом плотных координатных сеток N-мерного координатного пространства, количество и размеры которых зависят от указанных диапазонов.

Построение выполним следующим скриптом:

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
```

```
axes = axes3d.Axes3D(fig)
u, v = np.mgrid[0:2*np.pi:100j, 0:2*np.pi:100j]
x = 5*np.cos(u)*np.sin(v)
y = 5*np.sin(u)*np.sin(v)
z = 5*np.cos(v)
#axes.plot_surface(x, y, z,rstride=6, cstride=6,
                                              color='r')
axes.plot_wireframe(x, y, z, rstride=5, cstride=5,
                                              color='r')
plt.show()
```

Для получения массива плотных координатных сеток можно так же воспользоваться (рассмотренной выше) функцией **np.meshgrid:**

```
u1=np.linspace(0,2*np.pi,100)
v1=np.linspace(0,2*np.pi,100)
u, v = np.meshgrid(u1,v1)
```

При необходимости можно весь рисунок «растянуть» в kg раз по горизонтали и в kv раз по вертикали. Для этого укажем параметр метода plt.figure():

```
kg=2; kv=1
fig = plt.figure(figsize=plt.figaspect(1/kg)*kv)
```

П26. «Скроллинг» по оси X

Matplotlib включает в себя очень небольшое количество элементов управления, которые в терминах Matplotlib называются виджетами, которые располагаются пакете *matplotlib.widgets*. В этом пакете содержатся также виджеты для взаимодействия с графиками (виджеты для выделения областей) – подробнее см., например, http://jenyay.net/Matplotlib/Widgets.

Используя виджет Slider можно «растянуть» ось X:

```
from matplotlib.widgets import Slider
import math
import matplotlib.pyplot as plt
# подробнее см. например
# http://jenyay.net/Matplotlib/Widgets
fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25)
t = [i / 100. for i in range(0, int(math.pi) * 100, 1)]
s = [math.sin(i * 20) for i in t]
l, = plt.plot(t, s, lw=2, color='red')

plt.axis([0, 1, -10, 10])
plt.grid(True)

axcolor = 'lightgoldenrodyellow' #'gray'
```

```
ax_x_pos = plt.axes([0.25, 0.1, 0.65, 0.03],
facecolor=axcolor )
wsize = 10
x_pos = Slider(ax_x_pos, 'Position', 0,
    l            en(t) - wsize - 1, valfmt='%d', \
                                valinit=0,  color="g")

ax.set_xlim(t[0], t[wsize])
ax.set_ylim(-1.1, 1.1)

def update(val):
    #print(x_pos.val)
    pos = int(x_pos.val)
    ax.set_xlim(t[pos], t[pos + wsize])
    fig.canvas.draw_idle()

x_pos.on_changed(update)

plt.show()
```

Результат работы скрипта:

П27. «Динамические» графики

Без комментариев приведем пример «динамического» построения графика

см https://matplotlib.org/2.0.0/examples/animation/animate_decay.html:

```
'''
This example showcases a sinusoidal decay animation.
```

```
'''
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def data_gen(t=0):
    cnt = 0
    while cnt < 1000:
        cnt += 1
        t += 0.1
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)

def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []

def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)
    return line,

ani = animation.FuncAnimation(fig,
            run, data_gen, blit=False, interval=10,
                   repeat=False, init_func=init)
plt.show()
```

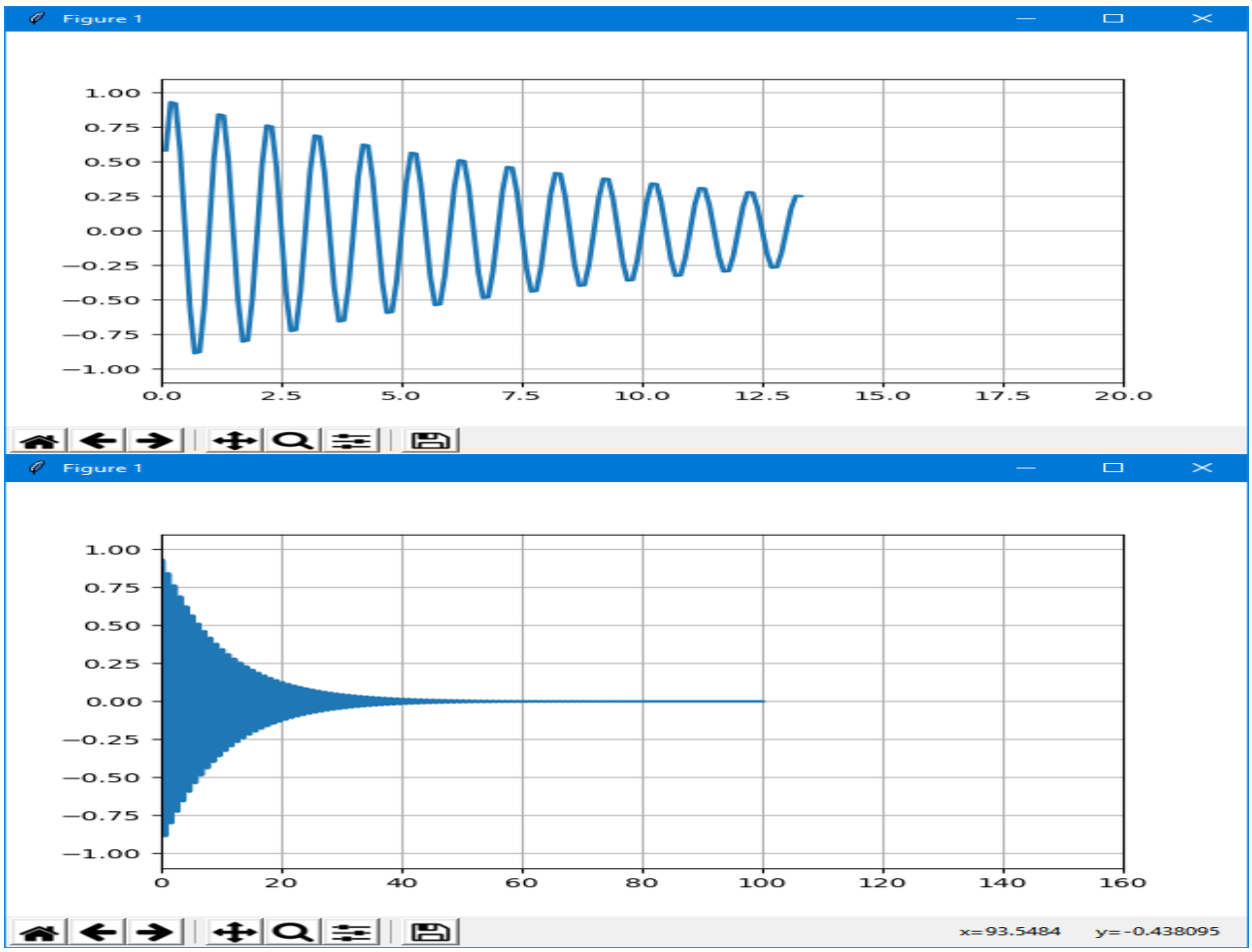


Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20
40
60
80
100
120
140
160
x=93.5484
y=-0.438095

III. Модуль math

Встроенный модуль math в Python предоставляет набор функций для выполнения математических, тригонометрических и логарифмических операций. Некоторые из основных функций модуля:

- **pow(num, power)**: возведение числа num в степень power
- **sqrt(num):** квадратный корень числа num
- **ceil(num):** округление числа до ближайшего наибольшего целого
- **floor(num):** округление числа до ближайшего наименьшего целого
- **factorial(num):** факториал числа
- **degrees(rad):** перевод из радиан в градусы
- **radians(grad):** перевод из градусов в радианы
- **cos(rad**): косинус угла в радианах
- **sin(rad):** синус угла в радианах
- **tan(rad):** тангенс угла в радианах
- **acos(rad):** арккосинус угла в радианах
- **asin(rad):** арксинус угла в радианах
- **atan(rad):** арктангенс угла в радианах
- **log(n, base):** логарифм числа n по основанию base
- **log10(n):** десятичный логарифм числа n

Пример применения некоторых функций:

```
import math

# возведение числа 2 в степень 3
n1 = math.pow(2, 3)
print(n1)  # 8

# ту же самую операцию можно выполнить так
n2 = 2**3
print(n2)

# возведение в квадрат
print(math.sqrt(9))  # 3

# ближайшее наибольшее целое число
print(math.ceil(4.56))  # 5

# ближайшее наименьшее целое число
print(math.floor(4.56))  # 4

# перевод из радиан в градусы
```

```
print(math.degrees(3.14159))  # 180

# перевод из градусов в радианы
print(math.radians(180))   # 3.1415.....
# косинус
print(math.cos(math.radians(60)))  # 0.5
# синус
print(math.sin(math.radians(90)))   # 1.0
# тангенс
print(math.tan(math.radians(0)))    # 0.0

print(math.log(8,2))    # 3.0
print(math.log10(100))    # 2.0
```

Также модуль `math` предоставляет ряд встроенных констант, такие как, PI и E:

```
import math
radius = 30
# площадь круга с радиусом 30
area = math.pi * math.pow(radius, 2)
print(area)

# натуральный логарифм числа 10
number = math.log(10, math.e)
print(number)
```

IV. Модуль ractions - рациональные числа (обыкновенные дроби)

Модуль `fractions` позволяет выполнять арифметические действия над рациональными числами, что в некоторых ситуациях бывает чрезвычайно удобно.

Создание обыкновенных дробей

Способов создания экземпляров рациональных чисел, довольно много поэтому мы разделим их на группы.

Самый простой способ создать обыкновенную дробь – указать числитель (`numerator`) и знаменатель (`denominator`):

```
>>> from fractions import Fraction
>>>
>>> Fraction()    #  по умолчанию numerator=0,
denominator=1
Fraction(0, 1)
```

```
>>>
>>> Fraction(numerator=1, denominator=2)     #
равносильно Fraction(1, 2)
Fraction(1, 2)
>>>
>>> Fraction(1, 2)
Fraction(1, 2)
```

Если указанные числитель и знаменатель имеют общие делители, то перед созданием рационального числа они будут сокращены:

```
>>> Fraction(8, 16), Fraction(15, 30)
(Fraction(1, 2), Fraction(1, 2))
```

В качестве числителя и (или) знаменателя могут быть указаны другие дроби:

```
>>> Fraction(3, Fraction(1, 2))
Fraction(6, 1)
>>>
>>> Fraction(Fraction(1, 2), 3)
Fraction(1, 6)
>>>
>>> Fraction(Fraction(1, 2), Fraction(3, 7))
Fraction(7, 6)
```

Целое и вещественное число, так же можно преобразовать в обыкновенную дробь:

```
>>> Fraction(10)
Fraction(10, 1)
>>>
>>> Fraction(11.11)
Fraction(781796747813847, 70368744177664)
>>>
>>> Fraction(1.25)
Fraction(5, 4)
>>>
>>> Fraction(2e-10)
Fraction(7737125245533627, 38685626227668133590597632)
```

А вот комплексные числа приведут к ошибке:

```
>>> Fraction(1j, 1 + 2j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: both arguments should be Rational instances
```

Но мнимой и действительной частью комплексного числа, могут быть указаны рациональные числа:

```
>>> complex(Fraction(3, 5), Fraction(4, 7))
(0.6+0.5714285714285714j)
```

Так же любая десятичная дробь модуля `Decimal` может быть преобразована в обыкновенную дробь:

```
>>> from decimal import Decimal
>>> Fraction(Decimal('7.7'))
Fraction(77, 10)
```

Любая строка, которая может быть преобразована в любое допустимое число, так же может быть использована для получения обыкновенных дробей:

```
>>> Fraction('111')
Fraction(111, 1)
>>>
>>> Fraction('1.11')
Fraction(111, 100)
>>>
>>> Fraction('1.11e+10')
Fraction(11100000000, 1)
>>>
>>> Fraction('-17/63')
Fraction(-17, 63)
>>>
>>> Fraction('-0.115')
Fraction(-23, 200)
>>>
>>> Fraction('-.115')
Fraction(-23, 200)
>>>
>>> Fraction('1.11 \t\n')
Fraction(111, 100)
>>> Fraction('\t\n 1.11 \t\n')
Fraction(111, 100)
>>> Fraction('\t\n 1.11')
Fraction(111, 100)
>>>
>>> Fraction('\t\n -13/37')
Fraction(-13, 37)
```

Математические операции над рациональными числами

Все арифметические операторы поддерживают вычисления с рациональными числами:

```
>>> x = Fraction(2, 5)
>>> y = Fraction(3, 7)
>>>
>>> x + y, y - x
(Fraction(29, 35), Fraction(1, 35))
>>>
>>> x*y, x/y
(Fraction(6, 35), Fraction(14, 15))
>>>
>>> x**0.5, 2**0.5/5**0.5
(0.6324555320336759, 0.6324555320336759)
>>>
>>> x**y, (x**3)**(1/7)
(0.6752339686501552, 0.6752339686501552)
>>>
>>> y//x
1
>>> x%y
Fraction(2, 5)
```

Однако арифметические операторы не способны к действиям над типами **'Fraction'** и **'decimal.Decimal'** в одном выражении:

```
>>> x + 1.1
1.5
>>>
>>> x + Decimal('1.1')    #  приведет к ошибке
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'Fraction' and 'decimal.Decimal'
```

Функции модуля math способны принимать рациональные числа в качестве аргументов, так как последние, по сути, могут быть просто преобразованы к вещественным числам:

```
>>> import math
>>>
>>> x = Fraction(4, 11)
>>>
>>> math.exp(x), math.exp(4/11)
(1.438551009577678, 1.438551009577678)
```

Fraction — атрибуты и методы

x.numerator

возвращает числитель дроби:

```
>>> from fractions import Fraction
>>>
>>> x = Fraction(3, 8)
>>> x
Fraction(3, 8)
>>>
>>> x.numerator
3
```

x.denominator

возвращает знаменатель дроби:

```
>>> x.denominator
8
```

Fraction.from_float(flt)

принимает `flt` – число типа `float` и возвращает обыкновенную дробь отношение числителя к знаменателю которой максимально приближается к значению `flt`:

```
>>> from fractions import Fraction
>>>
>>> Fraction.from_float(0.5)
Fraction(1, 2)
>>>
>>> Fraction.from_float(0.6)
Fraction(5404319552844595, 9007199254740992)
```

Как можно заметить, из-за хранения чисел с плавающей точкой в двоичном представлении `Fraction.from_float(0.6)` вовсе не равно `Fraction(6, 10)`:

```
>>> Fraction.from_float(0.6) == Fraction(6, 10)
False
```

Fraction.from_decimal(dec)

создает обыкновенную дробь, которая является точным представлением десятичной дроби указанной в `dec`, где `dec` – это экземпляр класса `decimal.Decimal`:

```
>>> from decimal import Decimal
>>>
```

```
>>> Fraction.from_decimal(Decimal('0.7'))
Fraction(7, 10)
>>>
>>> Fraction.from_decimal(Decimal('0.77'))
Fraction(77, 100)
>>>
>>> Fraction.from_decimal(Decimal('0.125'))
Fraction(1, 8)
```

Fraction.limit_denominator(max_denominator=1000000)

возвращает ближайшее рациональное представление указанного числа, знаменатель которого не превышает значение `max_denominator`.

```
>>> import math
>>> math.pi
3.141592653589793
>>>
>>> Fraction(math.pi).limit_denominator(10)
Fraction(22, 7)
>>>
>>> Fraction(math.pi).limit_denominator(1000)
Fraction(355, 113)
>>>
>>> Fraction(math.pi).limit_denominator(100000)
Fraction(312689, 99532)
```

Данный метод позволяет получать как рациональные приближения, так и восстанавливать рациональные значения по их неточному представлению в виде чисел с плавающей точкой:

```
>>> math.cos(math.pi/3)     #  точное значение 0.5
0.5000000000000001
>>>
>>> Fraction(math.cos(math.pi/3)).limit_denominator()
Fraction(1, 2)
>>>
>>>
>>> 2**(1/3)
1.2599210498948732
>>>
>>> Fraction(2**(1/3)).limit_denominator()
Fraction(1054215, 836731)
>>>
>>> 1054215/836731
1.2599210498953666
```

x.__floor__()

возвращает значение типа int, которое является наибольшим среди всех int <= x:

```
>>> Fraction(234, 47).__floor__()
4
```

Ну и учитывая что, все функции модуля math могут обрабатывать обыкновенные дроби, то воспользоваться данным методом можно и так:

```
>>> math.floor(Fraction(234, 47))
4
```

x.__ceil__()

возвращает значение типа int, которое является наименьшим среди всех int >= x:

```
>>> Fraction(234, 47).__ceil__()
5
>>>
>>> import math
>>>
>>> math.ceil(Fraction(234, 47))
5
```

x.__round__(ndigits=None)

Округляет до ближайшего четного числа.

Если `ndigits=None`, то округление выполняется до ближайшего четного целого числа:

```
>>> Fraction('1/2').__round__()
0
>>>
>>> Fraction('3/2').__round__()
2
```

Если ndigits не равно `None`, то округление выполняется до ближайшего числа кратного дроби `Fraction(1, 10**ndigits)`, при этом если последняя цифра 5 , то округление будет выполнено к ближайшему четной цифре:

```
>>> Fraction('7/3').__round__(2)
Fraction(233, 100)
>>>
>>> Fraction('7/3').__round__(3)
Fraction(2333, 1000)
>>>
>>>
>>> Fraction('555/1000').__round__(1)
Fraction(3, 5)
>>>
```

```
>>> Fraction('555/1000').__round__(2)
Fraction(14, 25)
```

Если ndigits не равно None, но меньше 0, то округление выполняется до разряда на который указывает abs(ndigits), при этом если последняя цифра 5 , то округление будет выполнено к ближайшей четной цифре:

```
>>> Fraction('-5555555/10').__round__(-1)
Fraction(-555560, 1)
>>> Fraction('-5555555/10').__round__(-3)
Fraction(-556000, 1)
>>>
>>> Fraction('77777/10').__round__(-3)
Fraction(8000, 1)
```

fractions.gcd(a, b)

возвращает наибольший общий делитель чисел a и b.

Возвращает НОД(a, b). Знак результата зависит от знака b, если b не равен 0, в противном случае знак результата равен знаку a. Возвращает 0, только если a и b оба равны 0:

```
>>> fractions.gcd(135, 15)
15
>>> fractions.gcd(135, -15)
-15
>>> fractions.gcd(-135, 15)
15
>>> fractions.gcd(-135, 0)
-135
>>> fractions.gcd(0, 0)
0
```

С версии 3.5. считается устаревшим, и предпочтительнее использовать команду math.gcd().

Пример:

Вычислить $2:\frac{3}{5}+\frac{3}{5}:2+1\frac{1}{2}:6+6\ 1\frac{1}{2}$

```
from fractions import Fraction as dr
rez=dr(2,1)/dr(3,5)+dr(3,10)+dr(3,2)/6+6/dr(3,2)
print(rez, ' = ',end="")
a=int( rez.numerator / rez.denominator )
b=rez - a
print(a,"+", b, sep="")

>>> 473/60  = 7+53/60
```

V. Модуль cmath

Модуль cmath – предоставляет функции для работы с комплексными числами.

cmath.phase(x) - возвращает фазу комплексного числа (её ещё называют аргументом). Эквивалентно math.atan2(x.imag, x.real). Результат лежит в промежутке [-π, π].

Получить модуль комплексного числа можно с помощью встроенной функции abs().

cmath.polar(x) - преобразование к полярным координатам. Возвращает пару (r, phi).

cmath.rect(r, phi) - преобразование из полярных координат.

cmath.exp(x) - e^x.

cmath.log(x[, base]) - логарифм x по основанию base. Если base не указан, возвращается натуральный логарифм.

cmath.log10(x) - десятичный логарифм.

cmath.sqrt(x) - квадратный корень из x.

cmath.acos(x) - арккосинус x.

cmath.asin(x) - арксинус x.

cmath.atan(x) - арктангенс x.

cmath.cos(x) - косинус x.

cmath.sin(x) - синус x.

cmath.tan(x) - тангенс x.

cmath.acosh(x) - гиперболический арккосинус x.

cmath.asinh(x) - гиперболический арксинус x.

cmath.atanh(x) - гиперболический арктангенс x.

cmath.cosh(x) - гиперболический косинус x.

cmath.sinh(x) - гиперболический синус x.

cmath.tanh(x) - гиперболический тангенс x.

cmath.isfinite(x) - True, если действительная и мнимая части конечны.

cmath.isinf(x) - True, если либо действительная, либо мнимая часть бесконечна.

cmath.isnan(x) - True, если либо действительная, либо мнимая часть NaN.

cmath.pi - π.

cmath.e - e.

VI. Модуль struct -упаковка данных в бинарный файл

В некоторых приложениях приходится иметь дело с упакованными двоичными данными, которые создаются, например, программами на языке C. Для их сохранения и восстановления можно воспользоваться модулем struct из стандартной библиотеки Python.

Пример записи в файл и чтения из файла:

```
import struct

myfile = open("data.bin", "wb")
# упаковываем данные
bytes = struct.pack(b'>i4sh', 7, b'spam', 8)
myfile.write(bytes)
myfile.close()

myfile = open("data.bin", "rb")
data = myfile.read()
# извлекаем данные
values = struct.unpack(b'>i4sh', data)
print(values)
```

Методы

struct.unpack(format, data)

Распаковывает данные из структуры

Примеры:

```
# два целых 4х байтовые числа прямого порядка
struct.unpack('>LL',
b'\x00\x00\x00\x9a\x00\x00\x00\x8d')
# (154, 141)

# два целых 4х байтовые числа прямого порядка
struct.unpack('>2L',
b'\x00\x00\x00\x9a\x00\x00\x00\x8d')
# (154, 141)

# два целых 4х байтовые числа прямого порядка,
пропустив 1 и 2 байта по краям
struct.unpack('>1x2L2x',
     b'\x00\x00\x00\x00\x9a\x00\x00\x00\x8d\x00\x00')
# (154, 141)
```

struct.pack(format, data)

Пакует данные в структуру

```
struct.pack('>L', 154)
# b'\x00\x00\x00\x9a'

struct.pack('>L', 141)
# b'\x00\x00\x00\x8d'
```

Спецификаторы формата

> - прямой порядок
< - обратный порядок
x - пропустить один байт
b - знаковый один байт
B - беззнаковый байт
h - знаковое короткое целое число, 2 байта
H - беззнаковое короткое целое число, 2 байта
i - знаковое целое число, 4 байта
I - беззнаковое целое число, 4 байта
l - знаковое длинное целое число, 4 байта
L - беззнаковое длинное целое число, 4 байта
Q - беззнаковое очень длинное целое число, 8 байт
f - число с плавающей точкой, 4 байта
d - число с плавающей точкой двойной точности, 8 байт
p - счетчик и символы, 1 + count байт
s - символы, count символов

Пример записи и чтения вещественных данных в бинаный файл

```
import sys
import struct
myfile = open("data.bin", "wb")
# упаковываем данные
for i in range(100):
    x1=10.0*float(i+0)
    x2=10.0*float(i+1)
    x3=10.0*float(i+2)
    x4=10.0*float(i+3)
    x5=10.0*float(i+4)
    print(x1,x2,x3,x4,x5)
    b=struct.pack(b'>fffff', x1,x2,x3,x4,x5)
    myfile.write(b)
myfile.close()
#sys.exit(0)
myfile = open("data.bin", "rb")
while (True):
    data = myfile.read(4*5)
    if data==b'':
        break
    # извлекаем данные
    values = struct.unpack(b'>fffff', data)
    y1=values[0]
```

```
    y2=values[1]
    y3=values[2]
    y4=values[3]
    y5=values[4]
    print(y1, y2, y3, y4, y5 )
myfile.close()
```

Пример записи файла на FORTRAN и чтения на PYTHON вещественных данных в бинаный файл

Формирование файла:

```
! компилятор ming gfortran
  implicit none
  integer :: j, i
  real(4), DIMENSION(5) :: buf
  character(40) :: fname='d:\float5.bin'
  open(unit=50, file=trim(fname), status='UNKNOWN',
form='UNFORMATTED',err=100)
!
!  запись в файле содержит в "начале и "в конце" записи
!  "дискриптор" ЧЕТЫРЕ байта - количество байт
!  в записи не считая дискрипторов
!
  do i=1,50
     buf(1)=float(i)
     buf(2)=float(i+1)
     buf(3)=float(i+2)
     buf(4)=float(i+3)
     buf(5)=float(i+4)
     write(50) (buf(j), j=1,5)
     write(*, '( 5(g12.5,1x) )' ) (buf(j), j=1,5)
  enddo
  close(50)
  stop 0
100 write(*,*) '* error open file ', trim(fname), '*'
    stop 16
    end
```

Чтение файла:

```
import sys
import struct
myfile = open(r"d:\float5.bin", "rb")
##kbyteB=myfile.read(4) # дискриптор записи
nzap=0
```

```
while (True):
    kbyteB=myfile.read(4) # дискриптор записи -
                          # кол-во байт в этой записи
    if kbyteB==b'':
        break
    nzap+=1
    kbyte = struct.unpack(b'<L', kbyteB)[0]
    print('запись № ',nzap,' содержит ',kbyte,' байт')
    data = myfile.read(4*5)  #чтение записи
    if data==b'':
        break
    # извлекаем данные
    values = struct.unpack(b'<fffff', data)
    y1=values[0]
    y2=values[1]
    y3=values[2]
    y4=values[3]
    y5=values[4]
    print(y1, y2, y3, y4, y5 ) #values)
    kbyteB=myfile.read(4) # дискриптор конца записи -
                          # кол-во байт в этой записи
myfile.close()
```

VII. Файлы CSV

Программисты часто сталкиваются с задачей обработки больших объемов структурированных данных.Python имеет встроенную библиотеку CSV, с помощью которой программист может работать со специальными CSV файлами. Это своего рода электронные таблицы.

Каждая строка в файле csv представляет отдельную запись или строку, которая состоит из отдельных столбцов, разделенных запятыми. Собственно поэтому формат и называется Comma Separated Values. Но хотя формат csv - это формат текстовых файлов, Python для упрощения работы с ним предоставляет специальный встроенный модуль csv

Что такое файлы CSV

Файл CSV – это особый вид файла, который позволяет структурировать большие объемы данных.

По сути, он является обычным текстовым файлом, однако каждый новый элемент отделен от предыдущего запятой или другим разделителем. Обычно каждая запись начинается с новой строки. Данные CSV можно легко экспортировать в электронные таблицы или базы данных. Программист может расширять CSV файл, добавляя новые строки.

Пример CSV файла, где в качестве разделителя используется запятая:

```
Имя,Профессия,Год рождения
Виктор,Токарь,1995
Сергей,Сварщик,1983
```

Как видно из примера, в первой строке обычно указывается, какая информация будет находиться в каждом столбце. Кроме того, после последнего элемента строки запятая не ставиться, интерпретатор определяет конец строки по символу переноса.

Вместо запятой можно использовать любой другой разделитель, поэтому при чтении CSV файла нужно заранее знать, какой символ используется.

Важно помнить, что CSV – это обычный текстовый файл, который не поддерживает символы в кодировках, отличающихся от ASCII или Unicode.

Библиотека CSV

Эта основная библиотека для работы с CSV файлами в Python.

Библиотека csv является встроенной, поэтому её не нужно скачивать, достаточно использовать обычный импорт:

```
import csv
```

Чтение из файлов (парсинг)

Для того чтобы прочитать данные из файла, программист должен создать объект **reader**:

```
reader_object = csv.reader(file, delimiter = ",")
```

reader имеет метод **__next__(),** то есть является итерируемым объектом, поэтому чтение из файла происходит следующим образом:

```
import csv
with open("classmates.csv", encoding='utf-8') as r_file:
    # Создаем объект reader, указываем символ-разделитель ","
    file_reader = csv.reader(r_file, delimiter = ",")
    # Счетчик для подсчета количества строк и вывода заголовков
столбцов
    count = 0
    # Считывание данных из CSV файла
    for row in file_reader:
        if count == 0:
            # Вывод строки, содержащей заголовки для столбцов
            print(f'Файл содержит столбцы: {", ".join(row)}')
        else:
            # Вывод строк
            print(f'{row[0]} - {row[1]} и он родился в {row[2]}
                                                        году.')
        count += 1
    print(f'Всего в файле {count} строк.')
```

Предположим, что у нас есть CSV файл, который содержит следующую информацию:

```
Имя,Успеваемость,Год рождения
Саша,отличник,200
Маша,хорошистка,1999
Петя,троечник,2000
```

Тогда, если открыть этот файл в нашей программе, то будут получены следующие результаты:

```
Файл содержит столбцы: Имя, Успеваемость, Год рождения
    Саша - отличник и он родился в 200 году.
    Маша - хорошистка и он родился в 1999 году.
    Петя - троечник и он родился в 2000 году.
Всего в файле 4 строк.
```

Использование конструкции **with…as** позволяет программисту быть уверенным, что файл будет закрыт, даже если при выполнении кода произойдет какая-то ошибка.

Обратите внимание, что при открытии нужно указать правильную кодировку, в которой сохранены данные. В данном случае **encoding=’utf-8′.** Если не указывать, то будет использоваться кодировка по умолчанию. Для Windows это **cp1251**.

Библиотека CSV позволяет работать с файлами, как со словарями, для этого нужно создать объект **DictReader**. Обращаться к элементам можно по имени столбцов, а не с помощью индексов. Для того, чтобы исходная программа делала аналогичный вывод, её следует изменить следующим образом:

```
import csv
with open("classmates.csv", encoding='utf-8') as r_file:
    #Создаем объект DictReader, указываем символ-разделитель ","
    file_reader = csv.DictReader(r_file, delimiter = ",")
    # Счетчик для подсчета количества строк и вывода
                                            заголовков столбцов
    count = 0
    # Считывание данных из CSV файла
    for row in file_reader:
        if count == 0:
            # Вывод строки, содержащей заголовки для столбцов
            print(f'Файл содержит столбцы: {", ".join(row)}')
        # Вывод строк
        print(f' {row["Имя"]} - {row["Успеваемость"]}', end='')
        print(f' и он родился в {row["Год рождения"]} году.')
        count += 1
    print(f'Всего в файле {count + 1} строк.')
```

Обращаться к элементам по названию столбца более удобно, кроме того, это упрощает понимание кода.

Обратите внимание, что в цикл for при первой итерации будет записан в row не шапка таблицы, а первая её строка. Поэтому при выводе количества строк переменную count увеличили на 1.

Дополнительные параметры объекта DictReader

DictReader имеет параметры:

- `dialect` — Набор параметров для форматирования информации. Подробнее про них ниже.
- `line_num` — Устанавливает количество строк, которое может быть прочитано.

- `fieldnames` — Определяет заголовки для столбцов, если не определить атрибут, то в него запишутся элементы из первой прочитанной строки файла. Заголовки нужны для того, чтобы легко было понять, какая информация содержится или должна содержаться в столбце.

Например, если бы в classmates.csv не было бы первой строки с заголовками, то можно было бы его открыть следующим образом:

```
fieldnames = ['Имя', 'Успеваемость', 'Год рождения']
file_reader = csv.DictReader(r_file, fieldnames =
fieldnames)
```

Также можно использовать метод __next__() для получения следующей строки. Этот метод делает объект reader итерируемым. То есть он вызывается при каждой итерации и возвращает следующую строку. Этот метод и используется при каждой итерации в цикле for для получения очередной строки.

Запись в файлы

Для записи информации в CSV файл необходимо создать объект writer:

```
file_writer = csv.writer(w_file, delimiter = "\t")
```

Для записи в файл данных используется метод writerow(), который имеет следующий синтаксис:

```
writerow(["Имя", "Фамилия", "Отчество"])
```

Код программы для записи в CSV файл выглядит так:

```
import csv
with open("classmates.csv", mode="w", encoding='utf-8')
as w_file:
    file_writer = csv.writer(w_file, delimiter = ",",
                                lineterminator="\r")
    file_writer.writerow(["Имя", "Класс", "Возраст"])
    file_writer.writerow(["Женя", "3", "10"])
    file_writer.writerow(["Саша", "5", "12"])
    file_writer.writerow(["Маша", "11", "18"])
```

Обратите внимание, что при записи использовался, `lineterminator="\r"`. Это разделитель между строками таблицы, по умолчанию он "\r\n".

После выполнения программы в файле CSV будет следующий текст:

```
Имя,Класс,Возраст
Женя,3,10
Саша,5,12
Маша,11,18
```

В качестве параметра метод writerow() принимает список, элементы которого будут записаны в строку через символ-разделитель.

Запись в файл также может быть осуществлена с помощью объекта DictWriter.

Важно помнить, что он требует явного указания параметра **fieldnames**. В качестве аргумента метода **writerow** используется словарь.

Код программы выглядит так:

```
import csv
with open("classmates.csv", mode="w", encoding='utf-8')
                                                    as w_file:
    names = ["Имя", "Возраст"]
    file_writer = csv.DictWriter(w_file, delimiter = ",",
                                       lineterminator="\r",
                                           fieldnames=names)
    file_writer.writeheader()
    file_writer.writerow({"Имя": "Саша", "Возраст": "6"})
    file_writer.writerow({"Имя": "Маша", "Возраст": "15"})
    file_writer.writerow({"Имя": "Вова", "Возраст": "14"})
```

Вывод в файл будет следующим:

```
Имя,Возраст
Саша,6
Маша,15
Вова,14
```

Дополнительные параметры DictWriter

Объект writer также имеет атрибут **dialect**, который определяет, как будут форматироваться данные при записи в файл, про него будет описано ниже.

Кроме того, **writer имеет методы:**

- **writerows(rows)** — Записывает все элементы строк.
- **writeheader()** —Выводит заголовки для столбцов. Заголовки должны быть переданы объекту **writer** в виде списка, как атрибут **fieldnames**. **writeheader** был использован в предыдущем примере.

Рассмотрим применение writerows:

```
file_writer.writerows([{"Имя": "Саша", "Возраст": "6"},
                       {"Имя": "Маша", "Возраст": "15"},
                      {"Имя": "Вова", "Возраст": "14"}])
```

Диалекты

Чтобы каждый раз не указывать формат входных и выходных данных, определенные параметры форматирования сгруппированы в диалекты (**dialect**).

При создании объекта **reader** или **writer** программист может указать нужный ему диалект, кроме того, некоторые параметры диалекта можно переопределить вручную, также указав их при создании объекта.

Для создания диалекта используется команда:

```
register_dialect("имя", delimiter = "\t", ...)
```

Класс **Dialect** позволяет определить следующие **атрибуты форматирования**:

Атрибут	Значение
`delimiter`	Устанавливает символ, с помощью которого разделяются элементы в файле. По умолчанию используется запятая.

Атрибут	Значение
doublequote	Если True, то символ quotechar удваивается, если False, то к символу qutechar добавляется ecsapechar в качестве префикса.
escapechar	Строка из одного символа, которая используется для экранирования символа-разделителя.
lineterminator	Определяет разделитель для строк, по умолчанию используется «\r\n»
quotechar	Определяет символ, который используется для окружения символа-разделителя. По умолчанию используются двойные кавычки, то есть quotechar = ‘ “ ‘.
quoting	Определяет символ, который используется для экранирования символа разделителя (если не используются кавычки).
skipinitialspace	Если установить значение этого параметра в True, то все пробелы после символа-разделителя будут игнорироваться.
strict	Если установить в True, то при неправильном вводе CSV будет возбуждаться исключение Error.

Пример использования:

```
import csv
csv.register_dialect('my_dialect', delimiter=':',
                                   lineterminator="\r")
with open("classmates.csv", mode="w", encoding='utf-8')
                                              as w_file:
    file_writer = csv.writer(w_file, 'my_dialect')
    file_writer.writerow(["Имя", "Класс", "Возраст"])
    file_writer.writerow(["Женя", "3", "10"])
    file_writer.writerow(["Саша", "5", "12"])
    file_writer.writerow(["Маша", "11", "18"])
```

В результате получим:

```
Имя:Класс:Возраст
Женя:3:10
Саша:5:12
Маша:11:18
```

Пример

```
import csv
FILENAME = "users.csv"
users = [
```

```
    [1, "математика", 75 ],
    [2, "программировани", 85],
    [3, "мат анализ", 34]
]

with open(FILENAME, "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(users)

with open(FILENAME, "a", newline="") as file:
    user = [4, "веб прог", 80]
    writer = csv.writer(file)
    writer.writerow(user)

d={}
with open(FILENAME, "r", newline="") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row[0], " - ", row[1], " - ", row[2])
        d[row[1]]=int(row[2])
print("d=",d)

#сформируем список D1 из словаря и d
# и его сортируем по ключу словаря key=lambda x:x[0]
# если хотим по значению, то пишите key=lambda x:x[1]
# если хотим в обратном порядке, то пишите key=lambda x:-x[1]
D1=sorted(d.items(),key=lambda x:-x[1]) # по ключам

print("D1=",D1)   # печать списка

#  отсортированный список в словарь D2 и его печать
D2={D1[i][0] : D1[i][1] for i in range(len(D1)) }
#печать таблицей D2
for kk, vv in D2.items():
    print(vv,'\t',kk)

ll=list(D2.items())
kkl=len(ll)-1
print("maximum score of", ll[0][1], "in the subject", ll[0][0])
print("minimum score of", ll[kkl][1], "in the subject",
ll[kkl][0])

user1=[]
nn=0
for kk, vv in D2.items():
    user1.append([nn,kk,vv])
    nn+=1

namerow=["№пп", "предмет", "оценка"]
with open(FILENAME, "w", newline="") as file: #encoding='utf-8'
    writer=csv.writer(file, delimiter=",", lineterminator="\n")
    writer.writerow(namerow)
    writer.writerows(user1)
```

VIII. Модуль shelve (*.ini)

Для работы с бинарными файлами в Python может применяться еще один модуль - `shelve`. Он сохраняет объекты в файл с определенным ключом. Затем по этому ключу может извлечь ранее сохраненный объект из файла. Процесс работы с данными через модуль `shelve` напоминает работу со словарями, которые также используют ключи для сохранения и извлечения объектов.

Для открытия файла модуль shelve использует функцию open():

```
open(путь_к_файлу[, flag="c"[, protocol=None[,
                                    writeback=False]]])
```

Где параметр `flag` может принимать значения:

- **c**: файл открывается для чтения и записи (значение по умолчанию). Если файл не существует, то он создается.
- **r**: файл открывается только для чтения.
- **w**: файл открывается для записи.
- **n**: файл открывается для записи Если файл не существует, то он создается. Если он существует, то он перезаписывается

Для закрытия подключения к файлу вызывается метод `close()`:

```
import shelve
d = shelve.open(filename)
d.close()
```

Либо можно открывать файл с помощью оператора **`with`**.

Сохраним и считаем в файл несколько объектов:

```
import shelve

FILENAME = "states2"
with shelve.open(FILENAME) as states:
    states["London"] = "Great Britain"
    states["Paris"] = "France"
    states["Berlin"] = "Germany"
    states["Madrid"] = "Spain"

with shelve.open(FILENAME) as states:
    print(states["London"])
    print(states["Madrid"])
```

Запись данных предполагает установку значения для определенного ключа:

```
states["London"] = "Great Britain"
```

А чтение из файла эквивалентно получению значения по ключу:

```
print(states["London"])
```

В качестве ключей используются строковые значения.

При чтении данных, если запрашиваемый ключ отсутствует, то генерируется исключение. В этом случае перед получением мы можем проверять на наличие ключа с помощью оператора **in**:

```
with shelve.open(FILENAME) as states:
    key = "Brussels"
    if key in states:
        print(states[key])
```

Также можно использовать метод `get()`. Первый параметр метода - ключ, по которому следует получить значение, а второй - значение по умолчанию, которое возвращается, если ключ не найден.

```
with shelve.open(FILENAME) as states:
    state = states.get("Brussels", "Undefined")
    print(state)
```

Используя цикл **for**, можно перебрать все значения из файла:

```
with shelve.open(FILENAME) as states:
    for key in states:
        print(key," - ", states[key])
```

Метод `keys()` возвращает все ключи из файла, а метод `values()` - все значения:

```
with shelve.open(FILENAME) as states:

    for city in states.keys():
        print(city, end=" ")        # London Paris
Berlin Madrid
    print()
    for country in states.values():
        print(country, end=" ")     # Great Britain
                                # France Germany Spain
```

Еще один метод `items()` возвращает набор кортежей. Каждый кортеж содержит ключ и значение.

```
with shelve.open(FILENAME) as states:

    for state in states.items():
        print(state)
```

Консольный вывод:

```
("London", "Great Britain")
("Paris", "France")
("Berlin", "Germany")
("Madrid", "Spain")
```

Обновление данных

Для изменения данных достаточно присвоить по ключу новое значение, а для добавления данных - определить новый ключ:

```
import shelve

FILENAME = "states2"
with shelve.open(FILENAME) as states:
    states["London"] = "Great Britain"
    states["Paris"] = "France"
    states["Berlin"] = "Germany"
    states["Madrid"] = "Spain"

with shelve.open(FILENAME) as states:

    states["London"] = "United Kingdom"
    states["Brussels"] = "Belgium"
    for key in states:
        print(key, " - ", states[key])
```

Удаление данных

Для удаления с одновременным получением можно использовать функцию `pop()`, в которую передается ключ элемента и значение по умолчанию, если ключ не найден:

```
with shelve.open(FILENAME) as states:

    state = states.pop("London", "NotFound")
    print(state)
```

Также для удаления может применяться оператор `del`:

```
with shelve.open(FILENAME) as states:

    del states["Madrid"]     # удаляем объект
                              #  с ключом Madrid
```

Для удаления всех элементов можно использовать метод `clear()`:

```
with shelve.open(FILENAME) as states:

    states.clear()
```

IX. Модуль OS и работа с файловой системой

Ряд возможностей по работе с каталогами и файлами предоставляет встроенный модуль `os`. Он содержит много функций, ниже рассмотрены только основные из них:

- mkdir(): создает новую папку
- rmdir(): удаляет папку

- rename(): переименовывает файл
- remove(): удаляет файл

Создание и удаление папки

Для создания папки применяется функция mkdir(), в которую передается путь к создаваемой папке:

```
import os

# путь относительно текущего скрипта
os.mkdir("hello")
# абсолютный путь
os.mkdir("c://somedir")
os.mkdir("c://somedir/hello")
```

Для удаления папки используется функция rmdir(), в которую передается путь к удаляемой папке:

```
import os

# путь относительно текущего скрипта
os.rmdir("hello")
# абсолютный путь
os.rmdir("c://somedir/hello")
```

Переименование файла

Для переименования вызывается функция rename(source, target), первый параметр которой - путь к исходному файлу, а второй - новое имя файла. В качестве путей могут использоваться как абсолютные, так и относительные.

Например, пусть в папке C://SomeDir/ располагается файл somefile.txt. Переименуем его в файл "hello.txt":

```
import os

os.rename("C://SomeDir/somefile.txt",
                        "C://SomeDir/hello.txt")
```

Удаление файла

Для удаления вызывается функция remove(), в которую передается путь к файлу:

```
import os

os.remove("C://SomeDir/hello.txt")
```

Существование файла

Если попытаться открыть файл, который не существует, то Python «выбросит» исключение `FileNotFoundError`.

Для «отлова» исключения можно использовать конструкцию `try...except`. Однако можно уже до открытия файла проверить, существует ли он или нет с помощью метода `os.path.exists(path)`.

В этот метод передается полное имя файла (путь и имя), существование которого необходимо проверить:

```
filename = input("Введите путь к файлу: ")
if os.path.exists(filename):
    print("Указанный файл существует")
else:
    print("Файл не существует")
```

Работа с операционной системой

Модуль os так же предоставляет множество функций для работы с операционной системой, причём их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми. Здесь будут приведены наиболее часто используемые из них (некоторые функции из этого модуля поддерживаются не всеми ОС).

os.name - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

os.environ - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

os.getlogin() - имя пользователя, вошедшего в терминал (Unix).

os.getpid() - текущий id процесса.

os.uname() - информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется реализацией), release - релиз, version - версия, machine - идентификатор машины.

os.access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True) - проверка доступа к объекту у текущего пользователя.

Флаги: **os.F_OK** - объект существует, **os.R_OK** - доступен на чтение, **os.W_OK** - доступен на запись, **os.X_OK** - доступен на исполнение.

os.chdir(path) - смена текущей директории.

os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True) - смена прав доступа к объекту (mode - восьмеричное число).

os.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True) - меняет id владельца и группы (Unix).

os.getcwd() - текущая рабочая директория.

os.link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True) - создаёт жёсткую ссылку.

os.listdir(path=".") - список файлов и директорий в папке.

os.mkdir(path, mode=0o777, *, dir_fd=None) - создаёт директорию. OSError, если директория существует.

os.makedirs(path, mode=0o777, exist_ok=False) - создаёт директорию, создавая при этом промежуточные директории.

os.remove(path, *, dir_fd=None) - удаляет путь к файлу.

os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None) - переименовывает файл или директорию из src в dst.

os.renames(old, new) - переименовывает old в new, создавая промежуточные директории.

os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None) - переименовывает из src в dst с принудительной заменой.

os.rmdir(path, *, dir_fd=None) - удаляет пустую директорию.

os.removedirs(path) - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

os.symlink(source, link_name, target_is_directory=False, *, dir_fd=None) - создаёт символическую ссылку на объект.

os.sync() - записывает все данные на диск (Unix).

os.truncate(path, length) - обрезает файл до длины length.

os.utime(path, times=None, *, ns=None, dir_fd=None, follow_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

os.walk(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

Пример – вывести на консоль список всех *.txt файлов, размещенных в папке l:\dist и её подпапках:

```
import os
from os.path import join
for (root, dirs, files) in os.walk('l:\\dist'):
      for filename in files:
            if filename.endswith('.txt'):
                  thefile = os. path.join(root, filename)
                  print( os.path.getsize(thefile), thefile)
```

os.system(command) - исполняет системную команду, возвращает код её завершения (в случае успеха 0).

os.urandom(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

X. Пример команд работы с файлами и файловой системой

Рассмотрим 8-м крайне важных команд для работы с файлами, папками и файловой системой в целом.

Показать текущий каталог

Самая простая и вместе с тем одна из самых важных команд для Python-разработчика. Она нужна потому, что чаще всего разработчики имеют дело с относительными путями. Но в некоторых случаях важно знать, где мы находимся.

Относительный путь хорош тем, что работает для всех пользователей, с любыми системами, количеством дисков и так далее.

Так вот, для того чтобы показать текущий каталог используем встроенную в Python OS-библиотеку:

```
import os
os.getcwd()
```

Имейте в виду, что возвращаемый путь является абсолютным.

Проверяем, существует файл или каталог

Прежде чем задействовать команду по созданию файла или каталога, стоит убедиться, что аналогичных элементов нет. Это поможет избежать ряда ошибок при работе приложения, включая перезапись существующих элементов с данными.

Функция os.path.exists () принимает аргумент строкового типа, который может быть либо именем каталога, либо файлом.

Проверим, существует ли каталог sample_data:

```
os.path.exists('sample_data')
```

```
[3]  os.path.exists('sample_data')

     True
```

Эта же команда подходит и для работы с файлами:

```
os.path.exists('sample_data/README.md')
```

```
[4]  os.path.exists('sample_data/README.md')

     True
```

Если папки или файла нет, команда возвращает false.

```
[5]  os.path.exists('foobar')

     False
```

Объединение компонентов пути

В предыдущем примере использовался слеш "/" для разделителя компонентов пути. В принципе это нормально, но не рекомендуется. Если вы хотите, чтобы ваше приложение было кросс платформенным, такой вариант не подходит. Так, некоторые старые версии ОС Windows распознают только слеш "\" в качестве разделителя.

Python решает эту проблему благодаря функции `os.path.join ()`.

Давайте перепишем вариант из примера в предыдущем пункте, используя эту функцию:

```
os.path.exists(os.path.join('sample_data',
                                                    'README.md'))
```

```
[7]  os.path.exists(os.path.join('sample_data', 'README.md'))

     True
```

Создание директории

Создадим директорию с именем `test_dir` внутри текущей директории. Для этого можно использовать функцию `os.mkdir()`:

```
os.mkdir('test_dir')
```

Давайте посмотрим, как это работает на практике.

```
[9]  print(f"test_dir existing: {os.path.exists('test_dir')}")
     os.mkdir('test_dir')
     print(f"test_dir existing: {os.path.exists('test_dir')}")

     test_dir existing: False
     test_dir existing: True
```

Если же мы попытаемся создать каталог, который уже существует, то получим исключение.

```
[11] os.mkdir('test_dir')

---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
<ipython-input-11-48d7b58469aa> in <module>()
----> 1 os.mkdir('test_dir')

FileExistsError: [Errno 17] File exists: 'test_dir'
```

Именно поэтому рекомендуется всегда проверять наличие каталога с определенным названием перед созданием нового:

```
if not os.path.exists('test_dir'):
    os.mkdir('test_dir')
```

Еще один совет по созданию каталогов. Иногда нам нужно создать подкаталоги с уровнем вложенности 2 или более. Если мы все еще используем `os.mkdir ()`, нам нужно будет сделать это несколько раз.

В этом случае мы можем использовать `os.makedirs()`. Эта функция создаст все промежуточные каталоги:

```
os.makedirs(os.path.join('test_dir', 'level_1',
                                    'level_2', 'level_3'))
```

Вот что получается в результате.

Показываем содержимое директории

Еще одна полезная команда — `os.listdir()`. Она показывает все содержимое каталога.

Команда отличается от `os.walk ()`, где последний рекурсивно показывает все, что находится «под» каталогом. `os.listdir ()` намного проще в использовании, потому что просто возвращает список содержимого:

```
os.listdir('sample_data')
```

```
[13] os.listdir('sample_data')

     ['README.md',
      'anscombe.json',
      'california_housing_train.csv',
      'california_housing_test.csv',
      'mnist_train_small.csv',
      'mnist_test.csv']
```

В некоторых случаях нужно что-то более продвинутое — например, поиск всех CSV-файлов в каталоге «sample_data». В этом случае самый простой способ — использовать встроенную библиотеку glob (см. ниже):

```
from glob import
globlist(glob(os.path.join('sample_data', '*.csv')))
```

```
[14] from glob import glob
```

```
[15] list(glob(os.path.join('sample_data', '*.csv')))
```

```
['sample_data/california_housing_train.csv',
 'sample_data/california_housing_test.csv',
 'sample_data/mnist_train_small.csv',
 'sample_data/mnist_test.csv']
```

Перемещение файлов

Самое время попробовать переместить файлы из одной папки в другую. Рекомендованный способ — еще одна встроенная библиотека `shutil` (см. ниже).

Переместим все CSV-файлы из директории «sample_data» в директорию «test_dir»:

```
import shutil for file in
list(glob(os.path.join('sample_data', '*.csv'))):
    shutil.move(file, 'test_dir')
```

Кстати, есть два способа выполнить задуманное. Например, мы можем использовать библиотеку OS, если не хочется импортировать дополнительные библиотеки. Как `os.rename`, так и `os.replace` подходят для решения задачи.

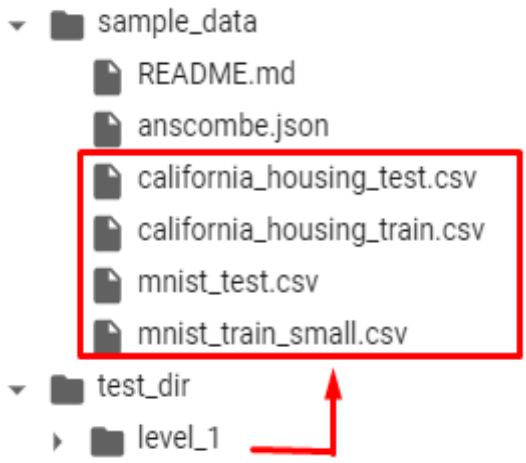
Но обе они недостаточно «умные», чтобы позволить перемесить файлы в каталог.

```
[21] for file in list(glob(os.path.join('test_dir', '*.csv'))):
         os.rename(file, 'sample_data')
```

```
---------------------------------------------------------------------------
IsADirectoryError                         Traceback (most recent call last)
<ipython-input-21-9675c94bbcdd> in <module>()
      1 for file in list(glob(os.path.join('test_dir', '*.csv'))):
----> 2     os.rename(file, 'sample_data')

IsADirectoryError: [Errno 21] Is a directory: 'test_dir/california_housing_train.csv' -> 'sample_data'
```

Чтобы все это работало, нужно явно указать имя файла в месте назначения. Ниже — код, который это позволяет сделать:

```
for file in list(glob(os.path.join('test_dir',
                                   '*.csv'))):
    os.rename(
        file,
        os.path.join(
            'sample_data',
            os.path.basename(file)
    ))
```



Здесь функция `os.path.basename()` предназначена для извлечения имени файла из пути с любым количеством компонентов.

Другая функция, `os.replace()`, делает то же самое. Но разница в том, что `os.replace()` не зависит от платформы, тогда как `os.rename()` будет работать только в системе Unix / Linux.

Еще один минус — в том, что обе функции не поддерживают перемещение файлов из разных файловых систем, в отличие от `shutil`.

Поэтому рекомендуется использовать `shutil.move()` для перемещения файлов.

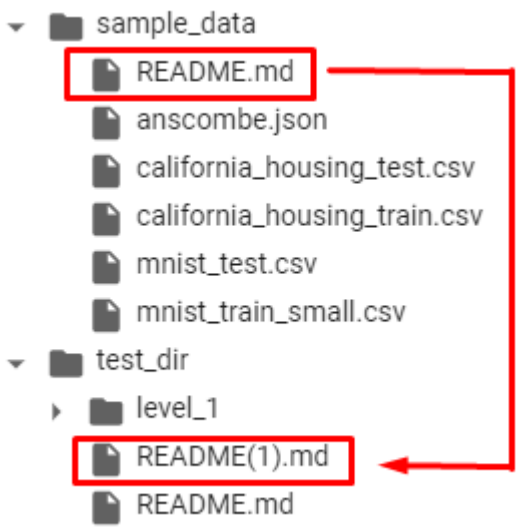
Копирование файлов

Аналогичным образом shutil подходит и для копирования файлов по уже упомянутым причинам.

Если нужно скопировать файл README.md из папки «sample_data» в папку «test_dir», поможет функция `shutil.copy()`:

```
shutil.copy(
    os.path.join('sample_data', 'README.md'),
    os.path.join('test_dir')
)
```

```
[26] shutil.copy(
         os.path.join('sample_data', 'README.md'),
         os.path.join('test_dir', 'README(1).md')
     )

     'test_dir/README(1).md'
```



Удаление файлов и папок

Когда нужно удалить файл, нужно воспользоваться командой `os.remove()`:

```
os.remove(os.path.join('test_dir', 'README(1).md'))
```

Если требуется удалить каталог, то используем `os.rmdir()`:

```
os.rmdir(os.path.join('test_dir', 'level_1', 'level_2',
                                              'level_3'))
```

Однако `os.rmdir()` может удалить только пустой каталог.

На приведенном выше рис. видим, что удалить можно лишь каталог level_3.

Что если необходимо рекурсивно удалить каталог level_1? В этом случае используем `shutil`.

```
[28] os.rmdir(os.path.join('test_dir', 'level_1'))

---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-28-352a126b9aab> in <module>()
----> 1 os.rmdir(os.path.join('test_dir', 'level_1'))

OSError: [Errno 39] Directory not empty: 'test_dir/level_1'
```

Функция `shutil.rmtree()` сделает все, что нужно:

```
shutil.rmtree(os.path.join('test_dir', 'level_1'))
```

Пользоваться ею нужно с осторожностью, поскольку она безвозвратно удаляет все содержимое каталога.

XI. Модуль shutil

Модуль `shutil` содержит набор функций высокого уровня для обработки файлов, групп файлов, и папок. В частности, доступные здесь функции позволяют копировать, перемещать и удалять файлы и папки. Часто используется вместе с модулем os.

Операции над файлами и директориями

`shutil.copyfileobj``(fsrc, fdst[, length])`
- скопировать содержимое одного файлового объекта (`fsrc`) в другой (`fdst`).

Необязательный параметр `length` - размер буфера при копировании (чтобы весь, возможно огромный, файл не читался целиком в память).

При этом, если позиция указателя в `fsrc` не 0 (т.е. до этого было сделано что-то наподобие `fsrc.read(47)`), то будет копироваться содержимое начиная с текущей позиции, а не с начала файла.

`shutil.copyfile`(src, dst, follow_symlinks=True)
- копирует содержимое (но не метаданные) файла `src` в файл `dst`.

Возвращает `dst` (т.е. куда файл был скопирован). `src` и `dst` это строки - пути к файлам. `dst` должен быть полным именем файла.

Если `src` и `dst` представляют собой один и тот же файл, исключение **`shutil.SameFileError`**.

Если `dst` существует, то он будет перезаписан.

Если follow_symlinks=False и src является ссылкой на файл, то будет создана новая символическая ссылка вместо копирования файла, на который эта символическая ссылка указывает.

shutil.copymode(src, dst, follow_symlinks=True)

- копирует права доступа из src в dst. Содержимое файла, владелец, и группа не меняются.

shutil.copystat(src, dst, follow_symlinks=True)

- копирует права доступа, время последнего доступа, последнего изменения, и флаги src в dst. Содержимое файла, владелец, и группа не меняются.

shutil.copy(src, dst, follow_symlinks=True) - копирует содержимое файла src в файл или папку dst. Если dst является директорией, файл будет скопирован с тем же названием, что было в src. Функция возвращает путь к местонахождению нового скопированного файла.

Если follow_symlinks=False, и src это ссылка, dst будет ссылкой.

Если follow_symlinks=True, и src это ссылка, dst будет копией файла, на который ссылается src

copy() копирует содержимое файла, и права доступа.

shutil.copy2(src, dst, follow_symlinks=True)

- как copy(), но пытается копировать все метаданные.

shutil.copytree(src, dst, symlinks=False,
ignore=None, copy_function=copy2,
ignore_dangling_symlinks=False)

- рекурсивно копирует всё дерево директорий с корнем в src, возвращает директорию назначения.

Директория dst не должна существовать. Она будет создана, вместе с пропущенными родительскими директориями.

Права и времена у директорий копируются copystat(), файлы копируются с помощью функции copy_function (по умолчанию shutil.copy2()).

Если symlinks=True, ссылки в дереве src будут ссылками в dst, и метаданные будут скопированы настолько, насколько это возможно.

Если False (по умолчанию), будут скопированы содержимое и метаданные файлов, на которые указывали ссылки.

Если symlinks=False, если файл, на который указывает ссылка, не существует, будет добавлено исключение в список ошибок, в исключении shutil.Error в конце копирования.

Можно установить флаг `ignore_dangling_symlinks=True`, чтобы скрыть данную ошибку.

Если `ignore` не `None`, то это должна быть функция, принимающая в качестве аргументов имя директории, в которой сейчас `copytree()`, и список содержимого, возвращаемый `os.listdir()`. Т.к. `copytree()` вызывается рекурсивно, `ignore` вызывается 1 раз для каждой поддиректории. Она должна возвращать список объектов относительно текущего имени директории (т.е. подмножество элементов во втором аргументе). Эти объекты не будут скопированы.

`shutil.ignore_patterns`(*patterns)

- функция, которая создаёт функцию, которая может быть использована в качестве `ignore` для `copytree()`, игнорируя файлы и директории, которые соответствуют glob-style шаблонам.

Например:

```
copytree(source, destination,
          ignore=ignore_patterns('*.pyc', 'tmp*'))
# Скопирует все файлы, кроме заканчивающихся
                        на .pyc или начинающихся с tmp
```

`shutil.rmtree`(path, ignore_errors=False, onerror=None)

- Удаляет текущую директорию и все поддиректории; `path` должен указывать на директорию, а не на символическую ссылку.

Если `ignore_errors=True`, то ошибки, возникающие в результате неудавшегося удаления, будут проигнорированы. Если `False` (по умолчанию), эти ошибки будут передаваться обработчику `onerror`, или, если его нет, то исключение.

На ОС, которые поддерживают функции на основе файловых дескрипторов, по умолчанию используется версия `rmtree()`, не уязвимая к атакам на символические ссылки.

На других платформах это не так: при подобранном времени и обстоятельствах "хакер" может, манипулируя ссылками, удалить файлы, которые недоступны ему в других обстоятельствах.

Чтобы проверить, уязвима ли система к подобным атакам, можно использовать атрибут `rmtree.avoids_symlink_attacks`.

Если задан `onerror`, это должна быть функция с 3 параметрами: `function`, `path`, `excinfo`.

Первый параметр, `function`, это функция, которая создала исключение; она зависит от платформы и интерпретатора. Второй параметр, `path`, это путь, передаваемый функции. Третий параметр, `excinfo` - это информация об исключении, возвращаемая `sys.exc_info()`. Исключения, вызванные `onerror`, не обрабатываются.

shutil.move(src, dst, copy_function=copy2)

- рекурсивно перемещает файл или директорию (`src`) в другое место (`dst`), и возвращает место назначения.

Если `dst` - существующая директория, то `src` перемещается внутрь директории. Если `dst` существует, но не директория, то оно может быть перезаписано.

shutil.disk_usage(path)

- возвращает статистику использования дискового пространства как namedtuple с атрибутами total, used и free, в байтах.

shutil.chown(path, user=None, group=None)

- меняет владельца и/или группу у файла или директории.

shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)

- возвращает путь к исполняемому файлу по заданной команде.

Если нет соответствия ни с одним файлом, то None. `mode` это права доступа, требующиеся от файла, по умолчанию ищет только исполняемые.

Архивация

Высокоуровневые функции для созданиия и чтения архивированных и сжатых файлов. Основаны на функциях из модулей `zipfile` и `tarfile`.

shutil.make_archive(base_name, format[, root_dir[,
base_dir[, verbose[, dry_run[,
owner[, group[, logger]]]]]]])

- создаёт архив и возвращает его имя.

`base_name` это имя файла для создания, включая путь, но не включая расширения (не нужно писать ".zip" и т.д.).

`format` - формат архива.

`root_dir` - директория (относительно текущей), которую мы архивируем.

`base_dir` - директория, в которую будет архивироваться (т.е. все файлы в архиве будут в данной папке).

Если `dry_run=True`, архив не будет создан, но операции, которые должны были быть выполнены, запишутся в `logger`.

`owner` и `group` используются при создании tar-архива.

shutil.get_archive_formats()

- список доступных форматов для архивирования.

```
>>>
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"),
('gztar', "gzip'ed tar-file"),
```

```
('tar', 'uncompressed tar file'),
('xztar', "xz'ed tar-file"),
('zip', 'ZIP file')]
```

shutil.unpack_archive(filename[, extract_dir[, format]])

- распаковывает архив. `filename` - полный путь к архиву.

`extract_dir` - то, куда будет извлекаться содержимое (по умолчанию в текущую).

`format` - формат архива (по умолчанию пытается угадать по расширению файла).

shutil.get_unpack_formats() - список доступных форматов для разархивирования.

Запрос размера терминала вывода

shutil.get_terminal_size(fallback=(columns, lines))

- возвращает размер окна терминала.

`fallback` вернётся, если не удалось узнать размер терминала (терминал не поддерживает такие запросы, или программа работает без терминала). По умолчанию (80, 24).

```
>>>
>>> shutil.get_terminal_size()
os.terminal_size(columns=102, lines=29)
>>> shutil.get_terminal_size()  # Уменьшили окно
os.terminal_size(columns=67, lines=17)
```

XII. Примеры использования модуля shutil.

Копирование файла

Функция `shutil.copyfile()` копирует содержимое источника в место назначения и вызывает исключение `IOError`, если у него нет разрешения на запись в файл назначения.

```
>>> import shutil, os
>>> from glob import glob
# создадим временную директорию
>>> os.mkdir('example')
# создадим тестовый файл
>>> open('example/test_file.txt', 'w').close()
# копирование
>>> shutil.copyfile('example/test_file.txt',
'example/test_file.txt.copy')
# 'example/test-file.txt.copy'

# смотрим результат
>>> pprint.pprint(glob('example/*'))
```

```
# ['example/test_file.txt.copy',
'example/test_file.txt']

# удаляем
>>> shutil.rmtree('example')
```

Функция `shutil.copy()` интерпретирует имя выходного файла как инструмент командной строки Unix `cp`. *Если путь назначения указан как каталог, а не файл, то в каталоге создается новый файл с использованием его базового имени.*

```
>>> import shutil, os
>>> from glob import glob
# создадим тестовый файл
>>> open('shutil_copy.txt', 'w').close()
# создадим временную директорию
>>> os.mkdir('example')
>>> glob('example/*')
# []

# копируем
>>> shutil.copy('shutil_copy.txt', 'example')
# 'example/shutil_copy.txt'

# смотрим результат
>>> glob('example/*')
# ['example/shutil_copy.txt']

# удаляем
>>> shutil.rmtree('example')
```

Рекурсивное копирование каталога

Функция `shutil.copytree()` рекурсивно копирует весь каталог.

```
# для того что бы протестировать функцию copytree()
# создадим иерархию каталогов
import pathlib, itertools, glob, shutil
path = 'test/'
tmp = {'script':'.py', 'text':'.txt'}
for d, ext in tmp.items():
    comb = itertools.combinations([d, 1, 0], r=2)
    for a, b in comb:
        name = f'{a}{b}{ext}'
        pathlib.Path(path, d).mkdir(parents=True,
exist_ok=True)
```

```
        pathlib.Path(path, d,
name).touch(exist_ok=True)

f = glob.glob('test/**/*', recursive=True)
print(f)
# ['test/text', 'test/script', 'test/text/10.txt',
# 'test/text/text1.txt', 'test/text/text0.txt',
# 'test/script/10.py', 'test/script/script0.py',
# 'test/script/script1.py']

# копируем
shutil.copytree('test', 'test_cp')
# смотрим результат
f_cp = glob.glob('test_cp/**/*', recursive=True)
print(f_cp)

# ['test/text', 'test/script', 'test/text/10.txt',
# 'test/text/text1.txt', 'test/text/text0.txt',
# 'test/script/10.py', 'test/script/script0.py',
# 'test/script/script1.py']

# удаляем
shutil.rmtree('test_cp')
shutil.rmtree('test')
```

Выборочное рекурсивное копирование файлов каталога

Пример, который использует помощник `shutil.ignore_patterns()`.

Здесь копируется все, кроме файлов `.pyc` и файлов или каталогов, чье имя начинается с `tmp`.

```
from shutil import copytree, ignore_patterns
copytree(source, destination,
             ignore=ignore_patterns('*.pyc', 'tmp*'))
```

Другой пример, который использует аргумент ignore для добавления вызова логирования копирования файлов:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored
```

```
copytree(source, destination, ignore=_logpath)
```

Рекурсивное удаление каталога

Работа функции `shutil.rmtree()`, которая выполняет рекурсивное удаление каталога, демонстрировалась выше. Разберем ситуации посложнее.

В этом примере показано, как удалить дерево каталогов в Windows, где для некоторых файлов установлен бит только для чтения.

Он использует обратный вызов `onerror`, чтобы очистить бит `readonly` и повторить попытку удаления.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

В функции `shutil.rmtree()`, так же можно использовать помощник `shutil.ignore_patterns()` для **выборочного рекурсивного удаления файлов** как это делается в выборочном рекурсивном копировании файлов.

Пример реализации функции `shutil.copytree()`

Этот пример является реализацией функции `shutil.copytree()`, с опущенной строкой документации. Он демонстрирует многие другие функции, предоставляемые этим модулем.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
```

```
                    copy2(srcname, dstname)
                # XXX What about devices, sockets etc.?
            except OSError as why:
                errors.append((srcname, dstname, str(why)))
            # catch the Error from the recursive copytree so that we
can
            # continue with other files
            except Error as err:
                errors.extend(err.args[0])
        try:
            copystat(src, dst)
        except OSError as why:
            # can't copy file access times on Windows
            if why.winerror is None:
                errors.extend((src, dst, str(why)))
        if errors:
            raise Error(errors)
```

Архивирование каталогов

В этом примере создадим архив tar-файлов с использованием `gzip`, содержащий все файлы, найденные в каталоге `.ssh` пользователя:

```
from shutil import make_archive
import os
archive_name = os.path.expanduser(os.path.join('~',
                                               'myarchive'))
root_dir = os.path.expanduser(os.path.join('~',
                                           '.ssh'))
make_archive(archive_name, 'gztar', root_dir)

'/Users/docs-python/myarchive.tar.gz'
```

Полученный архив содержит:

```
~$ tar -tzvf /home/docs-python/myarchive.tar.gz

drwx------ docs-python/docs-python     0 2019-12-17 16:57 ./
-rw------- docs-python/docs-python  1554 2019-12-17 16:53 ./known_hosts
-rw------- docs-python/docs-python  1554 2019-12-17 14:06 ./known_hosts.old
-rw------- docs-python/docs-python  1679 2019-09-16 12:02 ./id_rsa
-rw-r--r-- docs-python/docs-python   399 2019-09-16 12:02 ./id_rsa.pub
```

XIII. Модуль pathlib

Python 3 включает модуль `pathlib` для манипуляции путями файловых систем независимо от операционной системы.

`pathlib` похож на модуль `os.path`, но `pathlib` предлагает более развитый и удобный интерфейс по сравнению с `os.path`.

Обычно идентифицируют файлы на компьютере с помощью иерархических путей.

Например, можно идентифицировать файл `wave.txt` на компьютере с помощью этого пути: `/Users/sammy/ocean/wave.txt`.

Операционные системы представляют пути несколько по-разному. Windows может представлять путь к файлу `wave.txt` как `C:\Users\sammy\ocean\wave.txt`.

Модуль `pathlib` может быть полезен, если в программе Python вы создаете или перемещаете файлы в файловой системе, указывая все файлы в файловой системе, совпадающие с данным расширением или шаблоном, или создаете пути файла, соответствующие файловой системе на основе наборов неформатированных строк.

Хотя вы можете использовать другие инструменты, например модуль `os.path`, для выполнения большей части этих задач, но модуль `pathlib` позволяет выполнять эти операции с большей степенью читаемости и минимальным количеством кодов.

Рассмотрим некоторые способы использования модуля `pathlib` для представления и манипуляции путями файловых систем.

Модуль `pathlib` предоставляет несколько классов, но одним из наиболее важных является класс `Path`.

Экземпляры класса `Path` представляют путь к файлу или каталогу в файловой системе вашего компьютера.

Например, следующий код получает экземпляр `Path`, который представляет часть пути к файлу `wave.txt`:

```
from pathlib import Path

wave = Path("ocean", "wave.txt")
print(wave)
```

Если запустить этот код, результат будет выглядеть следующим образом:

```
Output
ocean/wave.txt
```

`from pathlib import Path` делает класс `Path` доступным для нашей программы.

Затем `Path("ocean", "wave.txt")` получает новый экземпляр `Path`.

Из вывода результата видно, что Python «добавил» соответствующий разделитель оперативной системы `/` между двумя заданными нами компонентами пути `"ocean"` и `"wave.txt"`.

`Примечание.` В зависимости от операционной системы вывод может немного отличаться от примеров, приведенных в данном руководстве. В

Windows, например, вывод для этого примера может выглядеть как `ocean\wave.txt`.

В примере объект `Path`, присвоенный переменной `wave`, содержит ***относительный путь***.

Можно использовать `Path.home()` для получения абсолютного пути к домашнему каталогу текущего пользователя:

```
home = Path.home()
wave_absolute = Path(home, "ocean", "wave.txt")
print(home)
print(wave_absolute)
```

Если запустить этот код, результат будет выглядеть приблизительно следующим образом:

```
Output
/Users/sammy
/Users/sammy/ocean/wave.txt
```

Примечание. Как упоминалось ранее, вывод будет зависеть от операционной системы

`Path.home()` возвращает экземпляр `Path` с абсолютным путем в домашний каталог текущего пользователя.

Затем мы передадим этот экземпляр `Path` и строки `"ocean"` и `"wave.txt"` в другой конструктор `Path`, чтобы создать абсолютный путь к файлу `wave.txt`.

Вывод показывает, что первая строка — это домашний каталог, а вторая строка — домашний каталог плюс `ocean/wave.txt`.

Этот пример также иллюстрирует важную функцию класса `Path`: конструктор `Path` принимает обе строки и ранее существовавшие объекты `Path`.

Давайте более детально рассмотрим поддержку строк и объектов `Path` в конструкторе `Path`:

```
shark = Path(Path.home(), "ocean", "animals",
                          Path("fish", "shark.txt"))
print(shark)
```

Если запустить этот код Python, результат будет выглядеть следующим образом:

```
Output
/Users/sammy/ocean/animals/fish/shark.txt
```

`shark` — это `Path` к файлу, который мы создали с помощью объектов `Path` (`Path.home()` и `Path("fish", "shark.txt")` и строк `"ocean"` и `"animals"`).

Конструктор `Path` интеллектуально обрабатывает оба типа объектов и аккуратно соединяет их с помощью соответствующего разделителя операционной системы, в данном случае `/`.

Доступ к атрибутам файла

Рассмотрим, как можно использовать экземпляры `Path` для доступа к информации о файле.

Мы можем использовать атрибуты `name` и `suffix` для доступа к именам и расширениям файлов:

```
wave = Path("ocean", "wave.txt")
print(wave)
print(wave.name)
print(wave.suffix)
```

Запустив этот код и получим вывод, аналогичный следующему:

```
Output
/Users/sammy/ocean/wave.txt
wave.txt
.txt
```

Этот вывод показывает, что имя файла в конце нашего пути — `wave.txt`, а расширение файла — `.txt`.

Экземпляры `Path` также предлагают функцию `with_name`, позволяющую беспрепятственно создавать новый объект `Path` с другим именем:

```
wave = Path("ocean", "wave.txt")
tides = wave.with_name("tides.txt")
print(wave)
print(tides)
```

Если запустить его, результат будет выглядеть следующим образом:

```
ocean/wave.txt
ocean/tides.txt
```

Код сначала создает экземпляр `Path`, указывающий на файл с именем `wave.txt`.

Затем вызывается метод `with_name` в `wave`, чтобы вернуть второй экземпляр `Path`, указывающий на новый файл с именем `tides.txt`.

Часть каталога `ocean/` остается неизмененной и оставляет финальный путь в виде `ocean/tides.txt`

Доступ к предшествующим объектам

Иногда полезно получить доступ к каталогам, содержащим определенный путь. Давайте рассмотрим пример:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
print(shark)
print(shark.parent)
```

Если запустить этот код, результат будет выглядеть следующим образом:

```
Output
ocean/animals/fish/shark.txt
ocean/animals/fish
```

Атрибут **parent** в экземпляре `Path` возвращает ближайшего предшественника пути данного файла. В этом случае он возвращает каталог с файлом `shark.txt`: `ocean/animals/fish`.

Можем получать доступ к атрибуту `parent` несколько раз в команде, чтобы пройти вверх по корневому дереву данного файла:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
print(shark)
print(shark.parent.parent)
```

Если выполнить этот код, то увидим следующие:

```
Output
ocean/animals/fish/shark.txt
ocean/animals
```

Вывод будет похож на предыдущий вывод, но теперь мы перешли на уровень выше, получив доступ к `.parent` во второй раз. Два каталога от `shark.txt` — это каталог `ocean/animals`.

Использование шаблона поиска для списка файлов

Также можно использовать класс `Path` для списка файлов с помощью метода **glob**.

Допустим, у нас есть структура каталога, которая выглядит следующим образом:

```
└── ocean
    ├── animals
    │   └── fish
    │       └── shark.txt
    ├── tides.txt
    └── wave.txt
```

Каталог ocean содержит файлы tides.txt и wave.txt. У нас есть файл с именем shark.txt, вложенный в каталог ocean, каталог animals и каталог fish: ocean/animals/fish.

Чтобы перечислить все файлы .txt в каталоге ocean, можно использовать:

```
for txt_path in Path("ocean").glob("*.txt"):
    print(txt_path)
```

Этот код произведет следующий вывод:

```
Output
ocean/wave.txt
ocean/tides.txt
```

Шаблон поиска "*.txt" находит все файлы, заканчивающиеся на .txt. Поскольку пример кода выполняет этот поиск в каталоге ocean, он возвращает два файла .txt в каталоге ocean: wave.txt и tides.txt.

Также можно использовать *метод glob рекурсивно*.

Чтобы перечислить все файлы .txt в каталоге ocean и все его подкаталоги, запишем:

```
for txt_path in Path("ocean").glob("**/*.txt"):
    print(txt_path)
```

Если запустить этот код, результат будет выглядеть следующим образом:

```
Output
ocean/wave.txt
ocean/tides.txt
ocean/animals/fish/shark.txt
```

Часть ** шаблона поиска будет соответствовать этому каталогу и всем каталогам под ним рекурсивно.

Поэтому в выводе у нас будут не только файлы wave.txt и tides.txt, но также мы получим файл shark.txt, вложенный в ocean/animals/fish.

Вычисление относительных путей

Можно использовать метод Path.relative_to для вычисления путей, относящихся друг к другу. Метод relative_to полезен, если, например, вы хотите получить часть длинного пути файла.

Рассмотрите следующий код:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
below_ocean = shark.relative_to(Path("ocean"))
```

```
below_animals = shark.relative_to(Path("ocean",
                                       "animals"))
print(shark)
print(below_ocean)
print(below_animals)
```

Если запустить его, результат будет выглядеть следующим образом:

```
Output
ocean/animals/fish/shark.txt
animals/fish/shark.txt
fish/shark.txt
```

Метод `relative_to` возвращает новый объект `Path`, относящийся к данному аргументу. В нашем примере мы вычислим `Path` к `shark.txt`, относящийся к каталогу `ocean`, а затем относящийся к обоим каталогам `ocean` и `animals`.

Если `relative_to` не сможет вычислить ответ, поскольку мы даем ему не связанный путь, он выдаст `ValueError`:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
shark.relative_to(Path("unrelated", "path"))
```

Мы получим исключение `ValueError`, возникшее из этого кода, которое будет выглядеть следующим образом:

```
Output
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/Python3.8/pathlib.py", line 899, in
                                                    relative_to
    raise ValueError("{!r} does not start with {!r}"
ValueError: 'ocean/animals/fish/shark.txt' does not start with
                                                  'unrelated/path'
```

`unrelated/path` не является частью `ocean/animals/fish/shark.txt`, поэтому Python не сможет вычислить относительный путь.

Модуль `pathlib` представляет дополнительные классы и утилиты, Можно воспользоваться ссылкой на документацию модуля `pathlib` для получения дополнительной информации.

XIV. Модуль glob

Модуль glob находит все пути, совпадающие с заданным шаблоном в соответствии с правилами, используемыми оболочкой Unix.

Обрабатываются символы "*" (произвольное количество символов), "?" (один символ), и диапазоны символов с помощью [].

Для использования тильды "~" и переменных окружения необходимо использовать `os.path.expanduser()` и `os.path.expandvars()`.

Для поиска спецсимволов, заключайте их в квадратные скобки. Например, [?] соответствует символу "?".

`glob.glob`(pathname)

- возвращение список (возможно, пустой) путей, соответствующих шаблону pathname. Путь может быть как абсолютным (например, /usr/src/Python-1.5/Makefile) или относительный (как ../../Tools/*/*.gif).

`glob.iglob`(pathname)

- возвращает итератор, дающий те же значения, что и glob.glob.

`glob.escape`(pathname)

- экранирует все специальные символы для glob ("?", "*" и "["). Специальные символы в имени диска не экранируются (так как они там не учитываются), то есть в Windows escape("//?/c:/Quo vadis?.txt") возвращает "//?/c:/Quo vadis[?].txt".

Рассмотрим, например, каталог, содержащий только следующие файлы: 1.gif, 2.txt и card.gif.

glob.glob() вернёт следующие результаты. (обратите внимание, что любые ведущие компоненты пути сохраняются):

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

Если каталог содержит файлы, начинающиеся с ".", они не будут включаться по умолчанию. Рассмотрим, например, каталог, содержащий card.gif и .card.gif:

```
>>>
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

XV. Модуль os.path

os.path является вложенным модулем в модуль os, и реализует некоторые полезные функции на работы с путями.

os.path.abspath(path) - возвращает нормализованный абсолютный путь.

os.path.basename(path) - базовое имя пути (эквивалентно os.path.split(path)[1]).

os.path.commonprefix(list) - возвращает самый длинный префикс всех путей в списке.

os.path.dirname(path) - возвращает имя директории пути path.

os.path.exists(path) - возвращает True, если path указывает на существующий путь или дескриптор открытого файла.

os.path.expanduser(path) - заменяет ~ или ~user на домашнюю директорию пользователя.

os.path.expandvars(path) - возвращает аргумент с подставленными переменными окружения ($name или ${name} заменяются переменной окружения name). Несуществующие имена не заменяет. На Windows также заменяет %name%.

os.path.getatime(path) - время последнего доступа к файлу, в секундах.

os.path.getmtime(path) - время последнего изменения файла, в секундах.

os.path.getctime(path) - время создания файла (Windows), время последнего изменения файла (Unix).

os.path.getsize(path) - размер файла в байтах.

os.path.isabs(path) - является ли путь абсолютным.

os.path.isfile(path) - является ли путь файлом.

os.path.isdir(path) - является ли путь директорией.

os.path.islink(path) - является ли путь символической ссылкой.

os.path.ismount(path) - является ли путь точкой монтирования.

os.path.join(path1[, path2[, ...]]) - соединяет пути с учётом особенностей операционной системы.

os.path.normcase(path) - нормализует регистр пути (на файловых системах, не учитывающих регистр, приводит путь к нижнему регистру).

os.path.normpath(path) - нормализует путь, убирая избыточные разделители и ссылки на предыдущие директории. На Windows преобразует прямые слеши в обратные.

os.path.realpath(path) - возвращает канонический путь, убирая все символические ссылки (если они поддерживаются).

os.path.relpath(path, start=None) - вычисляет путь относительно директории start (по умолчанию - относительно текущей директории).

os.path.samefile(path1, path2) - указывают ли path1 и path2 на один и тот же файл или директорию.

os.path.sameopenfile(fp1, fp2) - указывают ли дескрипторы fp1 и fp2 на один и тот же открытый файл.

os.path.split(path) - разбивает путь на кортеж (голова, хвост), где хвост - последний компонент пути, а голова - всё остальное. Хвост никогда не начинается со слеша (если путь заканчивается слешем, то хвост пустой). Если слешей в пути нет, то пустой будет голова.

os.path.splitdrive(path) - разбивает путь на пару (привод, хвост).

os.path.splitext(path) - разбивает путь на пару (root, ext), где ext начинается с точки и содержит не более одной точки.

os.path.supports_unicode_filenames - поддерживает ли файловая система Unicode.

XVI. Модуль sys

Модуль **sys** обеспечивает доступ к некоторым переменным и функциям, взаимодействующим с интерпретатором python.

sys.argv - список аргументов командной строки, передаваемых сценарию Python. sys.argv[0] является именем скрипта (пустой строкой в интерактивной оболочке).

Примерчик вывести все аргументы командной строки:

```
for param in sys.argv:
     print (param)
```

sys.byteorder - порядок байтов. Будет иметь значение 'big' при порядке следования битов от старшего к младшему, и 'little', если наоборот (младший байт первый).

sys.builtin_module_names - кортеж строк, содержащий имена всех доступных модулей.

sys.call_tracing(функция, аргументы) - вызывает функцию с аргументами и включенной трассировкой, в то время как трассировка включена.

sys.copyright - строка, содержащая авторские права, относящиеся к интерпретатору Python.

sys._clear_type_cache() - очищает внутренний кэш типа.

sys._current_frames() - возвращает словарь-отображение идентификатора для каждого потока в верхнем кадре стека в настоящее время в этом потоке в момент вызова функции.

sys.dllhandle - целое число, определяющее дескриптор DLL Python (Windows).

sys.exc_info() - возвращает кортеж из трех значений, которые дают информацию об исключениях, обрабатывающихся в данный момент.

sys.exec_prefix - каталог установки Python.

sys.executable - путь к интерпретатору Python.

sys.exit([arg]) - ***выход из Python. Возбуждает исключение SystemExit, которое может быть перехвачено***.

sys.flags - флаги командной строки. Атрибуты только для чтения.

sys.float_info - информация о типе данных float.

sys.float_repr_style - информация о применении встроенной функции repr() для типа float.

sys.getdefaultencoding() - возвращает используемую кодировку.

sys.getdlopenflags() - значения флагов для вызовов dlopen().

sys.getfilesystemencoding() - возвращает кодировку файловой системы.

sys.getrefcount(object) - возвращает количество ссылок на объект. Аргумент функции getrefcount - еще одна ссылка на объект.

sys.getrecursionlimit() - возвращает лимит рекурсии.

sys.getsizeof(object[, default]) - возвращает размер объекта (в байтах).

sys.getswitchinterval() - интервал переключения потоков.

sys.getwindowsversion() - возвращает кортеж, описывающий версию Windows.

sys.hash_info - информация о параметрах хэширования.

sys.hexversion - версия python как шестнадцатеричное число (для 3.2.2 final это будет 30202f0).

sys.implementation - объект, содержащий информацию о запущенном интерпретаторе python.

sys.int_info - информация о типе int.

sys.intern(строка) - возвращает интернированную строку.

sys.last_type, sys.last_value, sys.last_traceback - информация об обрабатываемых исключениях. По смыслу похоже на sys.exc_info().

sys.maxsize - максимальное значение числа типа Py_ssize_t (2^{31} на 32-битных и 2^{63} на 64-битных платформах).

sys.maxunicode - максимальное число бит для хранения символа Unicode.

sys.modules - словарь имен загруженных модулей. Изменяем, поэтому можно позабавиться :)

sys.path - список путей поиска модулей.

sys.path_importer_cache - словарь-кэш для поиска объектов.

sys.platform - информация об операционной системе.

Linux (2.x and 3.x)	'linux'
Windows	'win32'
Windows/Cygwin	'cygwin'
Mac OS X	'darwin'
OS/2	'os2'
OS/2 EMX	'os2emx'

sys.prefix - папка установки интерпретатора python.

sys.ps1, sys.ps2 - первичное и вторичное приглашение интерпретатора (определены только если интерпретатор находится в интерактивном режиме). По умолчанию sys.ps1 == ">>> ", а sys.ps2 == "... ".

sys.dont_write_bytecode - если true, python не будет писать .pyc файлы.

sys.setdlopenflags(flags) - установить значения флагов для вызовов dlopen().

sys.setrecursionlimit(предел) - установить максимальную глубину рекурсии.

sys.setswitchinterval(интервал) - установить интервал переключения потоков.

sys.settrace(tracefunc) - установить "след" функции.

sys.stdin - стандартный ввод.

sys.stdout - стандартный вывод.

sys.stderr - стандартный поток ошибок.

sys.__stdin__, sys.__stdout__, sys.__stderr__ - исходные значения потоков ввода, вывода и ошибок.

sys.tracebacklimit - максимальное число уровней отслеживания.

sys.version - версия python.

sys.api_version - версия C API.

sys.version_info - Кортеж, содержащий пять компонентов номера версии.

sys.warnoptions - реализация предупреждений.

sys.winver - номер версии python, использующийся для формирования реестра Windows.

XVII. Модуль itertools

Модуль itertools - сборник полезных итераторов.

itertools.count(start=0, step=1)

- бесконечная арифметическая прогрессия с первым членом start и шагом step.

itertools.cycle(iterable)

- возвращает по одному значению из последовательности, повторенной бесконечное число раз.

itertools.repeat(elem, n=Inf)

- повторяет elem n раз.

itertools.accumulate(iterable) - аккумулирует суммы.

```
accumulate([1,2,3,4,5]) --> 1 3 6 10 15
```

itertools.chain(*iterables)

- возвращает по одному элементу из первого итератора, потом из второго, до тех пор, пока итераторы не кончатся.

itertools.combinations(iterable, [r])

- комбинации длиной r из iterable без повторяющихся элементов.

```
combinations('ABCD', 2) --> AB AC AD BC BD CD
```

itertools.combinations_with_replacement(iterable, r)

- комбинации длиной r из iterable с повторяющимися элементами.

```
combinations_with_replacement('ABCD', 2) -->
                    AA AB AC AD BB BC BD CC CD DD
```

itertools.compress(data, selectors) - (d[0] if s[0]), (d[1] if s[1]), ...

```
compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
```

itertools.dropwhile(func, iterable)

- элементы iterable, начиная с первого, для которого func вернула ложь.

```
dropwhile(lambda x: x < 5, [1,4,6,4,1]) --> 6 4 1
```

itertools.filterfalse(func, iterable)

- все элементы, для которых func возвращает ложь.

itertools.groupby(iterable, key=None)

- группирует элементы по значению. Значение получается применением функции key к элементу (если аргумент key не указан, то значением является сам элемент).

```
>>>
>>> from itertools import groupby
>>> things = [("animal", "bear"), ("animal", "duck"),
                                      ("plant", "cactus"),
...                           ("vehicle", "speed boat"),
                            ("vehicle", "school bus")]
>>> for key, group in groupby(things, lambda x: x[0]):
...     for thing in group:
...         print("A %s is a %s." % (thing[1], key))
...     print()
A bear is a animal.
A duck is a animal.

A cactus is a plant.

A speed boat is a vehicle.
A school bus is a vehicle.
```

itertools.islice(iterable[, start], stop[, step])

- итератор, состоящий из среза.
-

itertools.permutations(iterable, r=None)

- перестановки длиной r из iterable.

itertools.product(*iterables, repeat=1)

- аналог вложенных циклов.

```
product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
```

itertools.starmap(function, iterable)

- применяет функцию к каждому элементу последовательности (каждый элемент распаковывается).

```
starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
```

itertools.takewhile(func, iterable)

- элементы до тех пор, пока func возвращает истину.

```
takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
```

itertools.tee(iterable, n=2)

- кортеж из n итераторов.

itertools.zip_longest(*iterables, fillvalue=None)

- как встроенная функция zip, но берет самый длинный итератор, а более короткие дополняет fillvalue.

```
zip_longest('ABCD', 'xy', fillvalue='-') -->
                                        Ax By C- D-
```

XVIII. Модуль locale

При форматировании чисел Python по умолчанию использует англосаксонскую систему, при которой разряды целого числа отделяются друг от друга запятыми, а дробная часть от целой отделяется точкой. В континентальной Европе, например, используется другая система, при которой разряды разделяются точкой, а дробная и целая часть – запятой (https://metanit.com/python/tutorial/6.3.php):

```
# англосаксонская система
1,234.567
# европейская система
1.234,567
```

Для решения проблемы форматирования под определенную локаль в Python имеется встроенный модуль `locale`.

Для выяснения какая локаль установлена можно выполнить:

```
import local
locale.getlocale()
```

Например, вывод:

```
('Russian_Russia', '1251')
```

Для установки локали в модуле **locale** определена функция `setlocale()`. Она принимает два параметра:

```
setlocale(category, locale)
```

Первый параметр указывает на категорию, к которой применяется функция - к числам, валютам или и числам, и валютам. В качестве значения для параметра можно передавать одну из следующих констант:

- locale.LC_ALL: применяет локализацию ко всем категориям - к форматированию чисел, валют, дат и т.д.
- locale.LC_NUMERIC: применяет локализацию к числам
- locale.LC_MONETARY: применяет локализацию к валютам
- locale.LC_TIME: применяет локализацию к датам и времени
- locale.LC_CTYPE: применяет локализацию при переводе символов в верхний или нижний регистр
- locale.LC_COLLIATE: применяет локаль при сравнении строк

Второй параметр функции `setlocale` указывает на локаль, которую надо использовать.

На ОС Windows можно использовать код страны по ISO из двух символов:

для США - "us",
для Германии - "de",
для России - "ru".

Но на MacOS необходимо указывать код языка и код страны, например, для английского в США - "en_US", для немецкого в Германии - "de_DE", для русского в России - "ru_RU". По умолчанию фактически используется локаль "en_US".

Непосредственно для форматирования чисел и валют модуль `locale` предоставляет две функции:

- `currency(num)` - форматирует валюту
- `format(str, num)` - подставляет число `num` вместо плейсхолдера в строку `str`

Применяются следующие плейсхолдеры:

- `d`: для целых чисел
- `f`: для чисел с плавающей точкой
- `e`: для экспоненциальной записи чисел

Перед каждым плейсхолдером ставится знак процента %, например: `"%d"`

При выводе дробных чисел перед плейсхолдером после точки можно указать, сколько знаков в дробной части должно отображаться:

```
%.2f        # два знака в дробной части
```

Применим локализацию чисел и валют в немецкой локале:

```
import locale

locale.setlocale(locale.LC_ALL, "de") # для  Windows
# locale.setlocale(locale.LC_ALL, "de_DE")# для MacOS

number = 12345.6789
formatted = locale.format("%f", number)
print(formatted) # 12345,678900
```

```
formatted = locale.format("%.2f", number)
print(formatted) # 12345,68

formatted = locale.format("%d", number)
print(formatted) # 12345

formatted = locale.format("%e", number)
print(formatted) # 1,234568e+04

money = 234.678
formatted = locale.currency(money)
print(formatted)    # 234,68 €
```

Если вместо конкретного кода в качестве второго параметра функции `setlocale()` передаеть пустую строку, то Python будет использовать локаль, которая применяется на текущей рабочей машине.

С помощью функции `getlocale()` можно узнать текущую локаль:

```
import locale

locale.setlocale(locale.LC_ALL, "")

number = 12345.6789
formatted = locale.format("%.02f", number)
print(formatted)    # 12345,68
print(locale.getlocale())  # ('Russian_Russia', '1251')
```

XIX. Модуль datetime

Основной функционал для работы с датами и временем сосредоточен в модуле datetime в виде следующих классов (по материалам https://metanit.com/python/):

Класс date

Для работы с датами воспользуемся классом `date`, который определен в модуле `datetime`. Для создания объекта `date` мы можем использовать конструктор date, который последовательно принимает три параметра: год, месяц и день:

```
date(year, month, day)
```

Например, создадим какую-либо дату:

```
import datetime

yesterday = datetime.date(2017,5, 2)
print(yesterday)      # 2017-05-02
```

Если необходимо получить текущую дату, то можно воспользоваться методом `today()`:

```
from datetime import date

today = date.today()
print(today)       # 2017-05-03
print("{}.{}.{}".format(today.day, today.month,
                        today.year))      # 2.5.2017
```

С помощью свойств day, month, year можно получить соответственно день, месяц и год

Класс time

За работу с временем отвечает класс time. Используя его конструктор, можно создать объект времени:

```
time([hour] [, min] [, sec] [, microsec])
```

Конструктор последовательно принимает часы, минуты, секунды и микросекунды. Все параметры необязательные, и если мы какой-то параметр не передадим, то соответствующее значение будет инициализироваться нулем.

```
from datetime import time

current_time = time()
print(current_time)    # 00:00:00

current_time = time(16, 25)
print(current_time)    # 16:25:00

current_time = time(16, 25, 45)
print(current_time)    # 16:25:45
```

Класс datetime

Класс `datetime` из одноименного модуля объединяет возможности работы с датой и временем. Для создания объекта datetime можно использовать следующий конструктор:

```
datetime(year, month, day [, hour] [, min] [, sec]
                                        [, microsec])
```

Первые три параметра, представляющие год, месяц и день, являются обязательными. Остальные необязательные, и если мы не укажем для них значения, то по умолчанию они инициализируются нулем.

```
from datetime import datetime
deadline = datetime(2017, 5, 10)
print(deadline)      # 2017-05-10 00:00:00

deadline = datetime(2017, 5, 10, 4, 30)
```

```
print(deadline)     # 2017-05-10 04:30:00
```

Для получения текущих даты и времени можно вызвать метод `now()`:

```
from datetime import datetime

now = datetime.now()
print(now)      # 2017-05-03 11:18:56.239443

print("{}.{}.{}  {}:{}".format(now.day, now.month,
now.year, now.hour, now.minute))  # 3.5.2017  11:21

print(now.date())
print(now.time())
```

С помощью свойств `day, month, year, hour, minute, second` можно получить отдельные значения даты и времени. А через методы `date()` и `time()` можно получить отдельно дату и время соответственно.

Преобразование из строки в дату

Из функциональности класса datetime следует отметить метод strptime(), который позволяет распарсить строку и преобразовать ее в дату. Этот метод принимает два параметра:

```
strptime(str, format)
```

Первый параметр `str` представляет строковое определение даты и времени, а второй параметр - формат, который определяет, как различные части даты и времени расположены в этой строке.

Для определения формата можно использовать следующие коды:

- **%d**: день месяца в виде числа
- **%m**: порядковый номер месяца
- **%y**: год в виде 2-х чисел
- **%Y**: год в виде 4-х чисел
- **%H**: час в 24-х часовом формате
- **%M**: минута
- **%S**: секунда

Применим различные форматы:

```
from datetime import datetime
deadline = datetime.strptime("22/05/2017", "%d/%m/%Y")
print(deadline)     # 2017-05-22 00:00:00

deadline = datetime.strptime("22/05/2017 12:30",
"%d/%m/%Y %H:%M")
print(deadline)     # 2017-05-22 12:30:00

deadline = datetime.strptime("05-22-2017 12:30",
                             "%m-%d-%Y %H:%M")
print(deadline)     # 2017-05-22 12:30:00
```

Операции с датами

Фоматирование дат и времени

Для форматирования объектов date и time в обоих этих классах предусмотрен метод `strftime(format)`. Этот метод принимает только один параметр, указывающий на формат, в который нужно преобразовать дату или время.

Для определения формата можно использовать один из следующих кодов форматирования:

- %a: аббревиатура дня недели. Например, Wed - от слова Wednesday (по умолчанию используются английские наименования)
- %A: день недели полностью, например, Wednesday
- %b: аббревиатура названия месяца. Например, Oct (сокращение от October)
- %B: название месяца полностью, например, October
- %d: день месяца, дополненный нулем, например, 01
- %m: номер месяца, дополненный нулем, например, 05
- %y: год в виде 2-х чисел
- %Y: год в виде 4-х чисел
- %H: час в 24-х часовом формате, например, 13
- %I: час в 12-ти часовом формате, например, 01
- %M: минута
- %S: секунда
- %f: микросекунда
- %p: указатель AM/PM
- %c: дата и время, отформатированные под текущую локаль
- %x: дата, отформатированная под текущую локаль
- %X: время, форматированное под текущую локаль

Используем различные форматы:

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%Y-%m-%d"))       # 2017-05-03
print(now.strftime("%d/%m/%Y"))       # 03/05/2017
print(now.strftime("%d/%m/%y"))       # 03/05/17
print(now.strftime("%d %B %Y (%A)"))
                              # 03 May 2017 (Wednesday)
print(now.strftime("%d/%m/%y %I:%M"))  03/05/17 01:36
```

При выводе названий месяцев и дней недели по умолчанию используются английские наименования. Если мы хотим использовать текущую локаль, но то мы можем ее предварительно установить с помощью модуля `locale`:

```
from datetime import datetime
```

```
import locale
locale.setlocale(locale.LC_ALL, "ru")

now = datetime.now()
print(now.strftime("%d %B %Y (%A)"))
```

Результат выполнения скрипта:

```
06 Март 2018 (вторник)
```

Сложение и вычитани дат и времени

Нередко при работе с датами возникает необходимость добавить к какой-либо дате определенный промежуток времени или, наоборот, вычесть некоторый период. И специально для таких операций в модуле `datetime` определен класс `timedelta`. Фактически этот класс определяет некоторый период времени.

Для определения промежутка времени можно использовать конструктор `timedelta`:

```
timedelta([days] [, seconds] [, microseconds]
   [, milliseconds] [, minutes] [, hours] [, weeks])
```

В конструктор последовательно передаются дни, секунды, микросекунды, миллисекунды, минуты, часы и недели.

Определим несколько периодов:

```
from datetime import timedelta

three_hours = timedelta(hours=3)
print(three_hours)       # 3:00:00
three_hours_thirty_minutes =
              timedelta(hours=3, minutes=30)  # 3:30:00

two_days = timedelta(2)  # 2 days, 0:00:00

two_days_three_hours_thirty_minutes =
    timedelta(days=2, hours=3, minutes=30)
                                    # 2 days, 3:30:00
```

Используя `timedelta`, можно складывать или вычитать даты. Например, получим дату, которая будет через два дня:

```
from datetime import timedelta, datetime

now = datetime.now()
print(now)
two_days = timedelta(2)
in_two_days = now + two_days
print(in_two_days)
```

Вывод скрипта:

```
2018-03-06 12:20:48.524307
2018-03-08 12:20:48.524307
```

Или узнаем, сколько было времени 10 часов 15 минут назад, то есть фактически нам надо вычесть из текущего времени 10 часов и 15 минут:

```
from datetime import timedelta, datetime

now = datetime.now()
till_ten_hours_fifteen_minutes = now -
                        timedelta(hours=10, minutes=15)
print(till_ten_hours_fifteen_minutes)
```

Свойства timedelta

Класс `timedelta` имеет несколько свойств, с помощью которых мы можем получить временной промежуток:

- `days`: возвращает количество дней
- `seconds`: возвращает количество секунд
- `microseconds`: возвращает количество микросекунд

Кроме того, метод `total_seconds()` возвращает общее количество секунд, куда входят и дни, и собственно секунды, и микросекунды.

Например, узнаем какой временной период между двумя датами:

```
from datetime import timedelta, datetime

now = datetime.now()
twenty_two_may = datetime(2017, 5, 22)
period = twenty_two_may - now
print("{} дней  {} секунд   {} микросекунд".
                    format(period.days, period.seconds,
                                    period.microseconds))
# 18 дней  17537 секунд   72765 микросекунд

print("Всего: {} секунд".
                        format(period.total_seconds()))
# Всего: 1572737.072765 секунд
```

Сравнение дат

Также как и строки и числа, даты можно сравнивать с помощью стандартных операторов сравнения:

```
from datetime import datetime

now = datetime.now()
deadline = datetime(2017, 5, 22)
if now > deadline:
    print("Срок сдачи проекта прошел")
elif now.day == deadline.day and now.month ==
deadline.month and now.year == deadline.year:
    print("Срок сдачи проекта сегодня")
else:
```

```
    period = deadline - now
    print("Осталось {} дней".format(period.days))
```

XX. Модуль logging

Представьте ситуацию, когда необходимо сохранить некоторые отладочные или другие важные сообщения где-нибудь, чтобы иметь возможность позже проверить, отработала ли программа, как ожидалось. Как “сохранить где-нибудь” эти сообщения?

Сделать это можно при помощи модуля `logging`. (по материалам https://wombat.org.ua/AByteOfPython/standard_library.html)

```
import os, platform, logging

if platform.platform().startswith('Windows'):
   ogging_file = os.path.join(os.getenv('HOMEDRIVE'), \
                              os.getenv('HOMEPATH'), \
                              'test.log')
else:
    logging_file = os.path.join(os.getenv('HOME'),
                                          'test.log')

print("Сохраняем лог в", logging_file)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s : %(levelname)s : %(message)s',
    filename = logging_file,
    filemode = 'w',)

logging.debug("Начало программы")
logging.info("Какие-то действия")
logging.warning("Программа умирает")
```

Вывод**:**

$ python3 use_logging.py

Сохраняем лог в C:\Users\bsu\test.log

Если открыть файл test.log, он будет выглядеть примерно так:

```
2018-03-06 12:31:35,033 : DEBUG : Начало программы
2018-03-06 12:31:35,033 : INFO : Какие-то действия
2018-03-06 12:31:35,033 : WARNING : Программа умирает
```

Как это работает:

Использовались три модуля из стандартной библиотеки: модуль `os` для взаимодействия с операционной системой, модуль `platform` для получения информации о платформе (т.е. операционной системе) и модуль `logging` для сохранения лога.

Прежде всего, при помощи строки, возвращаемой функцией `platform.platform()` мы проверяем, какая операционная система используется (для более подробной информации см. **`import`** `platform; help(platform))`. Если это Windows, то мы определяем диск, содержащий домашний каталог, путь к домашнему каталогу на нём и имя файла, в котором хотим сохранить информацию. Сложив все эти три части, мы получаем полный путь к файлу. Для других платформ нам нужно знать только путь к домашнему каталогу пользователя, и мы получим полный путь к файлу.

При помощи функции `os.path.join()` мы объединяем три части пути к файлу вместе. Мы используем эту функцию вместо простого объединения строк для того, чтобы гарантировать, что полный путь к файлу записан в формате, ожидаемом операционной системой.

Далее мы конфигурируем модуль `logging` таким образом, чтобы он записывал все сообщения в определённом формате в указанный файл.

Наконец, мы можем выводить сообщения, предназначенные для отладки, информирования, предупреждения и даже критические сообщения. После выполнения программы можно просмотреть этот файл и узнать, что происходило в программе, хотя пользователю, запустившему программу, ничего не было показано.

XXI. Создание GUI на Python с помощью библиотеки Tkinter

(по материалам сайта https://younglinux.info/tkinter/text.php)

Введение в tkinter

В многообразии программ, которые пишут программисты, выделяют приложения с графическим пользовательским интерфейсом (GUI). При создании таких программ становятся важными не только алгоритмы обработки данных, но и разработка для пользователя программы удобного интерфейса, взаимодействуя с которым, он будет определять поведение приложения.

Современный пользователь в основном взаимодействует с программой с помощью различных кнопок, меню, значков, вводя информацию в специальные поля, выбирая определенные значения в списках и т. д. Эти "изображения" в определенном смысле и формируют GUI, в дальнейшем мы их будем называть виджетами (от англ. widget - "штучка").

Для языка программирования Python такие виджеты включены в специальную библиотеку — tkinter. Если ее импортировать в программу (скрипт), то можно пользоваться ее компонентами, создавая графический интерфейс.

Последовательность шагов при создании графического приложения имеет свои особенности. Программа должна выполнять свое основное назначение, быть удобной для пользователя, реагировать на его действия. Мы не будем вдаваться в подробности разработки, а рассмотрим какие этапы приблизительно нужно пройти при программировании, чтобы получить программу с GUI:

1. Импорт библиотеки
2. Создание главного окна
3. Создание виджет
4. Установка их свойств
5. Определение событий
6. Определение обработчиков событий
7. Расположение виджет на главном окне
8. Отображение главного окна

Импорт модуля tkinter

Как и любой модуль, `tkinter` в Python можно импортировать двумя способами: командами **`import`** `tkinter` или **`from`** `tkinter` **`import`** *.

В дальнейшем мы будем пользоваться только вторым способом, т. к. это позволит не указывать каждый раз имя модуля при обращении к объектам, которые в нем содержатся. Следует обратить внимание, что в версии Python 3 имя модуля пишется со строчной буквы (`tkinter`), хотя в более ранних версиях использовалась прописная (`Tkinter`). Итак, первая строчка программы должна выглядеть так:

```
from tkinter import *
```

Создание главного окна

В современных операционных системах любое пользовательское приложение заключено в окно, которое можно назвать главным, т.к. в нем располагаются все остальные виджеты. Объект окна верхнего уровня создается при обращении к классу Tk модуля `tkinter`. Переменную связанную с объектом-окном принято называть `root` (хотя понятно, что можно назвать как угодно, но так уж принято). Вторая строчка кода:

```
root = Tk()
```

Создание виджет

Допустим в окне будет располагаться всего одна кнопка. Кнопка создается при обращении к классу `Button` модуля `tkinter`. Объект кнопка связывается с какой-нибудь переменной. У класса `Button` (как и всех остальных классов, за исключением Tk) есть обязательный параметр — объект, которому кнопка принадлежит (кнопка не может "быть ничейной"). Пока у нас есть единственное окно (`root`), оно и будет аргументом, передаваемым в класс при создании объекта-кнопки:

```
but = Button(root)
```

Установка свойств виджет

У кнопки много свойств: размер, цвет фона и надписи и др. Мы рассмотрим их на следующем уроке. Пока же установим всего одно свойство — текст надписи (`text`):

```
but["text"] = "Печать"
```

Определение событий и их обработчиков

Многообразие событий и способов их обработки будет рассмотрено на следующих уроках. Здесь же просто коснемся данного вопроса в связи с потребностью.

Что же будет делать кнопка и в какой момент она это будет делать? Предположим, что задача кнопки вывести какое-нибудь сообщение в поток вывода, используя функцию print. Делать она это будет при нажатии на нее левой кнопкой мыши.

Действия (алгоритм), которые происходят при том или ином событии, могут быть достаточно сложным. Поэтому часто их оформляют в виде функции, а затем вызывают, когда они понадобятся. Пусть у нас печать на экран будет оформлена в виде функции `printer`:

```
def printer(event):
    print ("Как всегда очередной 'Hello World!'")
```

Не забывайте, что функцию желательно (почти обязательно) размещать в начале кода. **Параметр `event` – это какое-либо событие.**

Событие нажатия левой кнопкой мыши выглядит так: `<Button-1>`. Требуется связать это событие с обработчиком (функцией `printer`). Для связи предназначен метод `bind`. Синтаксис связывания события с обработчиком выглядит так:

```
but.bind("<Button-1>",printer)
```

Событие – именился размер окна:

```
# обработчик res  закрытия окна
root.bind('<Configure>',res)
```

В обработчике можно получить новый размер графического окна:

```
    str=root.geometry()
    k1=str.index('x')
    xmax1=str[:k1]
    k2=str.index('+',k1+1)
    ymax1=str[k1+1:k2]
    xmax=int(xmax1)
    ymax=int(ymax1)
```

Событие – закрыть окно:

```
# обработчик window_deleted  закрытия окна
root.protocol('WM_DELETE_WINDOW', window_deleted)
```

Размещение виджет

Если вы заметите, то в любом приложении виджеты не разбросаны по окну как попало, а хорошо организованы, интерфейс продуман до мелочей и обычно подчинен определенным стандартам. До стандартов нам далеко, нужно просто кнопку как-то отобразить в окне. Самый простой способ — это использование метода `pack`.

```
but.pack()
```

Если не вставить эту строчку кода, то кнопка в окне так и не появится, хотя она есть в программе.

Отображение главного окна

Ну и наконец, главное окно тоже не появится, пока не будет вызван специальный метод `mainloop`:

```
root.mainloop()
```

Данная строчка кода должна быть всегда в конце скрипта!

В итоге, код программы может выглядеть таким образом:

```
from tkinter import *

def printer(event):
     print ("Как всегда очередной 'Hello World!'")

root = Tk()
but = Button(root)
but["text"] = "Печать"
but.bind("<Button-1>",printer)

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

При программировании графического интерфейса пользователя более эффективным оказывается объектно-ориентированный подход. Поэтому многие «вещи» оформляются в виде классов. В нашем примере также можно использовать класс:

```
from tkinter import *

class But_print:
     def __init__(self):
          self.but = Button(root)
          self.but["text"] = "Печать"
          self.but.bind("<Button-1>",self.printer)
          self.but.pack()
     def printer(self,event):
          print ("Как всегда очередной 'Hello World!'")
```

```
root = Tk()
obj = But_print()
root.mainloop()
```

Разметка виджетов в Tkinter — pack, grid и place

Познакомимся с **менеджерами разметки**. Когда мы **создаем графический интерфейс** нашего приложения, мы определяем, какие виджеты будем использовать, и как они будут расположены в приложении. Для того, чтобы организовать виджеты в приложении, используются специальные невидимые объекты – **менеджеры разметки**.

Существует два вида виджетов:

- контейнеры;
- дочерние виджеты.

Контейнеры объединяют виджеты для формирования разметки. У Tkinter есть три встроенных менеджера разметки: **pack**, **grid** и **place**.

- **Place** – это менеджер геометрии, который размещает виджеты, используя абсолютное позиционирование.
- **Pack** – это менеджер геометрии, который размещает виджеты по горизонтали и вертикали.
- **Grid** – это менеджер геометрии, который размещает виджеты в двухмерной сетке.

Метод place() в Tkinter — Абсолютное позиционирование

В большинстве случаев разработчикам необходимо использовать **менеджеры разметки**. Есть несколько ситуаций, в которых следует использовать именно **абсолютное позиционирование**. В рамках абсолютного позиционирования разработчик определяет позицию и размер каждого виджета в пикселях. Во время изменения размеров окна размер и позиция виджетов не меняются.

Изображения для примера: bardejov.jpg rotunda.jpg mincol.jpg сохраните в папке рядом с файлом `absolute.py` код для которого будет ниже.

Для работы этого скрипта необходимо установить пакет Image:

```
python -m pip install Image
```

Таким образом, на разных платформах приложения выглядят по-разному. То, что выглядит нормально на **Linux**, может отображаться некорректно на

Mac OS. Изменение шрифтов в нашем приложении также может испортить разметку. Если мы переведем наше приложение на другой язык, мы должны доработать и разметку.

absolute.py

```
from PIL import Image, ImageTk
from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Label, Style

class Example(Frame):

    def __init__(self):
        super().__init__()

        self.initUI()


    def initUI(self):

        self.master.title("Absolute positioning")
        self.pack(fill=BOTH, expand=1)

        Style().configure("TFrame", background="#333")

        bard = Image.open("bardejov.jpg")
        bardejov = ImageTk.PhotoImage(bard)
        label1 = Label(self, image=bardejov)
        label1.image = bardejov
        label1.place(x=20, y=20)

        rot = Image.open("rotunda.jpg")
        rotunda = ImageTk.PhotoImage(rot)
        label2 = Label(self, image=rotunda)
        label2.image = rotunda
        label2.place(x=40, y=160)

        minc = Image.open("mincol.jpg")
        mincol = ImageTk.PhotoImage(minc)
        label3 = Label(self, image=mincol)
        label3.image = mincol
        label3.place(x=170, y=50)


def main():

    root = Tk()
```

```
    root.geometry("300x280+300+300")
    app = Example()
    root.mainloop()


if __name__ == '__main__':
    main()
```

В этом примере мы расположили три изображения при помощи абсолютного позиционирования. ***Мы использовали менеджер геометрии place.***

```
from PIL import Image, ImageTk
```

Мы использовали **Image** и **ImageTk** из модуля **PIL** (Python Imaging Library).

```
style = Style()
style.configure("TFrame", background="#333")
```

При помощи стилей, мы изменили фон нашего окна на темно-серый.

```
bard = Image.open("bardejov.jpg")
bardejov = ImageTk.PhotoImage(bard)
```

Мы *создали объект изображения* и объект фото изображения из сохраненных ранее изображений в текущей рабочей директории.

```
label1 = Label(self, image=bardejov)
```

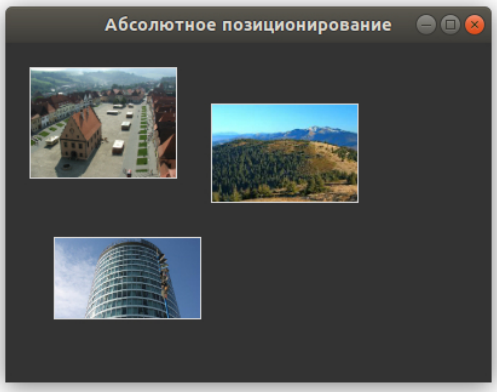
Мы создали `Label` (ярлык) с изображением. Данные ярлыки могут содержать как изображения, так и текст.

```
label1.image = bardejov
```

Нам нужно сохранить ссылку на изображение, чтобы не потерять его если сборщик мусора (Garbage collector) его не закроет.

```
label1.place(x=20, y=20)
```

Ярлык размещен в рамке по координатам `x=20` и `y=20`.



Tkinter pack() — размещение виджетов по горизонтали и вертикали

Менеджер геометрии `pack()` упорядочивает виджеты в горизонтальные и вертикальные блоки. Макетом можно управлять с помощью параметров `fill`, `expand` и `side`.

Пример создания кнопок в Tkinter

В следующем примере мы разместим две кнопки в нижнем правом углу нашего окна. Для этого мы воспользуемся менеджером **pack**.

```
from tkinter import Tk, RIGHT, BOTH, RAISED
from tkinter.ttk import Frame, Button, Style


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title("Кнопки в kinter")
        self.style = Style()
        self.style.theme_use("default")

        frame = Frame(self, relief=RAISED, borderwidth=1)
        frame.pack(fill=BOTH, expand=True)

        self.pack(fill=BOTH, expand=True)

        closeButton = Button(self, text="Закрыть")
        closeButton.pack(side=RIGHT, padx=5, pady=5)
        okButton = Button(self, text="Готово")
        okButton.pack(side=RIGHT)


def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example()
    root.mainloop()


if __name__ == '__main__':
    main()
```

У нас есть две рамки. **Первая рамка** – основная, а также вторая – дополнительная, которая растягивается в обе стороны и сдвигает две кнопки в нижнюю часть основной рамки. **Кнопки** находятся в горизонтальном контейнере и размещены в ее правой части.

```
frame = Frame(self, relief=RAISED, borderwidth=1)
frame.pack(fill=BOTH, expand=True)
```

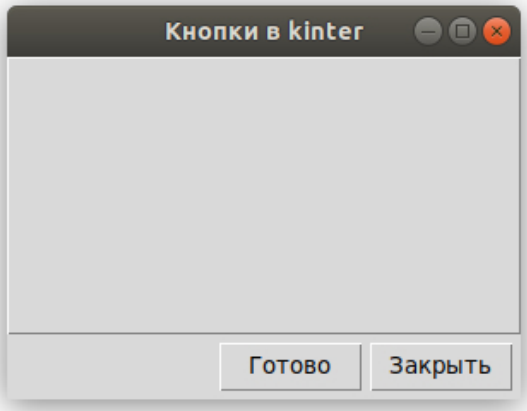
Мы создали еще один **виджет Frame**. Этот виджет занимает практически все пространство окна. Мы изменяем границы рамки, чтобы сама рамка была видна. По умолчанию она плоская.

```
closeButton = Button(self, text="Закрыть")
closeButton.pack(side=RIGHT, padx=5, pady=5)
```

Кнопка **closeButton** создана. Она расположена в горизонтальном контейнере. Параметр **side** позволяет поместить кнопку в правой части горизонтальной полосы. Параметры `padx` и `pady` позволяют **установить отступ** между виджетами. Параметр `padx` устанавливает пространство между виджетами кнопки `closeButton` и правой границей корневого окна.

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

Кнопка **okButton** размещена возле **closeButton** с установленным отступом (padding) в 5 пикселей.



Создаем приложение для отзывов на Tkinter

Менеджер pack – это простой **менеджер разметки**. Его можно использовать для простых задач разметки. Чтобы создать более сложную разметку, необходимо использовать больше рамок, каждая из которых имеет собственный менеджер разметки.

`review.py`

```
from tkinter import Tk, Text, BOTH, X, N, LEFT
from tkinter.ttk import Frame, Label, Entry


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()
```

```
    def initUI(self):
        self.master.title("Оставить отзыв")
        self.pack(fill=BOTH, expand=True)

        frame1 = Frame(self)
        frame1.pack(fill=X)

        lbl1 = Label(frame1, text="Заголовок", width=10)
        lbl1.pack(side=LEFT, padx=5, pady=5)

        entry1 = Entry(frame1)
        entry1.pack(fill=X, padx=5, expand=True)

        frame2 = Frame(self)
        frame2.pack(fill=X)

        lbl2 = Label(frame2, text="Автор", width=10)
        lbl2.pack(side=LEFT, padx=5, pady=5)

        entry2 = Entry(frame2)
        entry2.pack(fill=X, padx=5, expand=True)

        frame3 = Frame(self)
        frame3.pack(fill=BOTH, expand=True)

        lbl3 = Label(frame3, text="Отзыв", width=10)
        lbl3.pack(side=LEFT, anchor=N, padx=5, pady=5)

        txt = Text(frame3)
        txt.pack(fill=BOTH, pady=5, padx=5, expand=True)


def main():

    root = Tk()
    root.geometry("300x300+300+300")
    app = Example()
    root.mainloop()


if __name__ == '__main__':
    main()
```

На этом примере видно, как можно создать **более сложную разметку** с многочисленными рамками и **менеджерами pack()**.

```
1
self.pack(fill=BOTH, expand=True)
```

Первая рамка является базовой. На ней располагаются все остальные рамки. Стоит отметить, что даже при организации дочерних виджетов в рамках, мы управляем ими на базовой рамке.

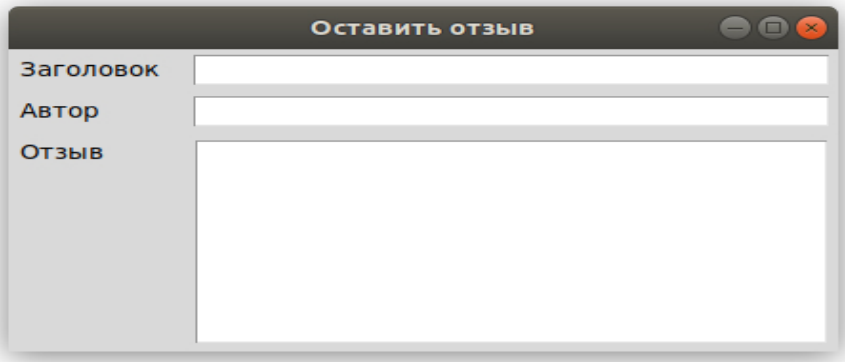
```
frame1 = Frame(self)
frame1.pack(fill=X)

lbl1 = Label(frame1, text="Заголовок", width=10)
```

```
lbl1.pack(side=LEFT, padx=5, pady=5)

entry1 = Entry(frame1)
entry1.pack(fill=X, padx=5, expand=True)
```

Первые два виджета размещены на первой рамке. Поле для ввода данных растянуто горизонтально с параметрами **fill** и **expand**.

```
frame3 = Frame(self)
frame3.pack(fill=BOTH, expand=True)

lbl3 = Label(frame3, text="Отзыв", width=10)
lbl3.pack(side=LEFT, anchor=N, padx=5, pady=5)

txt = Text(frame3)
txt.pack(fill=BOTH, pady=5, padx=5, expand=True)
```

В третьей рамке мы разместили ярлык и виджет для ввода текста. Ярлык закреплен по северной стороне `anchor=N`, а виджет текста занимает все остальное пространство.



Разметка grid() в Tkinter для создания калькулятора

Менеджер геометрии grid() в Tkinter используется для создания сетки кнопок для калькулятора.

calculator.py

```
from tkinter import Tk, W, E
from tkinter.ttk import Frame, Button, Entry, Style


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title("Калькулятор на Tkinter")
```

```
Style().configure("TButton", padding=(0, 5, 0, 5),
                                        font='serif 10')

self.columnconfigure(0, pad=3)
self.columnconfigure(1, pad=3)
self.columnconfigure(2, pad=3)
self.columnconfigure(3, pad=3)

self.rowconfigure(0, pad=3)
self.rowconfigure(1, pad=3)
self.rowconfigure(2, pad=3)
self.rowconfigure(3, pad=3)
self.rowconfigure(4, pad=3)

entry = Entry(self)
entry.grid(row=0, columnspan=4, sticky=W+E)
cls = Button(self, text="Очистить")
cls.grid(row=1, column=0)
bck = Button(self, text="Удалить")
bck.grid(row=1, column=1)
lbl = Button(self)
lbl.grid(row=1, column=2)
clo = Button(self, text="Закрыть")
clo.grid(row=1, column=3)
sev = Button(self, text="7")
sev.grid(row=2, column=0)
eig = Button(self, text="8")
eig.grid(row=2, column=1)
nin = Button(self, text="9")
nin.grid(row=2, column=2)
div = Button(self, text="/")
div.grid(row=2, column=3)

fou = Button(self, text="4")
fou.grid(row=3, column=0)
fiv = Button(self, text="5")
fiv.grid(row=3, column=1)
six = Button(self, text="6")
six.grid(row=3, column=2)
mul = Button(self, text="*")
mul.grid(row=3, column=3)

one = Button(self, text="1")
one.grid(row=4, column=0)
two = Button(self, text="2")
two.grid(row=4, column=1)
thr = Button(self, text="3")
thr.grid(row=4, column=2)
mns = Button(self, text="-")
mns.grid(row=4, column=3)

zer = Button(self, text="0")
zer.grid(row=5, column=0)
```

```
            dot = Button(self, text=".")
            dot.grid(row=5, column=1)
            equ = Button(self, text="=")
            equ.grid(row=5, column=2)
            pls = Button(self, text="+")
            pls.grid(row=5, column=3)

            self.pack()


    def main():
        root = Tk()
        app = Example()
        root.mainloop()


    if __name__ == '__main__':
        main()
```

Менеджер **grid()** используется для организации кнопок в контейнере рамки.

```
Style().configure("TButton", padding=(0, 5, 0, 5),
    font='serif 10')
```

Мы настроили **виджет кнопки** так, чтобы отображался специфический шрифт и применялся отступ (padding) в 3 пикселя.

```
self.columnconfigure(0, pad=3)
...
self.rowconfigure(0, pad=3)
```

Мы использовали методы **columnconfigure**() и **rowconfigure**() чтобы создать определенное пространство в сетке строк и столбцов. Благодаря этому шагу мы разделяем кнопки определенным пустым пространством.

```
entry = Entry(self)
entry.grid(row=0, columnspan=4, sticky=W+E)
```

Виджет графы ввода – это место, где будут отображаться цифры. Данный виджет расположен в первом ряду и охватывает все четыре столбца. Виджеты могут не занимать все пространство, которое выделяется клетками в созданной сетке.

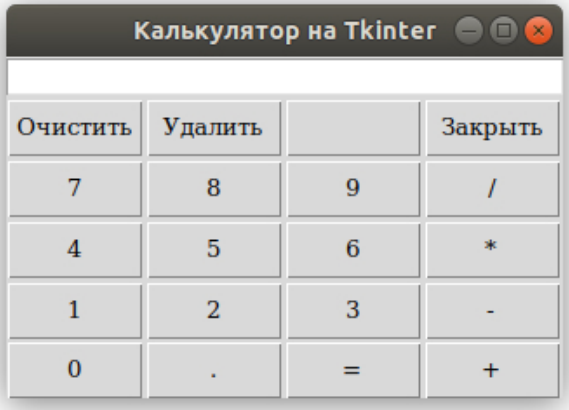
Параметр **sticky** расширяет виджет в указанном направлении. В нашем случае, мы можем убедиться, что наш виджет графы ввода был расширен слева направо **W+E** (восток-запад).

```
cls = Button(self, text="Очистить")
cls.grid(row=1, column=0)
```

Кнопка очистки установлена во второй строке и первом столбце. Стоит отметить, что строки и столбцы начинаются с нуля.

```
self.pack()
```

Метод pack() показывает виджет рамки и дает ей первоначальный размер. Если дополнительные параметры не указываются, размер будет таким, чтобы все дочерние виджеты могли поместиться. Этот метод компонует виджет рамки в верхнем корневом окне, которое также является контейнером. Менеджер **grid()** используется для организации кнопок в виджете рамки.



Пример создания диалогового окна в Tkinter

Следующий пример **создает диалоговое окно**, используя менеджер геометрии grid.

windows.py

```
from tkinter import Tk, Text, BOTH, W, N, E, S
from tkinter.ttk import Frame, Button, Label, Style


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()


    def initUI(self):

        self.master.title("Диалоговое окно в Tkinter")
        self.pack(fill=BOTH, expand=True)

        self.columnconfigure(1, weight=1)
        self.columnconfigure(3, pad=7)
        self.rowconfigure(3, weight=1)
        self.rowconfigure(5, pad=7)

        lbl = Label(self, text="Окна")
        lbl.grid(sticky=W, pady=4, padx=5)

        area = Text(self)
```

```
        area.grid(row=1, column=0, columnspan=2, rowspan=4,
                                          padx=5, sticky=E+W+S+N)

        abtn = Button(self, text="Активир.")
        abtn.grid(row=1, column=3)

        cbtn = Button(self, text="Закрыть")
        cbtn.grid(row=2, column=3, pady=4)

        hbtn = Button(self, text="Помощь")
        hbtn.grid(row=5, column=0, padx=5)

        obtn = Button(self, text="Готово")
        obtn.grid(row=5, column=3)


def main():

    root = Tk()
    root.geometry("350x300+300+300")
    app = Example()
    root.mainloop()


if __name__ == '__main__':
    main()
```

В этом примере мы использовали **виджет ярлыка**, **текстовой виджет** и четыре кнопки.

```
self.columnconfigure(1, weight=1)
self.columnconfigure(3, pad=7)
self.rowconfigure(3, weight=1)
self.rowconfigure(5, pad=7)
```

Мы добавили небольшое пространство между виджетами в сетке. Параметр **weight** создает возможность расширения второго столбца и четвертого ряда. В этом ряду и столбце находится **текстовой виджет**, поэтому оставшееся пространство заполняет данный виджет.

```
lbl = Label(self, text="Окна")
lbl.grid(sticky=W, pady=4, padx=5)
```

Виджет ярлыка также создается и *помещается в сетку*. Если не указываются ряд и столбец, тогда он займет первый ряд и столбец. Ярлык закрепляется у западной части окна `sticky=W` и имеет определенные отступы вокруг своих границ.

```
area = Text(self)
area.grid(row=1, column=0, columnspan=2, rowspan=4,
    padx=5, sticky=E+W+S+N)
```

Создается **текстовый виджет** и помещается во второй ряд и первый столбец. Он охватывает два столбца и четыре строки.

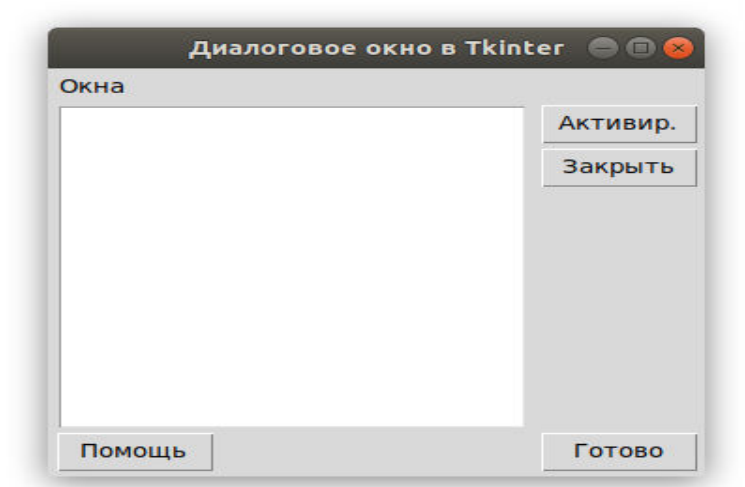
Между виджетом и левым краем корневого окна присутствует пространство в 4 пикселя. Также, виджет закреплен около всех четырех сторон. Поэтому, когда окно расширяется, виджеты текстов увеличиваются во всех направлениях.

```
abtn = Button(self, text="Активир.")
abtn.grid(row=1, column=3)

cbtn = Button(self, text="Закрыть")
cbtn.grid(row=2, column=3, pady=4)
```

Эти две кнопки находятся возле текстового виджета.

```
hbtn = Button(self, text="Помощь")
hbtn.grid(row=5, column=0, padx=5)

obtn = Button(self, text="Готово")
obtn.grid(row=5, column=3)
```

Эти две кнопки находятся под текстовым виджетом. Кнопка «Помощь» расположена в первом столбце, а кнопка «Готово» в последнем столбце.



XXII. Виджеты (графические объекты) и их свойства.

Рассмотрим часть графических объектов (виджет), содержащихся в библиотеке `Tkinter`: **кнопки, поля для ввода, метки, флажки, переключатели и списки.** Следует понимать, что графический интерфейс пользователя достаточно стандартен, и поэтому любые подобные

библиотеки-модули (в том числе и Tkinter) содержат приблизительно одинаковые виджеты.

Каждый класс виджет имеет определенные свойства, значения которых можно задавать при их создании, а также программировать их изменение при действии пользователя и в результате выполнения программы.

Кнопки

Объект-кнопка создается вызовом класса Button модуля tkinter. При этом обязательным аргументом является лишь родительский виджет (например, окно верхнего уровня). Другие свойства могут указываться при создании кнопки или задаваться (изменяться) позже. Синтаксис:

```
переменная = Button (родит_виджет,
                                [свойство=значение, … ….])
```

У кнопки много свойств, в примере ниже указаны лишь некоторые из них.

```
from tkinter import *

root = Tk()

but = Button(root,
           text="Это кнопка", #надпись на кнопке
           width=30,height=5, #ширина и высота
           bg="white",fg="blue") #цвет фона и надписи

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

bg и fg – это сокращения от background (фон) и foreground (передний план). Ширина и высота измеряются в знакоместах (количество символов).

Метки

Метки (или надписи) — это достаточно простые виджеты, содержащие строку (или несколько строк) текста и служащие в основном для информирования пользователя.

```
lab = Label(root, text="Это метка! \n Из двух строк.",
                                    font="Arial 18")
```

Однострочное текстовое поле

Такое поле создается вызовом класса Entry модуля tkinter. В него пользователь может ввести только одну строку текста.

```
ent = Entry(root,width=20,bd=3)
```

bd – это сокращение от borderwidth (ширина границы).

Элемент Entry представляет поле для ввода текста. Конструктор Entry принимает следующие параметры:

```
Entry(master, options)
```

Где master - ссылка на родительское окно, а options - набор следующих параметров:

- bg: фоновый цвет
- bd: толщина границы
- cursor: курсор указателя мыши при наведении на текстовое поле
- fg: цвет текста
- font: шрифт текста
- justify: устанавливает выравнивание текста. Значение LEFT выравнивает текст по левому краю, CENTER - по центру, RIGHT - по правому краю
- relief: определяет тип границы, по умолчанию значение FLAT
- selectbackground: фоновый цвет выделенного куска текста
- selectforeground: цвет выделенного текста
- show: задает маску для вводимых символов
- state: состояние элемента, может принимать значения NORMAL (по умолчанию) и DISABLED
- textvariable: устанавливает привязку к элементу StringVar
- width: ширина элемента

Определим элемент Entry и по нажатию на кнопку выведем его текст в отдельное окно с сообщением:

```
from tkinter import *
from tkinter import messagebox

def show_message():
    messagebox.showinfo("GUI Python", message.get())

root = Tk()
root.title("GUI на Python")
root.geometry("300x250")

message = StringVar()

message_entry = Entry(textvariable=message)
message_entry.place(relx=.5, rely=.1, anchor="c")

message_button = Button(text="Click Me", command=show_message)
message_button.place(relx=.5, rely=.5, anchor="c")

root.mainloop()
```

Для вывода сообщения здесь применяется дополнительный модуль messagebox, содержащий функцию showinfo(), которая собственно и выводит введенный в текстовое поле текст. Для получения введенного текста используется комп StringVar.

Теперь создадим более сложный пример с формой ввода:

```
from tkinter import *
from tkinter import messagebox


def display_full_name():
    messagebox.showinfo("GUI Python", name.get() + " " +
surname.get())

root = Tk()
root.title("GUI на Python")

name = StringVar()
surname = StringVar()

name_label = Label(text="Введите имя:")
surname_label = Label(text="Введите фамилию:")

name_label.grid(row=0, column=0, sticky="w")
surname_label.grid(row=1, column=0, sticky="w")

name_entry = Entry(textvariable=name)
surname_entry = Entry(textvariable=surname)

name_entry.grid(row=0,column=1, padx=5, pady=5)
surname_entry.grid(row=1,column=1, padx=5, pady=5)


message_button = Button(text="Click Me",
command=display_full_name)
message_button.grid(row=2,column=1, padx=5, pady=5, sticky="e")

root.mainloop()
```

Методы Entry

Элемент Entry имеет ряд методов. Основные из них:

- insert(index, str): вставляет в текстовое поле строку по определенному индексу
- get(): возвращает введенный в текстовое поле текст
- delete(first, last=None): удаляет символ по индексу first. Если указан параметр last, то удаление производится до индекса last. Чтобы удалить до конца, в качестве второго параметра можно использовать значение END.

Используем методы в программе:

```
from tkinter import *
from tkinter import messagebox


def clear():
    name_entry.delete(0, END)
```

```
    surname_entry.delete(0, END)


def display():
    messagebox.showinfo("GUI Python", name_entry.get() + " " +
surname_entry.get())

root = Tk()
root.title("GUI на Python")

name_label = Label(text="Введите имя:")
surname_label = Label(text="Введите фамилию:")

name_label.grid(row=0, column=0, sticky="w")
surname_label.grid(row=1, column=0, sticky="w")

name_entry = Entry()
surname_entry = Entry()

name_entry.grid(row=0,column=1, padx=5, pady=5)
surname_entry.grid(row=1,column=1, padx=5, pady=5)

# вставка начальных данных
name_entry.insert(0, "Tom")
surname_entry.insert(0, "Soyer")

display_button = Button(text="Display", command=display)
clear_button = Button(text="Clear", command=clear)

display_button.grid(row=2, column=0, padx=5, pady=5, sticky="e")
clear_button.grid(row=2, column=1, padx=5, pady=5, sticky="e")

root.mainloop()
```

При запуске программы сразу же в оба текстовых поля добавляется текст по умолчанию:

```
name_entry.insert(0, "Tom")
surname_entry.insert(0, "Soyer")
```

```
name_entry.insert("end", "Tom")
```

А так можно вставить текст «в конец поля»

Кнопка Clear очищает оба поля, вызывая метод delete:

```
def clear():
    name_entry.delete(0, END)
    surname_entry.delete(0, END)
```

Вторая кнопка, используя метод get, получает введенный текст:

```
def display():
```

```
    messagebox.showinfo("GUI Python", name_entry.get()
                        + " " + surname_entry.get())
```

Причем, как видно из примера, нам необязательно обращаться к тексту в Entry через переменные типа StringVar, мы можем это сделать напрямую через метод get

Многострочное текстовое поле

Text предназначен для предоставления пользователю возможности ввода не одной строки текста, а существенно больше.

```
tex = Text(root,width=40,
           font="Verdana 12",
           wrap=WORD)
```

Последнее свойство (wrap) в зависимости от своего значения позволяет переносить текст, вводимый пользователем либо по символам, либо по словам, либо вообще не переносить, пока пользователь не нажмет Enter.

Радиокнопки (переключатели)

Объект-радиокнопка никогда не используется по одному. Их используют группами, при этом в одной группе может быть «включена» лишь одна кнопка.

```
var=IntVar()
var.set(1)
rad0 = Radiobutton(root,text="Первая",
           variable=var,value=0)
rad1 = Radiobutton(root,text="Вторая",
           variable=var,value=1)
rad2 = Radiobutton(root,text="Третья",
           variable=var,value=2)
```

Одна группа определяет значение одной переменной, т. е. если в примере будет выбрана радиокнопка rad2, то значение переменной будет var будет 2. Изначально также требуется установить значение переменной (выражение var.set(1) задает значение переменной var равное 1).

Флажки

Объект `checkbutton` предназначен для выбора не взаимоисключающих пунктов в окне (в группе можно активировать один, два или более флажков или не один). В отличие от радиокнопок, значение каждого флажка привязывается к своей переменной, значение которой определяется опциями `onvalue` (включено) и `offvalue` (выключено) в описании флажка.

```
c1 = IntVar()
c2 = IntVar()
che1 = Checkbutton(root,text="Первый флажок",
           variable=c1,onvalue=1,offvalue=0)
che2 = Checkbutton(root,text="Второй флажок",
           variable=c2,onvalue=2,offvalue=0)
```

Списки

Вызов класса Listbox создает объект, в котором пользователь может выбрать один или несколько пунктов в зависимости от значения опции selectmode. В примере ниже значение SINGLE позволяет выбирать лишь один пункт из списка.

```
r = ['Linux','Python','Tk','Tkinter']
lis = Listbox(root,selectmode=SINGLE,height=4)
for i in r:
     lis.insert(END,i)
```

Изначально список (Listbox) пуст. С помощью цикла for в него добавляются пункты из списка (тип данных) r. Добавление происходит с помощью специального метода класса Listbox — insert. Данный метод принимает два параметра: куда добавить и что добавить.

Большинство методов различных виджет мы рассмотрим по ходу изучения данного курса.

Виджеты (графические объекты) и их свойства

Продолжим рассматривать графические объекты (виджеты), содержащихся в библиотеке Tkinter. Это будут рамка (frame), шкала (scale), полоса прокрутки (scrollbar), окно верхнего уровня (toplevel).

Frame (рамка)

Как выяснится позже, рамки (фреймы) хороший инструмент для организации остальных виджет в группы внутри окна, а также оформления.

```
from tkinter import *

root = Tk()

fra1 = Frame(root,width=500,height=100,bg="darkred")
fra2 =

    Frame(root,width=300,height=200,bg="green",bd=20)
fra3 = Frame(root,width=500,height=150,bg="darkblue")

fra1.pack()
fra2.pack()
fra3.pack()

root.mainloop()
```

Данный скрипт создает три фрейма разного размера. Свойство bd (сокращение от boderwidth) определяет расстояния от края рамки до заключенных в нее виджетов (если они есть).

На фреймах также можно размещать виджеты как на основном окне (root). Здесь текстовое поле находится на рамке fra2.

```
ent1 = Entry(fra2,width=20)
ent1.pack()
```

Scale (шкала)

Назначение шкалы — это предоставление пользователю выбора какого-то значения из определенного диапазона. Внешне шкала представляет собой горизонтальную или вертикальную полосу с разметкой, по которой пользователь может передвигать движок, осуществляя тем самым выбор значения.

```
sca1 = Scale(fra3,orient=HORIZONTAL,length=300,
          from_=0,to=100,tickinterval=10,resolution=5)
sca2 = Scale(root,orient=VERTICAL,length=400,
          from_=1,to=2,tickinterval=0.1,resolution=0.1)
```

Свойства:

- orient определяет направление шкалы;
- length – длина шкалы в пикселях;
- from_ и to – с какого значения шкала начинается и каким заканчивается (т. е. диапазон значений);
- tickinterval – интервал, через который отображаются метки для шкалы;
- resolution - минимальная длина отрезка, на которую пользователь может передвинуть движок.

Scrollbar (полоса прокрутки)

Данный виджет позволяет прокручивать содержимое другого виджета (например, текстового поля или списка). Прокрутка может быть как по горизонтали, так и по вертикали.

```
from tkinter import *

root = Tk()

tx = Text(root,width=40,height=3,font='14')
scr = Scrollbar(root,command=tx.yview)
tx.configure(yscrollcommand=scr.set)

tx.grid(row=0,column=0)
scr.grid(row=0,column=1)
root.mainloop()
```

В примере сначала создается текстовое поле (tx), затем полоса прокрутки (scr), которая привязывается с помощью опции command к полю tx по вертикальной оси (yview). Далее поле tx изменяется

(конфигурируется) с помощью метода configure: устанавливается значение опции `yscrollcommand`.

Здесь используется незнакомый нам пока еще метод `grid`, представляющий собой другой способ расположения виджет на окне.

Toplevel (окно верхнего уровня)

С помощью класс `Toplevel` создаются дочерние окна, на которых также могут располагаться виджеты. Следует отметить, что при закрытии главного окна (или родительского), окно `Toplevel` также закрывается. С другой стороны, закрытие дочернего окна не приводит к закрытию главного.

```
win = Toplevel(root,relief=SUNKEN,bd=10,bg="lightblue")
win.title("Дочернее окно")
win.minsize(width=400,height=200)
```

Метод `title` определяет заголовок окна. Метод `minsize` конфигурирует минимальный размер окна (есть метод `maxsize`, определяющий максимальный размер окна). Если значение аргументов `minsize` будет таким же как у `maxsize`, то пользователь не сможет менять размеры окна.

XXIII. Программирование событий.

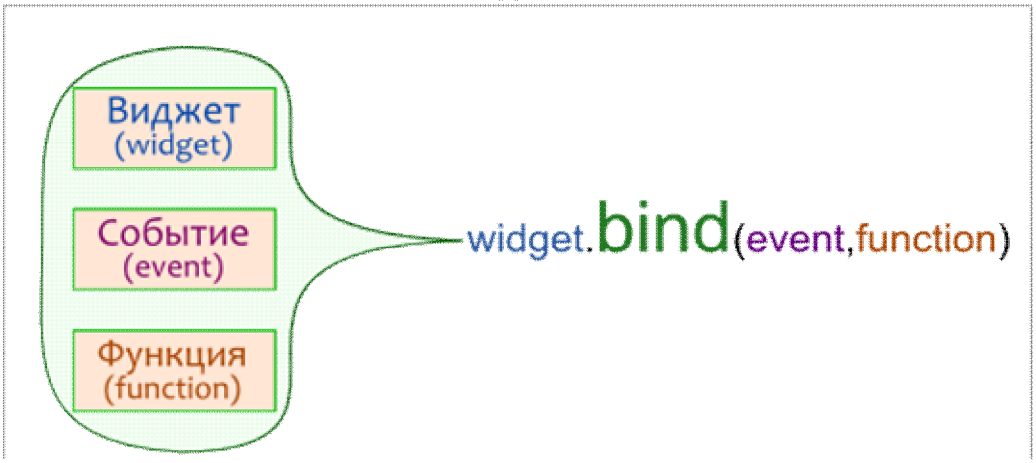
Метод bind модуля Tkinter.

Приложения с графическим интерфейсом пользователя (GUI) должны не просто красиво отображаться на экране, но и выполнять какие-либо действия, реализуя тем самым потребности пользователя. На прошлых уроках было рассказано как создать GUI, на этом уроке рассмотрим как добавить ему функциональность, т.е возможность совершать с его помощью те или иные действия.

В отличие от консольных приложений, которые обычно выполняются при минимальных внешних воздействиях, графическое приложение обычно ждет каких-либо внешних воздействий (щелчков кнопкой мыши, нажатий клавиш на клавиатуре, изменения виджетов) и затем выполняет заложенное программистом действие. Из такого принципа работы можно вывести следующую схему настройки функциональности GUI: на виджет что-то «влияет» из вне ? выполняется какая-то функция (действие). Внешнее воздействие на графический компонент называется событием. Событий достаточно много (основной их перечень мы рассмотрим на следующем занятии). На этом занятии будем использовать лишь два вида событий: щелчок левой кнопкой мыши () и нажатие клавиши `Enter` ().

Одним из способов связывания виджета, события и функции (того, что должно происходить после события) является использование метода `bind`. Синтаксис связывания представлен на рисунке ниже.

Добавление функциональности графическому элементу с помощью метода bind



Рассмотрим различные примеры добавления функциональности GUI.

Пример 1.

```
def output(event):
     s = ent.get()
     if s == "1":
          tex.delete(1.0,END)
          tex.insert(END,"Обслуживание клиентов на

                                          втором этаже")
     elif s == "2":
          tex.delete(1.0,END)
          tex.insert(END,"Пластиковые карты выдают в

                                       соседнем здании")
     else:
          tex.delete(1.0,END)
          tex.insert(END,"Введите 1 или 2 в поле

                                             слева")

from tkinter import *
root = Tk()

ent = Entry(root,width=1)   ## 16
but = Button(root,text="Вывести")   ## 17
tex =Text(root,width=20,height=3,font="12",wrap=WORD)

ent.grid(row=0,column=0,padx=20)     ## 22
but.grid(row=0,column=1)
tex.grid(row=0,column=2,padx=20,pady=10)

but.bind("<Button-1>",output)   ## 24
```

```
root.mainloop()
```

Рассмотрим код, начиная с 16-й строки.

В строках 16-18 создаются три виджета: однострочное текстовое поле, кнопка и многострочное текстовое поле. В первое поле пользователь должен что-то ввести, затем нажать кнопку и получить ответ во втором поле.

В строках 20-22 используется менеджер `grid` для размещения виджетов. Свойства `padx` и `pady` определяют количество пикселей от виджета до края рамки (или ячейки) по осям x и y соответственно.

В стоке 24 как раз и происходит связывание кнопки с событием нажатия левой кнопки мыши и функцией `output`. Все эти три компонента (виджет, событие и функция) связываются с помощью метода `bind`. В данном случае, при нажатии левой кнопкой мыши по кнопке `but` будет вызвана функция `output`.

Итак, если вдруг пользователь щелкнет левой кнопкой мыши по кнопке, то выполнится функция `output` (ни в каком другом случае она выполняться не будет). Данная функция (строки 1-11) выводит информацию во второе текстовое поле. Какую именно информацию, зависит от того, что пользователь ввел в первое текстовое поле. В качестве аргумента функции передается событие (в данном случае).

Внутри веток `if-elif-else` используются методы `delete` и `insert`. Первый из них удаляет символы из текстового поля, второй — вставляет. 1.0 — обозначает первую строку, первый символ (нумерация символов начинается с нуля).

Пример 2.

```
li = ["red","green"]
def color(event):
     fra.configure(bg=li[0])
     li[0],li[1] = li[1],li[0]

def outgo(event):
     root.destroy()

from tkinter import *
root = Tk()
fra = Frame(root,width=100,height=100)    # 12 & 13
but = Button(root,text="Выход")
fra.pack()
but.pack()
root.bind("<Return>",color)              #  18
but.bind("<Button-1>",outgo)              #  19

root.mainloop()
```

Здесь создаются два виджета (строки 12, 13): фрейм и кнопка.

Приложение реагирует на два события: нажатие клавиши `Enter` в пределах главного окна (строка 18) и нажатие левой кнопкой мыши по кнопке `but` (строка 19). В первом случае вызывается функция `color`, во втором — outgo.

Функция color изменяет цвет фона (`bg`) фрейма (`fra`) с помощью метода configure, который предназначен для изменения значения свойств виджетов в процессе выполнения скрипта. В качестве значения опции bg подставляется первый элемент списка. Затем в списке два элемента меняются местами, чтобы при следующем нажатии Enter цвет фрейма снова изменился.

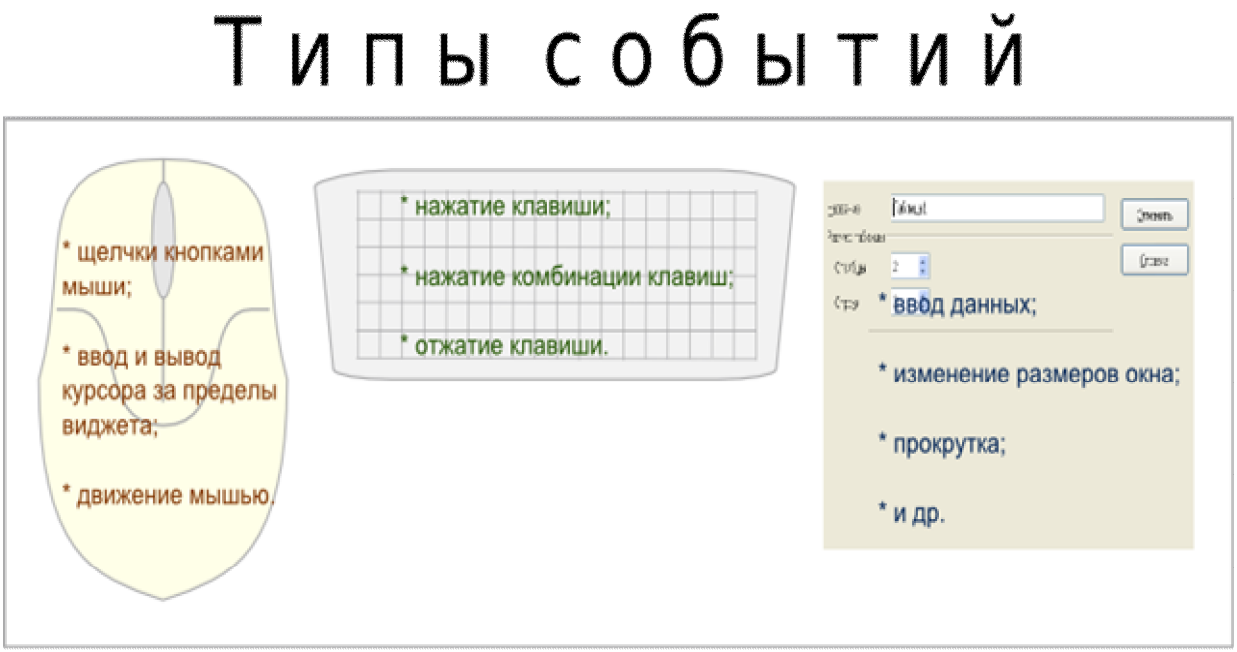
В функции outgo вызывается метод destroy по отношению к главному окну. Данный метод предназначен для «разрушения» виджета (окно закроется).

Программирование событий в Tkinter

Обычно, чтобы графическое приложение что-то сделало, должно случиться какое-нибудь событие, т. е. воздействие на GUI из вне.

Типы событий

Можно выделить три основных типа событий: производимые мышью, нажатиями клавиш на клавиатуре, а также события, возникающие в результате изменения других графических объектов.



Способ записи

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

Название события заключается в кавычки, а также в знаки < и >. Событие описывается с помощью зарезервированных последовательностей ключевых слов.

События, производимые мышью

- **<Button-1>** - щелчок левой кнопкой мыши
- **<Button-2>** - щелчок средней кнопкой мыши
- **<Button-3>** - щелчок правой кнопкой мыши
- **<Double-Button-1>** - двойной клик левой кнопкой мыши
- **<Motion>** - движение мыши

и т. д.

Пример:

```
from tkinter import *
def b1(event):
     root.title("Левая кнопка мыши")
def b3(event):
     root.title("Правая кнопка мыши")
def move(event):
     root.title("Движение мышью")

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

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

root.mainloop()
```

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

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

виджетом является главное окно, а событием – <Motion>, т. е. перемещение мыши.

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

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

```
from tkinter import *


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


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

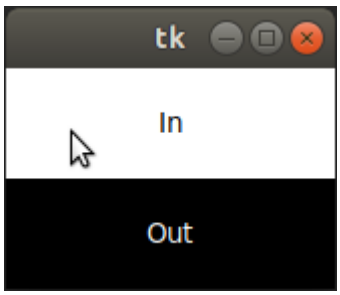
Пример выполнения программы:

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

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

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

Рассмотрим программу:




```
from tkinter import *


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


root = Tk()

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

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

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

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

Свойство `event.widget` содержит ссылку на виджет, сгенерировавший событие.

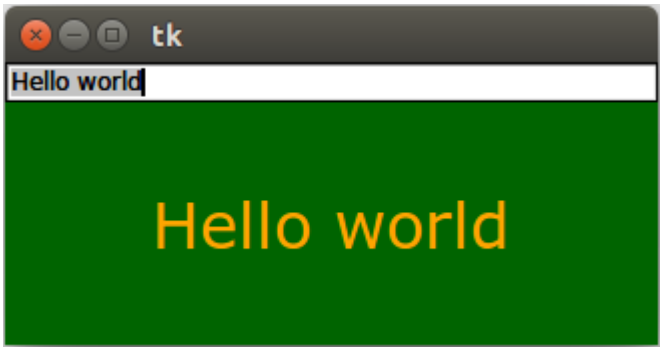
Свойство **`event.type`** описывает, что это было за событие.

У каждого события есть имя и номер. С помощью выражения `print(repr(event.type))` можно посмотреть его полное описание.

При этом на одних платформах `str(event.type)` возвращает имя события (например, `'Enter'`), на других – строковое представление номера события (например, '7').

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

```
from tkinter import *


def exit_win(event):
    root.destroy()


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


def select_all(event):

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

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


root = Tk()

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

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

root.mainloop()
```



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

События, производимые с помощью клавиатуры

- Буквенные клавиши можно записывать без угловых скобок (например, 'L').
- Для неалфавитных клавиш существуют специальные зарезервированные слова
  * <Return> - нажатие клавиши Enter;
  * <space>- пробел;
  * и т. д.
- Сочетания клавиш пишутся через тире. Например:
  * <Control-Shift> - одновременное нажатие клавиш Ctrl и Shift.

Примерчик:

```
from tkinter import *

def exit_(event):
     root.destroy()
def caption(event):
     t = ent.get()
     lbl.configure(text = t)

root = Tk()

ent = Entry(root, width = 40)
lbl = Label(root, width = 80)

ent.pack()
lbl.pack()
```

```
ent.bind('<Return>',caption)
root.bind('<Control-z>',exit_)

root.mainloop()
```

При нажатии клавиши Enter в пределах текстовой строки (ent) вызывается функция caption, которая помещает символы из текстовой строки (ent) в метку (lbl). Нажатие комбинации клавиш Ctrl + z приводит к закрытию главного окна.

XXIV. Переменные Tkinter.

Библиотека Tkinter содержит специальные классы, объекты которых выполняют роль переменных для хранения значений о состоянии различных виджет. Изменение значения такой переменной ведет к изменению и свойства виджета, и наоборот: изменение свойства виджета изменяет значение ассоциированной переменной.

Существует несколько таких классов Tkinter, предназначенных для обработки данных разных типов.

1. **StringVar**() - для строк;
2. **IntVar**() - целых чисел;
3. **DoubleVar**() - дробных чисел;
4. **BooleanVar**() - для обработки булевых значений (true и false).

Пример 1.

Раньше мы уже использовали переменную-объект типа **IntVar**() при создании группы радиокнопок:

```
var=IntVar()
var.set(1)
rad0 =
   Radiobutton(root,text="Первая",variable=var,value=0)
rad1 =
   Radiobutton(root,text="Вторая",variable=var,value=1)
rad2 =
   Radiobutton(root,text="Третья",variable=var,value=2)
```

Здесь создается объект класса `IntVar` и связывается с переменной `var`. С помощью метода `set` устанавливается начальное значение, равное 1. Три радиокнопки относятся к одной группе: об этом свидетельствует одинаковое значение опции (свойства) `variable`. `Variable` предназначена для связывания переменной Tkinter с радиокнопкой. Опция value определяет значение, которое будет передано переменной, если данная кнопка будет в состоянии "включено". Если в процессе выполнения скрипта значение переменной `var` будет изменено, то это отразится на группе кнопок. Например, это делается во второй строчке кода: включена кнопка `rad1`.

Если метод set позволяет устанавливать значения переменных, то метод get, наоборот, позволяет получать (узнавать) значения для последующего их использования.

```
def display(event):
     v = var.get()
     if v == 0:
          print ("Включена первая кнопка")
     elif v == 1:
          print ("Включена вторая кнопка")
     elif v == 2:
          print ("Включена третья кнопка")

but = Button(root,text="Получить значение")
but.bind('<Button-1>',display)
```

При вызове функции display в переменную v “записывается“ значение, связанное в текущий момент с переменной var. Чтобы получить значение переменной var, используется метод get (вторая строчка кода).

Пример 2.

Несколько сложнее обстоит дело с флажками. Поскольку состояния флажков независимы друг друга, то для каждого должна быть введена собственная ассоциированная переменная-объект.

```
from tkinter import *

root = Tk()

var0=StringVar() # значение каждого флажка ...
var1=StringVar() # ... хранится в собственной
переменной
var2=StringVar()
# если флажок установлен, то в ассоциированную
переменную ...
# ...(var0,var1 или var2) заносится значение onvalue,
...
# ...если флажок снят, то - offvalue.
ch0 = Checkbutton(root,text="Окружность",variable=var0,
          onvalue="circle",offvalue="-")
ch1 = Checkbutton(root,text="Квадрат",variable=var1,
          onvalue="square",offvalue="-")
ch2 =
Checkbutton(root,text="Треугольник",variable=var2,
          onvalue="triangle",offvalue="-")

lis = Listbox(root,height=3)
def result(event):
```

```
    v0 = var0.get()
    v1 = var1.get()
    v2 = var2.get()
    l = [v0,v1,v2] # значения переменных заносятся в
                                                список
    lis.delete(0,2) # предыдущее содержимое удаляется
                                              из Listbox
    for v in l: # содержимое списка l последовательно
        lis.insert(END,v) # ...вставляется в Listbox

but = Button(root,text="Получить значения")
but.bind('<Button-1>',result)

ch0.deselect()# "по умолчанию" флажки сняты
ch1.deselect()
ch2.deselect()

ch0.pack()
ch1.pack()
ch2.pack()
but.pack()
lis.pack()

root.mainloop()
```

Пример 3.

Помимо свойства (опции) variable, связывающей виджет с переменной-объектом Tkinter (IntVar, StringVar и др.), у многих виджет существует опция textvariable, которая определяет текст-содержимое или текст-надпись виджета. Несмотря на то, что «текстовое свойство» может быть установлено для виджета и изменено в процессе выполнения кода без использования ассоциированных переменных, иногда такой способ изменения оказывается более удобным.

```
from tkinter import *
root = Tk()
v = StringVar()
ent1 = Entry (root, textvariable =
                              v,bg="black",fg="white")
ent2 = Entry(root, textvariable = v)
ent1.pack()
ent2.pack()
root.mainloop()
```

Здесь содержимое одного текстового поля немедленно, отображается в другом, т.к. оба поля привязаны к одной и той же переменной v.

XXV. Объект Меню в GUI.

Что такое меню

Меню — это объект, который присутствует во многих пользовательских приложениях. Находится оно под строкой заголовка и представляет собой выпадающие списки под словами; каждый такой список может содержать другой вложенный в него список. Каждый пункт списка представляет собой команду, запускающую какое-либо действие или открывающую диалоговое окно.

Создание меню в Tkinter

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

m = Menu(root) #создается объект Меню на главном окне
root.config(menu=m) #окно конфигурируется с указанием
                                        меню для него

fm = Menu(m) #создается пункт меню с размещением на
                                          основном меню (m)
m.add_cascade(label="File",menu=fm) #пункту
                                             располагается
                                      на основном меню (m)
fm.add_command(label="Open...") #формируется список
                                        команд пункта меню
fm.add_command(label="New")
fm.add_command(label="Save...")
fm.add_command(label="Exit")

hm = Menu(m) #второй пункт меню
m.add_cascade(label="Help",menu=hm)
hm.add_command(label="Help")
hm.add_command(label="About")

root.mainloop()
```

Метод `add_cascade` добавляет новый пункт в меню, который указывается как значение опции `menu`.

Метод `add_command` добавляет новую команду в пункт меню. Одна из опций данного метода (в примере выше ее пока нет) — `command` – связывает данную команду с функцией- обработчиком.

Можно создать вложенное меню. Для этого создается еще одно меню и с помощью `add_cascade` привязать к родительскому пункту.

```
nfm = Menu(fm)
```

```
fm.add_cascade(label="Import",menu=nfm)
nfm.add_command(label="Image")
nfm.add_command(label="Text")
```

Привязка функций к меню

Каждая команда меню обычно должна быть связана со своей функцией, выполняющей те или иные действия (выражения). Связь происходит с помощью опции `command` метода `add_command`. Функция обработчик до этого должна быть определена.

Для примера выше далее приводятся исправленные строки добавления команд “`About`”, “`New`” и “`Exit`”, а также функции, вызываемые, когда пользователь щелкает левой кнопкой мыши по соответствующим пунктам подменю.

```
def new_win():
     win = Toplevel(root)

def close_win():
     root.destroy()

def about():
     win = Toplevel(root)
     lab = Label(win,text="Это просто программа-тест \n
                                        меню в Tkinter")
     lab.pack()
….
fm.add_command(label="New",command=new_win)
….
fm.add_command(label="Exit",command=close_win)

….
hm.add_command(label="About",command=about)
```

Упражнение - пример

Напишем приложение с меню, содержащим два пункта: Color и Size. Пункт Color должен содержать три команды (Red, Green и Blue), меняющие цвет рамки на главном окне. Пункт Size должен содержать две команды (500x500 и 700x400), изменяющие размер рамки.

Примерное решение:

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

def colorR():
     fra.config(bg="Red")
def colorG():
     fra.config(bg="Green")
def colorB():
```

```
     fra.config(bg="Blue")

def square():
     fra.config(width=500)
     fra.config(height=500)
def rectangle():
     fra.config(width=700)
     fra.config(height=400)

fra = Frame(root,width=300,height=100,bg="Black")
fra.pack()

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

cm = Menu(m)
m.add_cascade(label="Color",menu=cm)
cm.add_command(label="Red",command=colorR)
cm.add_command(label="Green",command=colorG)
cm.add_command(label="Blue",command=colorB)

sm = Menu(m)
m.add_cascade(label="Size",menu=sm)
sm.add_command(label="500x500",command=square)
sm.add_command(label="700x400",command=rectangle)

root.mainloop()
```

XXVI. Диалоговые окна в Tkinter

Диалоговые окна, как элементы графического интерфейса, предназначены для вывода сообщений пользователю, получения от него какой-либо информации, а также управления.

Диалоговые окна весьма разнообразны. Здесь будут рассмотрены лишь несколько.

Рассмотрим, как запрограммировать с помощью `Tkinter` вызов диалоговых окон открытия и сохранения файлов и работу с ними. При этом требуется дополнительно импортировать "подмодуль" `Tkinter` - `tkinter.filedialog`, в котором описаны классы для окон данного типа.

```
from tkinter import *
from tkinter.filedialog import *

root = Tk()
op = askopenfilename()
```

```
sa = asksaveasfilename()

root.mainloop()
```

Здесь создаются два объекта (op и sa): один вызывает диалоговое окно "Открыть", а другой "Сохранить как...". При выполнении скрипта, они друг за другом выводятся на экран после появления главного окна. Если не создать root, то оно все-равно появится на экране, однако при попытке его закрытия в конце возникнет ошибка.

Давайте теперь разместим многострочное текстовое поле на главном окне и в дальнейшем попробуем туда загружать содержимое небольших текстовых файлов. Поскольку окно сохранения файла нам пока не нужно, то закомментируем эту строчку кода или удалим. В результате должно получиться примерно так:

```
from tkinter import *
from tkinter.filedialog import *

root = Tk()
txt = Text(root,width=40,height=15,font="12")
txt.pack()

op = askopenfilename()

root.mainloop()
```

При запуске скрипта появляется окно с текстовым полем и сразу диалоговое окно "Открыть". Однако, если мы попытаемся открыть какой-нибудь текстовый файл, то в лучшем случае ничего не произойдет. Как же связать содержимое текстового файла с текстовым полем через диалог "Открыть"?

Что если просто вставить содержимое переменной op в текстовое поле:

```
txt.insert(END,op)
```

После запуска скрипта и попытки открытия файла в текстовом поле оказывается адрес файла. Значит содержимое файла надо прочитать каким-то методом (функцией).

Метод `input` модуля `fileinput` может принимать в качестве аргумента адрес файла, читать его содержимое, формируя список строк. Далее с помощью цикла **`for`** можно извлекать строки последовательно и помещать их, например, в текстовое поле.

```
.....
import fileinput
.....
for i in fileinput.input(op):
     txt.insert(END,i)
.....
```

Обратите внимание на то, как происходит обращение к функции `input` модуля `fileinput` и его импорт. Дело в том, что в `Python` уже встроена

своя функция input (ее назначение абсолютно иное) и во избежание "конфликта" требуется четко указать, какую именно функцию мы имеем ввиду. Поэтому вариант импорта '**from** fileinput **import** input' здесь не подходит.

Окно "Открыть" запускается сразу при выполнении скрипта. На самом деле так не должно быть. Необходимо связать запуск окна с каким-нибудь событием. Пусть это будет щелчок на пункте меню.

```
from tkinter import *
from tkinter.filedialog import *
import fileinput

def _open():
     op = askopenfilename()
     for l in fileinput.input(op):
          txt.insert(END,l)

root = Tk()

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

fm = Menu(m)
m.add_cascade(label="File",menu=fm)
fm.add_command(label="Open...",command=_open)

txt = Text(root,width=40,height=15,font="12")
txt.pack()

root.mainloop()
```

Теперь попробуем сохранять текст, набранный в текстовом поле. Добавим в код пункт меню и следующую функцию:

```
def _save():
     sa = asksaveasfilename()
     letter = txt.get(1.0,END)
     f = open(sa,"w")
     f.write(letter)
     f.close()
```

В переменной sa храниться адрес файла, куда будет производиться запись. В переменной letter – текст, "полученный" из текстового поля. Затем файл открывается для записи, в него записывается содержимое переменной letter, и файл закрывается (на всякий случай).

Еще одна группа диалоговых окон описана в модуле tkinter.messagebox. Это достаточно простые диалоговые окна для вывода сообщений, предупреждений, получения от пользователя ответа "да" или "нет" и т. п.

Дополним нашу программу пунктом Exit в подменю File и пунктом About program в подменю Help.

```
from tkinter.messagebox import *
….
def close_win():
     if askyesno("Exit", "Do you want to quit?"):
          root.destroy()

def about():
     showinfo("Editor", "This is text editor.\n

                                          (test version)")
...
fm.add_command(label="Exit",command=close_win)
....
hm = Menu(m)
m.add_cascade(label="Help",menu=hm)
hm.add_command(label="About",command=about)
...
```

В функции about происходит вызов окна showinfo, позволяющее выводить сообщение для пользователя с кнопкой OK. Первый аргумент — это то, что выведется в заголовке окна, а второй — то, что будет содержаться в теле сообщения. В функции close_win вызывается окно askyesno, которое позволяет получить от пользователя два ответа (true и false). В данном случае при положительном ответе сработает ветка if и главное окно будет закрыто. В случае нажатия пользователем кнопки "No" окно просто закроется (хотя можно было запрограммировать в ветке else какое-либо действие).

XXVII. Геометрические примитивы графического элемента Canvas (холст)

Canvas (холст) — это достаточно сложный объект библиотеки tkinter. Он позволяет располагать на самом себе другие объекты. Это могут быть как геометрические фигуры, узоры, вставленные изображения, так и другие виджеты (например, метки, кнопки, текстовые поля). И это еще не все. Отображенные на холсте объекты можно изменять и перемещать (при желании) в процессе выполнения скрипта. Учитывая все это, canvas находит широкое применение при создании GUI-приложений с использованием tkinter (создание рисунков, оформление других виджет, реализация функций графических редакторов, программируемая анимация и др.).

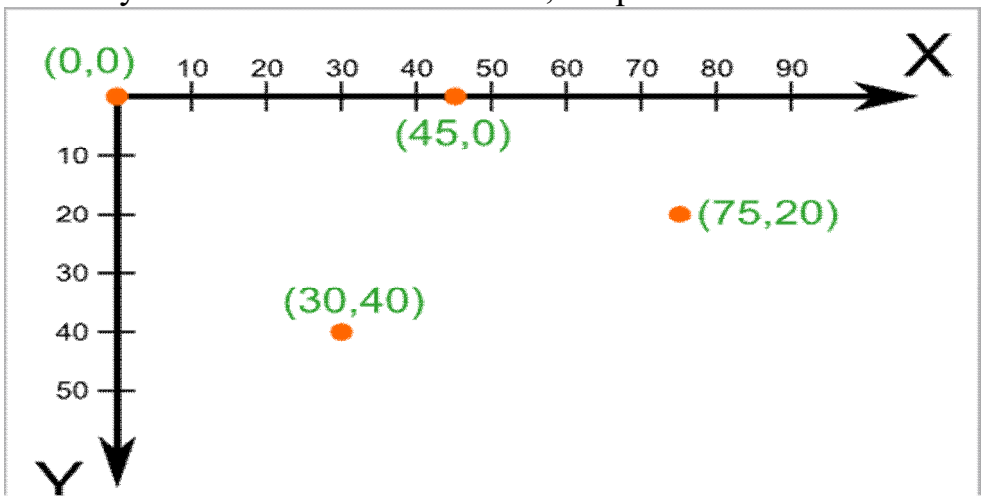
В данном уроке будет рассмотрено создание на холсте графических примитивов (линии, прямоугольника, многоугольника, дуги (сектора), эллипса) и текста.

Для того, чтобы создать объект-холст необходимо вызвать соответствующий класс модуля tkinter и установить некоторые значения свойств (опций). Например:

```
canv = Canvas(root,width=500,height=500,bg="lightblue",
                                              cursor="pencil")
```

Далее с помощью любого менеджера геометрии разместить на главном окне.

Перед тем как создавать геометрические фигуры на холсте следует разобраться с координатами и единицами измерения расстояния. Нулевая точка (0,0) для объекта Canvas располагается в верхнем левом углу. Единицы измерения пиксели (точки экрана). Для «ориентации в пространстве» объекта Canvas рассмотрите рисунок ниже. У любой точки первое число — это расстояние от нулевого значения по оси X, второе — по оси Y.



Чтобы нарисовать линию на холсте следует к объекту (в нашем случае, canv) применить метод `create_line`.

```
canv.create_line(200,50,300,50,width=3,fill="blue")
canv.create_line(0,0,100,100,width=2,arrow=LAST)
```

Четыре числа — это пары координат начала и конца линии, т.е в примере первая линия начинается из точки (200,50), а заканчивается в точке (300,50). Вторая линия начинается в точке (0,0), заканчивается — в (100,100).

Свойство `fill` позволяет задать цвет линии отличный от черного, а `arrow` – установить стрелку (в конце, начале или по обоим концам линии).

Метод `create_rectangle` создает прямоугольник. Аналогично линии в скобках первыми аргументами прописываются четыре числа. Первые две координаты обозначают верхний левый угол прямоугольника, вторые — правый нижний. В примере ниже используется немного иной подход. Он может быть полезен, если начальные координаты объекта могут изменяться, а его размер строго регламентирован.

```
x = 75
y = 110
canv.create_rectangle(x,y,x+80,y+50,fill="white",
                                              outline="blue")
```

Опция `outline` определяет цвет границы прямоугольника.

Чтобы создать произвольный многоугольник, требуется задать пары координат для каждой его точки.

```
canv.create_polygon([250,100],[200,150],[300,150],
                                          fill="yellow")
```

Квадратные скобки при задании координат используются для удобочитаемости (их можно не использовать). Свойство smooth задает сглаживание.

```
canv.create_polygon([250,100],[200,150],[300,150],
                                          fill="yellow")
canv.create_polygon([300,80],[400,80],
                                    [450,75],[450,200],
                                  [300,180],[330,160],
                                outline="white",smooth=1)
```

При создании эллипса задаются координаты гипотетического прямоугольника, описывающего данный эллипс.

```
canv.create_oval([20,200],[150,300],fill="gray50")
```

Более сложные для понимания фигуры получаются при использовании метода `create_arc`. В зависимости от значения опции style можно получить сектор (по умолчанию), сегмент (CHORD) или дугу (ARC). Координаты по-прежнему задают прямоугольник, в который вписана окружность, из которой «вырезают» сектор, сегмент или дугу. От опций `start` и `extent` зависит угол фигуры.

```
canv.create_arc([160,230],[230,330],start=0,extent=140,
                                          fill="lightgreen")
canv.create_arc([250,230],[320,330],start=0,extent=140,
                                 style=CHORD,fill="green")
canv.create_arc([340,230],[410,330],start=0,extent=140,
                    style=ARC,outline="darkgreen",width=2)
```

Последний метод объекта `canvas`, который будет рассмотрен в этом уроке — это метод создающий текстовую надпись.

```
canv.create_text(20,330,
  text="Опыты с графическими примитивами\nна холсте",
         font="Verdana 12",anchor="w",justify=CENTER,
                                              fill="red")
```

Трудность здесь может возникнуть с пониманием опции `anchor` (якорь). По умолчанию в заданной координате располагается центр текстовой надписи. Чтобы изменить это и, например, разместить по указанной координате левую границу текста, используется якорь со значением w (от англ. `west` – запад). Другие значения: `n, ne, e, se, s, sw, w, nw.`

Если букв, задающих сторону привязки две, то вторая определяет вертикальную привязку (вверх или вниз «уйдет» текст от координаты). Свойство `justify` определяет лишь выравнивание текста относительно себя самого.

В конце следует отметить, что часто требуется «нарисовать» на холсте какие-либо повторяющиеся элементы. Для того, чтобы не загружать код, используют циклы. Например, так:

```
x=10
while x < 450:
      canv.create_rectangle(x,400,x+50,450)
      x = x + 60
```

Если вы напишите код приведенный в данном уроке (предварительно совершив импорт модуля Tkinter и создание главного окна, а также не забыв расположить на окне холст, и в конце «сделать» mainloop), то при его выполнении увидите такую картину:

Canvas (холст) - методы, идентификаторы и теги

Ранне были рассмотрены методы объекта canvas, формирующие на нем геометрические примитивы и текст. Однако это лишь часть методов холста. В другую условную группу можно выделить методы, изменяющие свойства уже существующих объектов холста (например, геометрических фигур). И тут возникает вопрос: как обращаться к уже созданным фигурам? Ведь если при создании было прописано что-то вроде canvas.create_oval(30,10,130,80) и таких овалов, квадратов и др. на холсте очень много, то как к ним обращаться?

Для решения этой проблемы в tkinter для объектов холста можно использовать идентификаторы и теги, которые затем передаются другим методам. У любого объекта может быть как идентификатор, так и тег. Использование идентификаторов и тегов немного различается.

Рассмотрим несколько методов изменения уже существующих объектов с использованием при этом **идентификаторов**. Для начала создадим холст и три объекта на нем. При создании объекты "возвращают" свои идентификаторы, которые можно связать с переменными (`oval, rect и`

trial в примере ниже) и потом использовать их для обращения к конкретному объекту.

```
c = Canvas(width=460,height=460,bg='grey80')
c.pack()
oval = c.create_oval(30,10,130,80)
rect = c.create_rectangle(180,10,280,80)
trian = c.create_polygon(330,80,380,10,430,80,
                         fill='grey80', outline="black")
```

Если вы выполните данный скрипт, то увидите на холсте три фигуры: овал, прямоугольник и треугольник.

Далее можно использовать методы-"модификаторы" указывая в качестве первого аргумента идентификатор объекта. Метод move перемещает объект на по оси X и Y на расстояние указанное в качестве второго и третьего аргументов. Следует понимать, что это не координаты, а смещение, т. е. в примере ниже прямоугольник опустится вниз на 150 пикселей. Метод itemconfig изменяет указанные свойства объектов, coords изменяет координаты (им можно менять и размер объекта).

```
c.move(rect,0,150)
c.itemconfig(trian,outline="red",width=3)
c.coords(oval,300,200,450,450)
```

Если запустить скрипт, содержащий две приведенные части кода (друг за другом), то мы сразу увидим уже изменившуюся картину на холсте: прямоугольник опустится, треугольник приобретет красный контур, а эллипс сместится и сильно увеличится в размерах. Обычно в программах изменения должны наступать при каком-нибудь внешнем воздействии. Пусть по щелчку левой кнопкой мыши прямоугольник передвигается на два пикселя вниз (он будет это делать при каждом щелчке мышью):

```
def mooove(event):
     c.move(rect,0,2)
...
c.bind('<Button-1>',mooove)
```

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

```
oval = c.create_oval(30,10,130,80,tag="group1")
c.create_line(10,100,450,100,tag="group1")
…
def color(event):
     c.itemconfig('group1',fill="red",width=3)
…
```

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

Еще один метод, который стоит рассмотреть, это `delete`, который удаляет объект по указанному идентификатору или тегу. В `tkinter` существуют зарезервированные теги: например, `all` обозначает все объекты холста. Так в примере ниже функция clean просто очищает холст.

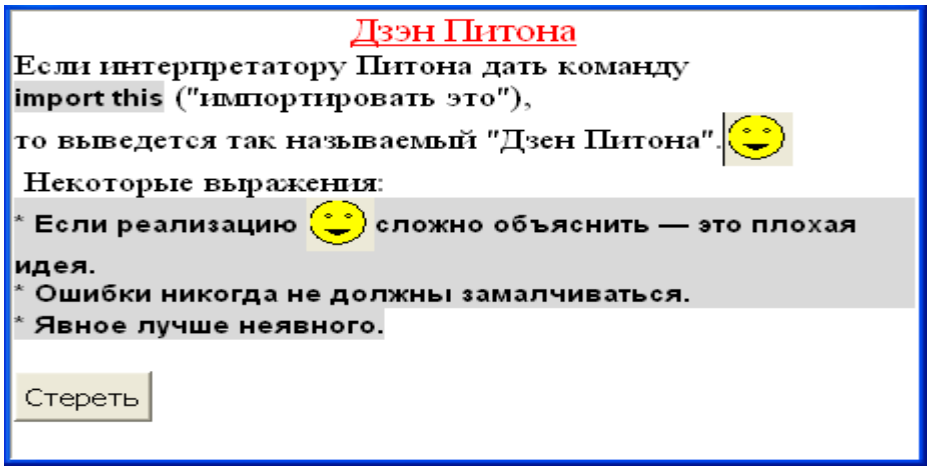
```
def clean(event):
     c.delete('all')
...
c.bind('<Button-2>',clean)
```

Метод `tag_bind` позволяет привязать событие (например, щелчок кнопкой мыши) к определенному объекту. Таким образом, можно реализовать обращение к различным областям холста с помощью одного и того же события. Пример ниже это наглядно иллюстрирует: изменения на холсте зависят от того, где произведен щелчок мышью.

```
from tkinter import *

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

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

def oval_func(event):
     c.delete(oval)
     c.create_text(30,10,
                       text="Здесь был круг",anchor="w")
def rect_func(event):
     c.delete("rect")
     c.create_text(180,10,
           text="Здесь был\nпрямоугольник",anchor="nw")
def triangle(event):
     c.create_polygon(350,70,380,20,410,70,
                         fill='yellow',outline="black")

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

mainloop()
```

Особенности работы с виджетом Text модуля Tkinter.

Графический элемент `Text` предоставляет большие возможности для работы с текстовой информацией. Помимо разнообразных операций с текстом и его форматированием в экземпляр объекта `Text` можно вставлять другие виджеты (следует отметить, что такая же возможность существует и для `Canvas`). В данном уроке рассматриваются лишь некоторые возможности виджета `Text` на примере создания окна с текстовым полем, содержащим форматированный текст, кнопку и возможность добавления экземпляров холста.



1. Для начала создадим текстовое поле, установив при этом некоторые из его свойств:

```
#текстовое поле и его первоначальные настройки
tx =
Text(font=('times',12),width=50,height=15,wrap=WORD)
tx.pack(expand=YES,fill=BOTH)
```

2. Теперь допустим нам нужно добавить какой-нибудь текст. Сделать это можно с помощью метода `insert`, передав ему два обязательных аргумента: место, куда вставить, и объект, который следует вставить. Объектом может быть строка, переменная, ссылающаяся на строку или какой-либо другой объект. Место вставки может указываться несколькими способами. Один из них — это индексы. Они записываются в виде 'x.y', где x – это строка, а y – столбец. При этом нумерация строк начинается с единицы, а столбцов с нуля. Например, первый символ в первой строке имеет индекс '1.0', а десятый символ в пятой строке — '5.9'.

```
tx.insert(1.0,'Дзэн Питона\n\
Если интерпретатору Питона дать команду\n\
import this ("импортировать это"),\n\
       то выведется так называемый "Дзен Питона".\n
                     Некоторые выражения:\n\
       * Если реализацию сложно объяснить
       - — это плохая идея.\n\
         * Ошибки никогда не должны замалчиваться.\n\
      * Явное лучше неявного.\n\n')
```

Комбинация символов '\n' создает новую строку (т.е. при интерпретации последующий текст начнется с новой строки). Одиночный символ '\' никак не влияет на отображение текста при выполнении кода, его следует вставлять при переносе текста при написании скрипта.

Когда содержимого текстового поля нет вообще, то единственный доступный индекс — '1.0'. В заполненном текстовом поле вставлять можно в любое место (где есть содержимое).

Если выполнить скрипт, содержащий только данный код (+ импорт модуля Tkinter, + создание главного окна, + mainloop() в конце), то мы увидим текстовое поле с восемью строчками текста. Текст не оформлен.

3. Теперь отформатируем разные области текста по-разному. Для этого сначала зададим теги для нужных нам областей, а затем для каждого тега установим настройки шрифта и др.

```
#установка тегов для областей текста
tx.tag_add('title','1.0','1.end')
tx.tag_add('special','6.0','8.end')
tx.tag_add('special','3.0','3.11')

#конфигурирование тегов
tx.tag_config('title',foreground='red',
          font=('times',14,'underline'),justify=CENTER)
tx.tag_config('special',background='grey85',
                          font=('Dejavu',10,'bold'))
```

Добавление тега осуществляется с помощью метода tag_add. Первый атрибут — имя тега (произвольное), далее с помощью индексов указывается к какой области текстового поля он прикрепляется (начальный символ и конечный). Вариант записи как '1.end' говорит о том, что нужно взять текст до конца указанной строки. Разные области текста могут быть помечены одинаковым тегом.

Метод `tag_config` применяет те или иные свойства к тегу, указанному в качестве первого аргумента.

4. В многострочное текстовое поле можно добавлять не только текст, но и другие объекты. Например, вставим в поле кнопку (ну и функцию заодно).

```
def erase():
     tx.delete('1.0',END)
...
#добавление кнопки
bt = Button(tx,text='Стереть',command=erase)
tx.window_create(END,window=bt)
```

Кнопка — это виджет. Виджеты добавляются в текстовое поле с помощью метода `window_create`, где в качестве первой опции указывается место добавления, а второй (window) — в качестве значения присваивается переменная, связанная с объектом.

При щелчке ЛКМ (левой кнопкой мыши) по кнопке будет вызываться функция erase, в которой с помощью метода `delete` удаляется все содержимое поля (от '1.0' до END).

5. А вот более интересный пример добавления виджета в поле `Text`:

```
def smiley(event):
      cv = Canvas(height=30,width=30)
      cv.create_oval(1,1,29,29,fill="yellow")
      cv.create_oval(9,10,12,12)
      cv.create_oval(19,10,22,12)
      cv.create_polygon(9,20,15,24,22,20)
      tx.window_create(CURRENT,window=cv)
…
#ЛКМ -> смайлик
tx.bind('<Button-1>',smiley)
```

Здесь при щелчке ЛКМ в любом месте текстового поля будет вызываться функция smiley. В теле данной функции создается объект холста, который в конце с помощью метода window_create добавляется на объект tx. Место вставки указано как CURRENT, т. е. "текущее" - это там, где был произведен щелчок мышью.

XXVIII. Несколько примеров

Игра «жизнь»

(по https://habrahabr.ru/company/mailru/blog/228379/)

```
from tkinter import Tk,
                  Canvas, Button, Frame, BOTH, NORMAL, HIDDEN
# функция получает координаты мыши
# в момент нажатия или передвижения с зажатой кнопкой
def draw_a(e):
     ii = (e.y-3) / cell_size
     jj = (e.x-3) / cell_size
     ii=int(ii);jj=int(jj)
     print("e=",e, ' ii=',ii, '  jj=',jj)
     canvas.itemconfig(cell_matrix[addr(ii, jj)], state=NORMAL,
tags='vis')

# эта функция преобразует двумерную
# координату в простой адрес одномерного массива
def addr(ii,jj):
     if(ii < 0 or jj < 0 or ii >= field_height or jj >=
field_width):
          return int( len(cell_matrix)-1 )
     else:
          return int( ii * (win_width / cell_size) + jj )
```

```
# функция обновления «картинки»
def refresh():
     for i in range(field_height):
          for j in range(field_width):
               k = 0
              # считаем число соседей
               for i_shift in range(-1, 2):
                    for j_shift in range(-1, 2):
                         if (canvas.gettags(
                                cell_matrix[addr(i + i_shift,
                                j + j_shift)])[0] == 'vis' and
                                 (i_shift != 0 or j_shift != 0)):
                             k += 1
               current_tag =
                    canvas.gettags(cell_matrix[addr(i, j)])[0]
               # в зависимости от их числа устанавливаем
               # состояние клетки
               # здесь можно экспериментировать
               # с самими "правилами" игры
               if(k == 3):
                    canvas.itemconfig(cell_matrix[addr(i, j)],
                                      tags=(current_tag, 'to_vis'))
               if(k <= 1 or k >= 4):
                    canvas.itemconfig(cell_matrix[addr(i, j)],
                                       tags=(current_tag, 'to_hid'))
               if(k == 2 and
                           canvas.gettags(cell_matrix[
                                    addr(i, j)])[0] == 'vis'):
                   canvas.itemconfig(cell_matrix[addr(i, j)],
                                       tags=(current_tag, 'to_vis'))

# сам шаг: обновляем состояние и рисуем
def step():
     refresh()
     repaint()

def clear():
     for i in range(field_height):
          for j in range(field_width):
               canvas.itemconfig(cell_matrix[addr(i, j)],
                                 state=HIDDEN, tags=('hid','0'))

# перерисовываем поле по буферу
# согласно второго тега элемента
def repaint():
     for i in range(field_height):
          for j in range(field_width):
               if (canvas.gettags(cell_matrix[addr(i, j)])[1] ==
                                                   'to_hid'):
                    canvas.itemconfig(cell_matrix[addr(i, j)],
                                 state=HIDDEN, tags=('hid','0'))
```

```
                if (canvas.gettags(cell_matrix[addr(i, j)])[1] ==
                                                    'to_vis'):
                    canvas.itemconfig(cell_matrix[addr(i, j)],
                                state=NORMAL, tags=('vis','0'))

# «создание и поддержка выполнения»
#                                   автоматического режима
def loop():
    ##label['text'] = str(n)
    step()
    if (flag_run):
            root.after(500, loop) # call loop() in 0.5 seconds

# «выполнение» автоматического режима
def run():
        global flag_run, btn3
        #print('flag_run=',flag_run)
        if not flag_run:
                btn3.config(text="STOP")
                flag_run=True
                #print('stop')
                loop()
        else:
                btn3.config(text="RUN")
                flag_run=False
                #print('run')

flag_run=False # работает «автоматически» или по шагам

# создаем само окно
root = Tk()
root.title('4 cuba')
# это ширина и высота окна
win_width = 360 #350
win_height = 380 #370
# «строим» конфигурационную строку – размер окна
config_string = "{0}x{1}".format(win_width, win_height + 32)
#  Задаем цвет клетки
fill_color = "green"
# методом geometry() задаем размеры, можно написать и
# просто строчку вида '360x380'
root.geometry(config_string)
# это ширина самой клетки, с учетом «просвета»
cell_size = 20
# создается объект типа Canvas, на котором будет
# происходить непосредственно рисование,
# он делается дочерним по отношению к самому окну root
canvas = Canvas(root, height=win_height)
# пакуем его, аналог show() в других системах
canvas.pack(fill=BOTH)
```

```
# определяем размеры поля в клетках
field_height = int( win_height / cell_size )
field_width = int( win_width / cell_size )
# создаем массив для клеток, он одномерный,
# но используя вспомогательную фунукцию addr -
# будем преобразовавать индексы 2-х мерный к 1-мерному
cell_matrix = []
# здесь в цикле создаем экземпляры клеток
#  и делаем их скрытыми
for i in range(field_height):
     for j in range(field_width):
               # здесь создаем экземпляр клетки
               # и делаем его скрытыми
          square = canvas.create_rectangle(2 + cell_size*j,
                                           2 + cell_size*i,
                                cell_size + cell_size*j - 2,
                                cell_size + cell_size*i - 2,
                                            fill=fill_color)
          canvas.itemconfig(square, state=HIDDEN,
                                           tags=('hid','0'))
           # «добавляем» в массив
          cell_matrix.append(square)
#  создадим  фиктивный элемент,
#  он как бы «повсюду» вне поля
fict_square =
               canvas.create_rectangle(0,0,0,0, state=HIDDEN,
                                           tags=('hid','0'))
cell_matrix.append(fict_square)
# задаем «клетки» которые находятся на поле
                                    #       «по умолчанию»
canvas.itemconfig(cell_matrix[addr(8, 8)], state=NORMAL,
                                                tags='vis')
canvas.itemconfig(cell_matrix[addr(10, 9)], state=NORMAL,
                                                tags='vis')
canvas.itemconfig(cell_matrix[addr(9, 9)], state=NORMAL,
                                                tags='vis')
canvas.itemconfig(cell_matrix[addr(9, 8)], state=NORMAL,
                                                tags='vis')
canvas.itemconfig(cell_matrix[addr(9, 7)], state=NORMAL,
                                                tags='vis')
canvas.itemconfig(cell_matrix[addr(10, 7)], state=NORMAL,
                                                tags='vis')

# создаем фрейм для хранения кнопок и
# аналогичным образом, как с Canvas,
# устанавливаем кнопки дочерные фрейму

frame = Frame(root)
# выполнить «шаг»
btn1 = Button(frame, text='Step', command = step)  # Eval
```

```
# очистить поле
btn2 = Button(frame, text='Clear', command = clear)
# смена режимов «автоматически»  -  «по шагам»
btn3 = Button(frame, text='Run', command = run)
# пакуем кнопки

btn1.pack(side='left')
btn2.pack(side='right')
btn3.pack(side='right')
# пакуем фрейм
frame.pack(side='bottom')

# привязываем события клика и
# движения мыши над canvas к функции draw_a
canvas.bind('<B1-Motion>', draw_a)
canvas.bind('<ButtonPress>', draw_a)
# через 500 мкс передать управление функции loop
root.after(500, loop)
# стандартный цикл, организующий события
# и общую работу оконного приложения
root.mainloop()
```

Разное

```
import sys
from math import *
from tkinter import *
from tkinter.filedialog import *
from tkinter.messagebox import *
#### http://younglinux.info/tkinter/canvas.php

def f1():
    print('f1 - run.........')
def b1(event):
     root.title("Левая кнопка мыши")
def b3(event):
     root.title("Правая кнопка мыши")
def move(event):
     root.title("Движение мышью")
def res(event):
    root.title("изменение размера окна")
    print("изменение размера окна")
    xm, ym = xymax(root.geometry())
    print("--------xm---ym----------> ", xm, "  ", ym)
    print("---------------------> ",canv. .geometry() )
    canv.config(width = xm, height = ym)

def xymax(str):
    """ получим новый размер окна """
    print("===> ", str )
    k1=str.index('x')
```

```
    xmax1=str[:k1]
    k2=str.index('+',k1+1)
    ymax1=str[k1+1:k2]
    xmax=int(xmax1)
    ymax=int(ymax1)
    print(xmax,'  ',ymax)
    return  xmax,ymax

def button_clicked():
    sss=root.geometry()
    print ("Клик!")
    print('root.maxsize()=<', root.maxsize(),'>'  )
    print('canv.height()=<', canv.height(),'>'  )
    print('canv.size()=<', canv.size(),'>'  )
    print(root.geometry() )
    print('---canv.winfo_height()=',canv.winfo_height() )
     print('---canv.winfo_width() =',canv.winfo_width()  )
     return
def window_deleted():
    print ('Окно закрыто')
    #canv.delete('all')
    if askyesno("Завершение программы",
            "Действительно Вы хотите завершить программу?"):
            #self.save_file()
            root.destroy()
    else:
            pass
   #root.destroy()
   # root.quit() # явное указание на выход из программы
   # exit(0)
f = input('f(x):')

root = Tk()

f_1=Frame(root,bg="Yellow",)
f_2=Frame(root,bg="Blue")
f_1.pack(side=BOTTOM)
f_2.pack(side=BOTTOM)
button3 = Button(f_1, text=u"__В3 привет", command=getF)
button3.pack(side=LEFT)

button4 = Button(f_2, text="_Б4 привет", command=getF)
button4.pack(side=RIGHT) #LEFT)

canv = Canvas(root, width = 500, height = 500, bg = "white")

canv.create_line(500,1000,500,0,width=2,arrow=LAST)
canv.create_line(0,500,1000,500,width=2,arrow=LAST)

canv.create_line(200,50,300,50,width=3,fill="blue")
canv.create_line(0,0,100,100,width=2,arrow=LAST)

print('root.resizable()=<',root.resizable(),'>'  )
```

```
print('root.root.maxsize()=<', root.maxsize(),'>'  )

# кнопка по умолчанию
button1 = Button(root, text=u"file read!", command=getF)
button1.pack(side=TOP) #LEFT) #'top') #place

# кнопка с указанием родительского виджета и несколькими
                                                    аргументами
button2 = Button(root, bg="red", text=u"Кликни меня!",
command=button_clicked)
button2.pack(side=TOP) #'top') #top')


root.protocol('WM_DELETE_WINDOW', window_deleted) # обработчик
                                                  закрытия окна
#sys.exit(0)

First_x = -500;

for i in range(16000):
     if (i % 800 == 0):
          k = First_x + (1 / 16) * i
          canv.create_line(k + 500, -3 + 500, k + 500, 3 + 500,
width = 0.5, fill = 'black')
          canv.create_text(k + 515, -10 + 500, text = str(k),
                           fill="purple", font=("Helvectica", "10"))
          if (k != 0):
               canv.create_line(-3 + 500, k + 500, 3 + 500,
                                k + 500, width = 0.5, fill = 'black')
               canv.create_text(20 + 500, k + 500, text = str(k),
                           fill="purple", font=("Helvectica", "10"))
     try:
          x = First_x + (1 / 16) * i
          new_f = f.replace('x', str(x))
          y = -eval(new_f) + 500
          x += 500
          # canv.create_oval(x, y, x + 1, y + 1, fill = 'black')
          canv.create_line(x, y, x + 1, y + 1, fill = 'black')
     except:
          pass
canv.pack()
################
root.bind('<Button-1>',b1)
root.bind('<Button-3>',b3)
root.bind('<Motion>',move)
root.bind('<Configure>',res)
#############
canv.focus_set()
root.mainloop()
print('@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@')
```

XXIX. Символьные вычисления на языке Python.

SymPy представляет собой открытую библиотеку символьных вычислений на языке Python. SymPy полностью написан на языке Python и не требует сторонних библиотек.

Ниже, в познавательных целях (по материалам сайта http://www.asmeurer.com/sympy_doc/dev-py3k/tutorial/tutorial.ru.html) , описаны *самые основные* возможностях пакета SymPy.

Более подробно смотрите http://www.sympy.org/en/index.html и https://www.sympy.org/ru/index.html

Математическая библиотека Python SymPy

SymPy — это библиотека Python для выполнения символьных вычислений. Это система компьютерной алгебры, которая может выступать как отдельное приложение, так и в качестве библиотеки для других приложений. Поработать с ней онлайн можно на https://live.sympy.org/. Поскольку это чистая библиотека Python, ее можно использовать даже в интерактивном режиме.

В SymPy есть разные функции, которые применяются в сфере символьных вычислений, математического анализа, алгебры, дискретной математики, квантовой физики и так далее. SymPy может представлять результат в разных форматах: LaTeX, MathML и так далее. Распространяется библиотека по лицензии New BSD. Первыми эту библиотеку выпустили разработчики Ondřej Čertík и Aaron Meurer в 2007 году.

Вот где применяется SymPy:

- Многочлены
- Математический анализ
- Дискретная математика
- Матрицы
- Геометрия
- Построение графиков
- Физика
- Статистика
- Комбинаторика

Установка SymPy

Для работы SymPy требуется одна важная библиотека под названием `mpmath`. Она используется для вещественной и комплексной арифметики с числами с плавающей точкой произвольной точности. Однако pip установит ее автоматически при загрузке самой SymPy:

```
python -m pip install sympy
```

Использование SymPy в качестве калькулятора

SymPy поддерживает три типа численных данных: **Float**, **Rational** и **Integer**.

Rational представляет собой обыкновенную дробь, которая задается с помощью двух целых чисел: числителя и знаменателя. Например, Rational(1, 2) представляет дробь 1/2, Rational(5, 2) представляет дробь 5/2, и так далее.

```
>>> from sympy import Rational
>>> a = Rational(1, 2)

>>> a
1/2

>>> a*2
1

>>> Rational(2)**50/Rational(10)**50
1/888178419700125232338905334472656 25
```

Важная особенность Python-интерпретатора - при делении двух «питоновских» чисел типа int с помощью оператора “/” получается «питоновский» тип float. Этот же стандарт “true division” по умолчанию включен и в sympy:

```
>>> 1/2
0.5
```

Обратите внимание, что и в том и в другом случае вы имеете дело не с объектом Number из библиотеки SymPy, который представляет число в SymPy, а с питоновскими числами, которые создаются самим интерпретатором Python. Скорее всего, вам нужно будете работать с дробными числами из библиотеки SymPy, поэтому для того чтобы получать результат в виде объектов SymPy убедитесь, что вы используете класс Rational. Кому-то может показаться удобным обозначать Rational как **R**:

```
import sympy
R = sympy.Rational
R(1, 2)
R(1)/2
```

В модуле Sympy имеются особые константы, такие как **e** и **pi**, которые ведут себя как переменные (то есть выражение `1 + pi` не преобразуется сразу в число, а так и останется `1 + pi`):

```
>>> from sympy import pi, E
>>> pi**2
pi**2

>>> pi.evalf()
```

```
3.14159265358979

>>> (pi + E).evalf()
5.85987448204884
```

как видно, функция **evalf** переводит исходное выражение в число с плавающей точкой. Вычисления можно проводить с большей точностью. Для этого нужно передать в качестве аргумента этой функции требуемое число десятичных знаков:

```
>>> pi.evalf(100)
3.14159265358979323846264338327950288419716939937510582
09749445923078164062862089986280348253421170 68
>>>
```

Для работы с математической бесконечностью используется символ **oo**:

```
>>> from sympy import oo
>>> oo > 99999
True
>>> oo + 1
oo
```

Переменные

В отличие от многих других систем компьютерной алгебры, нужно явно декларировать символьные переменные:

```
>>> from sympy import symbols
>>> x = symbols('x')
>>> y = symbols('y')
```

В левой части этого выражения находится переменная Python, которая питоновским присваиванием соотносится с объектом класса Symbol из SymPy.

```
>>> from sympy.abc import x, theta
```

Символьные переменные могут также задаваться и с помощью функций `symbols` или `var`. Они допускают указание диапазона. Их отличие состоит в том, что `var` добавляет созданные переменные в текущее пространство имен:

```
>>> from sympy import symbols, var
>>> a, b, c = symbols('a,b,c')
>>> d, e, f = symbols('d:f')
>>> var('g:h')
(g, h)
>>> var('g:2')
(g0, g1)
```

Экземпляры класса `Symbol` взаимодействуют друг с другом. Таким образом, с помощью них конструируются алгебраические выражения:

```
>>> x + y + x - y
2*x
```

```
>>> (x + y)**2
(x + y)**2
#раскрыть скобки
>>> ((x + y)**2).expand()
x**2 + 2*x*y + y**2
```

Переменные могут быть заменены на другие переменные, числа или выражения с помощью **функции подстановки `subs(old, new)`:**

```
>>> ((x + y)**2).subs(x, 1)
(y + 1)**2

>>> ((x + y)**2).subs(x, y)
4*y**2

>>> ((x + y)**2).subs(x, 1 - y)
1
```

Теперь, с этого момента, для всех написанных ниже примеров мы будем предполагать, что запустили следующую команду по настройке системы отображения результатов (и используем процедуру pprint):

```
>>> from sympy import init_printing
>>> init_printing(use_unicode=False, wrap_line=False,
no_global=True)
```

Она придаст более качественное отображение выражений. Подробнее по системе отображения и печати написано ниже в разделе Печать. Если же установлен шрифт с юникодом, то можно использовать опцию `use_unicode=True` для еще более красивого вывода.

Алгебра

Чтобы **разложить выражение на простейшие дроби** используется функция **`apart(expr, x)`:**

```
#apart(expr, x):
from sympy import apart
from sympy.abc import x, y, z
from sympy import Integral, pprint
z= 1/( (x + 2)*(x + 1) )
pprint(z)
z=apart(1/( (x + 2)*(x + 1) ), x)
pprint(z)
z=apart((x + 1)/(x - 1), x)
pprint(z)
```

Вывод скрипта:

```
      1
```

```
---------------
(x + 1)*(x + 2)

    1       1
- ----- + -----
  x + 2   x + 1

      2
1 + -----
    x - 1
```

http://www.asmeurer.com/sympy_doc/dev-py3k/tutorial/tutorial.ru.html

Чтобы **привести дробь к общему знаменателю** используется функция **together(expr, x):**

```
>>> from sympy import together
>>> together(1/x + 1/y + 1/z)
x*y + x*z + y*z
---------------
     x*y*z

>>> together(apart((x + 1)/(x - 1), x), x)
x + 1
-----
x - 1

>>> together(apart(1/( (x + 2)*(x + 1) ), x), x)
       1
---------------
(x + 1)*(x + 2)
```

Вычисления

Пределы

Для вычисления предела функции, используйте функцию **limit(function, variable, point).**

Например, чтобы вычислить предел f(x) при x -> 0, нужно ввести limit(f, x, 0):

```
>>> from sympy import limit, Symbol, sin, oo
>>> x = Symbol("x")
>>> limit(sin(x)/x, x, 0)
1
```

также можно вычислять пределы при x, стремящемся к бесконечности:

```
>>> limit(x, x, oo)
```

```
oo

>>> limit(1/x, x, oo)
0

>>> limit(x**x, x, 0)
1
```

Более сложные примеры вычисления пределов см. например - test_demidovich.py

Дифференцирование

Продифференцировать любое выражение SymPy, можно используя **diff(func, var).** Примеры:

```
>>> from sympy import diff, Symbol, sin, tan
>>> x = Symbol('x')
>>> diff(sin(x), x)
cos(x)
>>> diff(sin(2*x), x)
2*cos(2*x)

>>> diff(tan(x), x)
   2
tan (x) + 1
```

Можно, через пределы проверить правильность вычислений производной:

```
>>> from sympy import limit
>>> from sympy.abc import delta
>>> limit((tan(x + delta) - tan(x))/delta, delta, 0)
   2
tan (x) + 1
```

Производные более высших порядков можно вычислить, используя дополнительный параметр этой же функции **diff(func, var, n):**

```
>>> diff(sin(2*x), x, 1)
2*cos(2*x)

>>> diff(sin(2*x), x, 2)
-4*sin(2*x)

>>> diff(sin(2*x), x, 3)
-8*cos(2*x)
```

Разложение в ряд

Для разложения в ряд используйте метод **series(var, point, order):**

```
>>> from sympy import Symbol, cos
>>> x = Symbol('x')
```

```
>>> cos(x).series(x, 0, 10)

     2    4     6      8
    x    x     x      x       / 10\
1 - -- + -- - --- + ----- + O\x  /
    2    24   720   40320

>>> (1/cos(x)).series(x, 0, 10)

     2      4       6        8
    x    5*x    61*x    277*x     / 10\
1 + -- + ---- + ----- + ------ + O\x  /
    2     24     720     8064
```

Еще один простой пример:

```
>>> from sympy import Integral, pprint

>>> y = Symbol("y")
>>> e = 1/(x + y)
>>> s = e.series(x, 0, 5)

>>> print(s)
1/y - x/y**2 + x**2/y**3 - x**3/y**4 + x**4/y**5 +
O(x**5)
>>> pprint(s)
          2    3    4
1   x    x    x    x     / 5\
- - -- + -- - -- + -- + O\x /
y    2    3    4    5
    y    y    y    y
```

Суммы

Вычисляет значение суммы f(от заданной переменной) на заданных пределах

summation(f, (i, a, b)) – нахождение суммы слагаемых f, i изменяется от a до b:

$$\mathbf{summation(f, (i, a, b)) = \sum_{i=a}^{b} f}$$

If it cannot compute the sum, it prints the corresponding summation formula. Repeated sums can be computed by introducing additional limits:

Если сумма не «рассчитывается», то печатается соответствующая формула:

```
>>> from sympy import summation, oo, symbols, log
>>> i, n, m = symbols('i n m', integer=True)

>>> summation(2*i - 1, (i, 1, n))
 2
n


>>> summation(1/2**i, (i, 0, oo))
2


>>> summation(1/log(n)**n, (n, 2, oo))
  oo
 ____
 \   `
  \      -n
  /   log (n)
 /___,
n = 2


>>> summation(i, (i, 0, n), (n, 0, m))
      3    2
m    m    m
-- + -- + -
6    2    3

>>> from sympy.abc import x
>>> from sympy import factorial
>>> summation(x**n/factorial(n), (n, 0, oo))
 x
e
```

Интегрирование

SymPy поддерживает вычисление определенных и неопределенных интегралов с помощью функции `integrate()`. Она использует расширенный алгоритм Риша-Нормана и некоторые шаблоны и эвристики. Можно вычислять интегралы трансцендентных, простых и специальных функций:

```
>>> from sympy import integrate, erf, exp, sin, log,
                                   oo, pi, sinh, symbols
```

```
>>> x, y = symbols('x,y')
```

Вы можете интегрировать простейшие функции:

```
>>> integrate(6*x**5, x)
 6
x
>>> integrate(sin(x), x)
-cos(x)
>>> integrate(log(x), x)
x*log(x) - x
>>> integrate(2*x + sinh(x), x)
 2
x  + cosh(x)
```

Примеры интегрирования некоторых специальных функций:

```
>>> integrate(exp(-x**2)*erf(x), x)

  _____      2
\/ pi *erf (x)
--------------
      4
```

Возможно также вычислить определенный интеграл:

```
>>> integrate(x**3, (x, -1, 1))
0
>>> integrate(sin(x), (x, 0, pi/2))
1
>>> integrate(cos(x), (x, -pi/2, pi/2))
2
```

Поддерживаются и несобственные интегралы:

```
>>> integrate(exp(-x), (x, 0, oo))
1
>>> integrate(log(x), (x, 0, 1))
-1
```

Комплексные числа

Помимо мнимой единицы I, которое является мнимым числом, символы тоже могут иметь специальные атрибуты (real, positive, complex и т.д), которые определяют поведение этих символов при вычислении символьных выражений:

```
>>> from sympy import Symbol, exp, I
>>> x = Symbol("x") # a plain x with no attributes
>>> exp(I*x).expand()
 I*x
e
>>> exp(I*x).expand(complex=True)

   -im(x)                -im(x)
I*e      *sin(re(x)) + e      *cos(re(x))
```

```
>>> x = Symbol("x", real=True)
>>> exp(I*x).expand(complex=True)

I*sin(x) + cos(x)
```

Функции

тригонометрические

```
>>> from sympy import asin, asinh, cos, sin, sinh,
symbols, I
>>> x, y = symbols('x,y')

>>> sin(x + y).expand(trig=True)
sin(x)*cos(y) + sin(y)*cos(x)

>>> cos(x + y).expand(trig=True)
-sin(x)*sin(y) + cos(x)*cos(y)

>>> sin(I*x)
I*sinh(x)

>>> sinh(I*x)
I*sin(x)

>>> asinh(I)
I*pi
----
 2

>>> asinh(I*x)
I*asin(x)

>>> sin(x).series(x, 0, 10)

     3     5     7        9
    x     x     x        x        / 10\
x - -- + --- - ---- + ------ + O\x  /
    6    120   5040   362880

>>> sinh(x).series(x, 0, 10)

     3     5     7        9
    x     x     x        x        / 10\
x + -- + --- + ---- + ------ + O\x  /
```

```
    6    120   5040   362880

>>> asin(x).series(x, 0, 10)

     3      5      7       9
    x    3*x    5*x    35*x      / 10\
x + -- + ---- + ---- + ----- + O\x  /
    6     40    112     1152

>>> asinh(x).series(x, 0, 10)

     3      5      7       9
    x    3*x    5*x    35*x      / 10\
x - -- + ---- - ---- + ----- + O\x  /
    6     40    112     1152
```

сферические

```
>>> from sympy import Ylm
>>> from sympy.abc import theta, phi


>>> Ylm(1, 0, theta, phi)


  ___
\/ 3 *cos(theta)
----------------
     _____
    2*\/ pi

>>> Ylm(1, 1, theta, phi)

      ___  I*phi
-\/ 6 *e     *sin(theta)
------------------------
          _____
        4*\/ pi

>>> Ylm(2, 1, theta, phi)

      ____  I*phi
-\/ 30 *e     *sin(theta)*cos(theta)
-----------------------------------
               _____
             4*\/ pi
```

факториалы и гамма-функции

```
>>> from sympy import factorial, gamma, Symbol
>>> x = Symbol("x")
>>> n = Symbol("n", integer=True)

>>> factorial(x)
x!

>>> factorial(n)
n!

>>> gamma(x + 1).series(x, 0, 3) # i.e. factorial(x)

                      /           2     2\
                    2 |EulerGamma    pi |    / 3\
1 - EulerGamma*x + x *|----------- + ---| + O\x /
                      \     2          12/
```

дзета-функции

```
>>> from sympy import zeta
>>> zeta(4, x)
zeta(4, x)

>>> zeta(4, 1)

  4
pi
---
 90

>>> zeta(4, 2)

       4
     pi
-1 + ---
      90

>>> zeta(4, 3)

         4
  17   pi
- -- + ---
  16    90
```

```
>>> from sympy import assoc_legendre, chebyshevt,
legendre, hermite
>>> chebyshevt(2, x)

   2
2*x  - 1

>>> chebyshevt(4, x)

   4      2
8*x  - 8*x  + 1

>>> legendre(2, x)

   2
3*x    1
---- - -
 2     2

>>> legendre(8, x)

      8          6          4         2
6435*x    3003*x    3465*x    315*x     35
------- - ------- + ------- - ------ + ---
  128         32         64        32    128

>>> assoc_legendre(2, 1, x)

        __________
       /     2
-3*x*\/  - x  + 1


>>> assoc_legendre(2, 2, x)

     2
- 3*x  + 3

>>> hermite(3, x)

   3
8*x  - 12*x
```

Дифференциальные уравнения

```
В isympy:
>>> from sympy import Function, Symbol, dsolve
>>> f = Function('f')
>>> x = Symbol('x')
>>> f(x).diff(x, x) + f(x)
f(x) + Derivative(f(x), x, x)
>>> from sympy import Integral, pprint
>>> uuu=f(x).diff(x, x) + f(x)
>>> pprint(uuu)

         2
        d
f(x) + ---(f(x))
         2
       dx

>>> dsolve(f(x).diff(x, x) + f(x), f(x))
Eq(f(x), C1*sin(x) + C2*cos(x))
>>> ud=dsolve(f(x).diff(x, x) + f(x), f(x))
>>> ud
Eq(f(x), C1*sin(x) + C2*cos(x))
>>> pprint(ud)
f(x) = C1*sin(x) + C2*cos(x)
```

Алгебраические уравнения

```
>>> from sympy import solve, symbols
>>> x, y = symbols('x,y')
>>> solve(x**4 - 1, x)
[-1, 1, -I, I]

>>> solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
{x: -3, y: 1}
```

Линейная алгебра

Матрицы

Матрицы задаются с помощью конструктора **Matrix**:

```
>>> from sympy import Matrix, Symbol
>>> Matrix([[1, 0], [0, 1]])
[1  0]
[    ]
[0  1]
```

В матрицах вы также можете использовать символьные переменные:

```
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> A = Matrix([[1, x], [y, 1]])
>>> A
[1  x]
[    ]
[y  1]

>>> A**2
[x*y + 1    2*x  ]
[                ]
[  2*y    x*y + 1]
```

Для того, чтобы узнать о матрицах подробнее, прочитайте, пожалуйста, Руководство по Линейной Алгебре.

Сопоставление с образцом

Чтобы сопоставить выражения с образцами, используйте функцию **.match()** вместе со вспомогательным классом **Wild**. Эта функция вернет словарь с необходимыми заменами, например:

```
>>> from sympy import Symbol, Wild
>>> x = Symbol('x')
>>> p = Wild('p')
>>> (5*x**2).match(p*x**2)
{p: 5}

>>> q = Wild('q')
>>> (x**2).match(p*x**q)
{p: 1, q: 2}
```

Если же сопоставление не удалось, функция вернет``**None**``:

```
>>> print((x + 1).match(p**x))
None
```

Также можно использовать параметр **exclude** для исключения некоторых значений из результата:

```
>>> p = Wild('p', exclude=[1, x])
>>> print((x + 1).match(x + p)) # 1 is excluded
None
>>> print((x + 1).match(p + 1)) # x is excluded
None
>>> print((x + 1).match(x + 2 + p)) # -1 is not
excluded
{p_: -1}
```

Печать

Реализовано несколько способов вывода выражений на экран.

Стандартный

Стандартный способ представлен функцией `str(expression)`, которая работает следующим образом:

```
>>> from sympy import Integral
>>> from sympy.abc import x
>>> print(x**2)
x**2
>>> print(1/x)
1/x
>>> print(Integral(x**2, x))
Integral(x**2, x)
```

«Красивая печать»

Этот способ печати выражений основан на ascii-графике и реализован через функцию **pprint**:

```
>>> from sympy import Integral, pprint
>>> from sympy.abc import x
>>> pprint(x**2)

 2
X

>>> pprint(1/x)

1
-
X

>>> pprint(Integral(x**2, x))
  /
 |
 |  2
 | x  dx
 |
/
```

Если у вас установлен шрифт с юникодом, он будет использовать `Pretty-print` с юникодом по умолчанию. Эту настройку можно отключить, используя `use_unicode`:

```
>>> pprint(Integral(x**2, x), use_unicode=True)
⌠
⎮  2
⎮ x  dx
⌡
```

Для изучения подробных примеров работы Pretty-print с юникодом вы можете обратится к статье Pretty Printing (https://github.com/sympy/sympy/wiki/Pretty-Printing) на Вики.

Для того, чтобы сделать **pprint** по умолчанию в стандартном интерпретаторе, производим следующую процедуру:

```
>>> from sympy import *
>>> import sys
>>> sys.displayhook = pprint
```

Примеры:

```
>>> from sympy import init_printing, var, Integral
>>> init_printing(use_unicode=False,
                      wrap_line=False, no_global=True)
>>> var("x")
X

>>> x**3/3

 3
x
--
3

>>> Integral(x**2, x) #doctest: +NORMALIZE_WHITESPACE
  /
 |
 |  2
 | x  dx
 |
/
```

Печать объектов Python

```
>>> from sympy.printing.python import python
>>> from sympy import Integral
>>> from sympy.abc import x
>>> print(python(x**2))
x = Symbol('x')
e = x**2
>>> print(python(1/x))
x = Symbol('x')
e = 1/x
>>> print(python(Integral(x**2, x)))
x = Symbol('x')
e = Integral(x**2, x)
```

Печать в формате LaTeX

```
>>> from sympy import Integral, latex
>>> from sympy.abc import x
>>> latex(x**2)
x^{2}
>>> latex(x**2, mode='inline')
$x^{2}$
>>> latex(x**2, mode='equation')
\begin{equation}x^{2}\end{equation}
>>> latex(x**2, mode='equation*')
\begin{equation*}x^{2}\end{equation*}
>>> latex(1/x)
\frac{1}{x}
>>> latex(Integral(x**2, x))
\int x^{2}\, dx
```

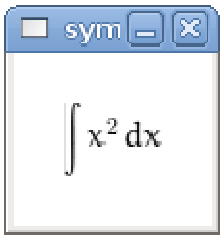
MathML

```
>>> from sympy.printing.mathml import mathml
>>> from sympy import Integral, latex
>>> from sympy.abc import x
>>> print(mathml(x**2))
<apply><power/><ci>x</ci><cn>2</cn></apply>
>>> print(mathml(1/x))
<apply><power/><ci>x</ci><cn>-1</cn></apply>
```

Pyglet

```
>>> from sympy import Integral, preview
>>> from sympy.abc import x
>>> preview(Integral(x**2, x))
```

Появится окно pyglet с отрисованным выражением LaTeX:



Примечания

Если установлены дополнительные пакеты.

isympy вызывает **pprint** автоматически, по этой причине `Pretty-print` будет включен в **isympy** по умолчанию.

Также доступен модуль печати - **sympy.printing**. Через этот модуль доступны следующий функции печати:

`pretty(expr), pretty_print(expr), pprint(expr) –`

Возвращает или выводит на экран, соответственно, “Красивое” представление выражения expr.

`latex(expr), print_latex(expr) –`

Возвращает или выводит на экран, соответственно, LaTeX - представление `expr`

`mathml(expr), print_mathml(expr) -`

Возвращает или выводит на экран, соответственно, MathML (http://www.w3.org/Math/) -представление `expr`.

Чтобы узнать о SymPy подробнее, обратитесь:

`Руководство пользователя SymPy`
`http://www.asmeurer.com/sympy_doc/dev-py3k/guide.html`
и `Описание модулей SymPy`.
`http://www.asmeurer.com/sympy_doc/dev-py3k/modules/index.html`

Также можно обратится на wiki.sympy.org - сайт, который содержит множество полезных примеров, руководств и советов.

Примеры применения пакета SymPy

SymPy - это пакет для символьных вычислений на питоне, подобный системе `Mathematica`. Он работает с выражениями, содержащими символы.

```
from sympy import *
init_printing()
```

Основными кирпичиками, из которых строятся выражения, являются символы. Символ имеет имя, которое используется при печати выражений. Объекты класса Symbol нужно создавать и присваивать переменным питона, чтобы их можно было использовать.

В принципе, имя символа и имя переменной, которой мы присваиваем этот символ - две независимые вещи, и можно написать abc=Symbol('xyz').

Но тогда при вводе программы Вы будете использовать abc, а при печати результатов SymPy будет использовать xyz, что приведёт к ненужной путанице. Поэтому лучше, чтобы имя символа совпадало с именем переменной питона, которой он присваивается.

В языках, специально предназначенных для символьных вычислений, таких, как `Mathematica`, если Вы используете переменную, которой ничего не было присвоено, то она автоматически воспринимается как символ с тем же именем. Питон не был изначально предназначен для символьных вычислений. Если Вы используете переменную, которой ничего не было присвоено, Вы получите сообщение об ошибке. Объекты типа Symbol нужно создавать явно.

```
x=Symbol('x')
```

```
a=x**2-1
a
```

$$x^2 - 1$$

```
type(a)
```

```
sympy.core.add.Add
```

Можно определить несколько символов одновременно. Строка разбивается на имена по пробелам.

```
y,z=symbols('y z')
```

Подставим вместо x выражение y+1

```
a.subs(x,y+1)
```

$$(y+1)^2 - 1$$

Многочлены и рациональные функции

SymPy не раскрывает скобки автоматически. Для этого используется функция **expand**.

```
a=(x+y-z)**6
a
```

$$(x+y-z)^6$$

```
a=expand(a)
a
```

$$x^6 + 6x^5y - 6x^5z + 15x^4y^2 - 30x^4yz + 15x^4z^2 + 20x^3y^3 - 60x^3y^2z + 60x^3yz^2 - 20x^3z^3 + 15x^2y^4 - 60x^2y^3z + 90x^2y^2z^2 - 60x^2yz^3 + 15x^2z^4 + 6xy^5 - 30xy^4z + 60xy^3z^2 - 60xy^2z^3 + 30xyz^4 - 6xz^5 + y^6 - 6y^5z + 15y^4z^2 - 20y^3z^3 + 15y^2z^4 - 6yz^5 + z^6$$

Степень многочлена a по x **`degree`**

```
degree(a,x)
```

6

Соберём вместе члены с определёнными степенями x **`collect`**

```
collect(a,x)
```

$$x^6 + x^5(6y - 6z) + x^4(15y^2 - 30yz + 15z^2) + x^3(20y^3 - 60y^2z + 60yz^2 - 20z^3) + x^2(15y^4 - 60y^3z + 90y^2z^2 - 60yz^3 + 15z^4) + x(6y^5 - 30y^4z + 60y^3z^2 - 60y^2z^3 + 30yz^4 - 6z^5) + y^6 - 6y^5z + 15y^4z^2 - 20y^3z^3 + 15y^2z^4 - 6yz^5 + z^6$$

Многочлен с целыми коэффициентами можно записать в виде произведения таких многочленов (причём каждый сомножитель уже невозможно расфакторизовать дальше, оставаясь в рамках многочленов с целыми коэффициентами).

Существуют эффективные алгоритмы для решения этой задачи - **`factor`**.

```
a=factor(a)
a
```

$$(x+y-z)^6$$

SymPy не сокращает отношения многочленов на их наибольший общий делитель автоматически. Для этого используется функция **cancel**.

```
a=(x**3-y**3)/(x**2-y**2)
a
```

$$\frac{x^3-y^3}{x^2-y^2}$$

```
cancel(a)
```

$$\frac{x^2+xy+y^2}{x+y}$$

SymPy не приводит суммы рациональных выражений к общему знаменателю автоматически. Для этого используется функция **together**.

```
a=y/(x-y)+x/(x+y)
a
```

$$\frac{x}{x+y}+\frac{y}{x-y}$$

```
together(a)
```

$$\frac{x(x-y)+y(x+y)}{(x-y)(x+y)}$$

Функция **simplify** пытается переписать выражение ***в наиболее простом виде***. Это понятие не имеет чёткого определения (в разных ситуациях ***наиболее простыми*** могут считаться разные формы выражения), и не существует алгоритма такого упрощения.

Функция **symplify** работает эвристически, и невозможно заранее предугадать, какие упрощения она попытается сделать. Поэтому её удобно использовать в интерактивных сессиях, чтобы посмотреть, удастся ли ей записать выражение в каком-нибудь разумном виде, но нежелательно использовать в программах. В них лучше применять более специализированные функции, которые выполняют одно определённое преобразование выражения.

```
simplify(a)
```

$$\frac{x^2+y^2}{x^2-y^2}$$

Разложение на элементарные дроби по отношению к *x* и *y* - **`apart`**

```
apart(a,x)
```

$$-\frac{y}{x+y}+\frac{y}{x-y}+1$$

```
apart(a,y)
```

$$\frac{x}{x+y}+\frac{x}{x-y}-1$$

Подставим конкретные численные значения вместо переменных x и y **subs**

```
a=a.subs({x:1,y:2})
a
```

$$-\frac{5}{3}$$

А сколько это будет численно? Метод **n()**

```
a.n()
```

$$-1.66666666666667$$

Элементарные функции

SymPy автоматически применяет упрощения элементарных функция (*которые справедливы во всех случаях*).

```
sin(-x)
```

$$-\sin(x)$$

```
cos(pi/4),tan(5*pi/6)
```

$$\left(\frac{\sqrt{2}}{2},\quad -\frac{\sqrt{3}}{3}\right)$$

SymPy может работать с числами с плавающей точкой, имеющими сколь угодно большую точность. Вот π с 100 значащими цифрами – метод `n(100)`.

```
pi.n(100)
```

$$3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534$$

E - это основание натуральных логарифмов.

```
log(1),log(E)
```

$$(0,\quad 1)$$

```
exp(log(x)),log(exp(x))
```

$$(x,\quad \log(e^x))$$

```
sqrt(0)
```

$$0$$

```
sqrt(x)**4,sqrt(x**4)
```

$$\left(x^2,\quad \sqrt{x^4}\right)$$

Символы могут иметь некоторые свойства. Например, они могут быть положительными. Тогда SymPy может сильнее упростить квадратные корни.

```
p,q=symbols('p q',positive=True)
sqrt(p**2)
```

p

```
sqrt(12*x**2*y),sqrt(12*p**2*y)
```

$\left(2\sqrt{3}\sqrt{x^2 y},\quad 2\sqrt{3}p\sqrt{y}\right)$

Пусть символ *n* будет целым (I - это мнимая единица).

```
n=Symbol('n',integer=True)
simplify(exp(2*pi*I*n))
```

1

```
sin(pi*n)
```

0

Метод **`rewrite`** пытается переписать выражение в терминах заданной функции.

```
cos(x).rewrite(exp),exp(I*x).rewrite(cos)
```

$\left(\frac{e^{ix}}{2}+\frac{1}{2}e^{-ix},\quad i\sin(x)+\cos(x)\right)$

```
asin(x).rewrite(log)
```

$-i\log\left(ix+\sqrt{-x^2+1}\right)$

Функция **`trigsimp`** пытается переписать тригонометрическое выражение в наиболее простом виде. В программах лучше использовать более специализированные функции.

```
trigsimp(2*sin(x)**2+3*cos(x)**2)
```

$\cos^2(x)+2$

Функция **`expand_trig`** разлагает синусы и косинусы сумм и кратных углов.

```
expand_trig(sin(x-y)),expand_trig(sin(2*x))
```

$\left(\sin(x)\cos(y)-\sin(y)\cos(x),\quad 2\sin(x)\cos(x)\right)$

Чаще нужно обратное преобразование - произведений и степеней синусов и косинусов в выражения, линейные по этим функциям.

Например, пусть мы работаем с отрезком ряда Фурье.

```
a1,a2,b1,b2=symbols('a1 a2 b1 b2')
a=a1*cos(x)+a2*cos(2*x)+b1*sin(x)+b2*sin(2*x)
a
```

$a_1\cos(x)+a_2\cos(2x)+b_1\sin(x)+b_2\sin(2x)$

Мы хотим возвести его в квадрат и опять получить отрезок ряда Фурье.

```
a=(a**2).rewrite(exp).expand().rewrite(cos).expand()
a
```

$$\frac{a_1^2}{2}\cos(2x)+\frac{a_1^2}{2}+a_1a_2\cos(x)+a_1a_2\cos(3x)+a_1b_1\sin(2x)+a_1b_2\sin(x)+a_1b_2\sin(3x)+\frac{a_2^2}{2}\cos(4x)+\frac{a_2^2}{2}-a_2b_1\sin(x)+a_2b_1\sin(3x)+a_2b_2\sin(4x)-\frac{b_1^2}{2}\cos(2x)+\frac{b_1^2}{2}+b_1b_2\cos(x)-b_1b_2\cos(3x)-\frac{b_2^2}{2}\cos(4x)+\frac{b_2^2}{2}$$

```
a.collect([cos(x),cos(2*x),cos(3*x),sin(x),sin(2*x),sin(3*x)])
```

$$\frac{a_1^2}{2}+a_1b_1\sin(2x)+\frac{a_2^2}{2}\cos(4x)+\frac{a_2^2}{2}+a_2b_2\sin(4x)+\frac{b_1^2}{2}-\frac{b_2^2}{2}\cos(4x)+\frac{b_2^2}{2}+\left(\frac{a_1^2}{2}-\frac{b_1^2}{2}\right)\cos(2x)+(a_1a_2-b_1b_2)\cos(3x)+(a_1a_2+b_1b_2)\cos(x)+(a_1b_2-a_2b_1)\sin(x)+(a_1b_2+a_2b_1)\sin(3x)$$

Функция **expand_log** преобразует логарифмы произведений и степеней в суммы логарифмов (только для положительных величин);

logcombine производит обратное преобразование.

```
a=expand_log(log(p*q**2))
a
```

$$\log(p)+2\log(q)$$

```
logcombine(a)
```

$$\log\left(pq^2\right)$$

Функция **expand_power_exp** переписывает степени, показатели которых - суммы, через произведения степеней.

```
expand_power_exp(x**(p+q))
```

$$x^px^q$$

Функция **expand_power_base** переписывает степени, основания которых - произведения, через произведения степеней.

```
expand_power_base((x*y)**n)
```

$$x^ny^n$$

Функция **powsimp** выполняет обратные преобразования.

```
powsimp(exp(x)*exp(2*y)),powsimp(x**n*y**n)
```

$$\left(e^{x+2y},\quad (xy)^n\right)$$

Можно вводить функции пользователя. Они могут иметь произвольное число аргументов.

```
f=Function('f')
f(x)+f(x,y)
```

$$f(x)+f(x,y)$$

Структура выражений

Внутреннее представление выражения - это дерево. Функция **srepr** возвращает строку, представляющую его.

```
srepr(x+1)
```

```
"Add(Symbol('x'), Integer(1))"
```

```
srepr(x-1)
```

```
"Add(Symbol('x'), Integer(-1))"
```

```
srepr(x-y)
```

```
"Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```

```
srepr(2*x*y/3)
```

```
"Mul(Rational(2, 3), Symbol('x'), Symbol('y'))"
```

```
srepr(x/y)
```

```
"Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```

Вместо бинарных операций +, *, ** и т.д. можно использовать функции **Add**, **Mul**, **Pow** и т.д.

```
Mul(x,Pow(y,-1))==x/y
```

```
True
```

```
srepr(f(x,y))
```

```
"Function('f')(Symbol('x'), Symbol('y'))"
```

Атрибут **func** - это функция верхнего уровня в выражении, а **args** - список её аргументов.

```
a=2*x*y**2
a.func
```

```
sympy.core.mul.Mul
```

```
a.args
```

$$(2,\quad x,\quad y^2)$$

```
a.args[0]
```

$$2$$

```
for i in a.args:
    print(i)
```

```
2
x
y**2
```

Функция **subs** заменяет переменную на выражение.

```
a.subs(y,2)
```

$8x$

Она может заменить несколько переменных. Для этого ей передаётся список кортежей или словарь.

```
a.subs([(x,pi),(y,2)])
```

8π

```
a.subs({x:pi,y:2})
```

8π

Она может заменить не переменную, а подвыражение - функцию с аргументами.

```
a=f(x)+f(y)
a.subs(f(y),1)
```

$f(x)+1$

```
(2*x*y*z).subs(x*y,z)
```

$2z^2$

```
(x+x**2+x**3+x**4).subs(x**2,y)
```

x^3+x+y^2+y

Подстановки производятся последовательно. В данном случае сначала x заменился на y, получилось y^3+y^2; потом в этом результате y заменилось на x

```
a=x**2+y**3
a.subs([(x,y),(y,x)])
```

x^3+x^2

Если написать эти подстановки в другом порядке, результат будет другим.

```
a.subs([(y,x),(x,y)])
```

y^3+y^2

Но можно передать функции subs ключевой параметр **`simultaneous=True`**, тогда подстановки будут производиться одновременно. Таким образом можно, например, поменять местами x и y

```
a.subs([(x,y),(y,x)],simultaneous=True)
```

x^3+y^2

Можно заменить функцию на другую функцию.

```
g=Function('g')
a=f(x)+f(y)
a.subs(f,g)
```

$$g(x) + g(y)$$

Метод **`replace`** ищет подвыражения, соответствующие образцу (содержащему произвольные переменные), и заменяет их на заданное выражение (оно может содержать те же произвольные переменные).

```
a=Wild('a')
(f(x)+f(x+y)).replace(f(a),a**2)
```

$$x^2 + (x + y)^2$$

```
(f(x,x)+f(x,y)).replace(f(a,a),a**2)
```

$$x^2 + f(x, y)$$

```
a=x**2+y**2
a.replace(x,x+1)
```

$$y^2 + (x + 1)^2$$

Соответствовать образцу должно целое подвыражение, это не может быть часть сомножителей в произведении или меньшая степень в большей.

```
a=2*x*y*z
a.replace(x*y,z)
```

$$2xyz$$

```
(x+x**2+x**3+x**4).replace(x**2,y)
```

$$x^4 + x^3 + x + y$$

Решение уравнений

```
a,b,c,d,e,f=symbols('a b c d e f')
```

Уравнение записывается как функция Eq с двумя параметрами. Функция **`solve`** возвращает список решений.

```
solve(Eq(a*x,b),x)
```

$$\left[\frac{b}{a}\right]$$

Впрочем, можно передать функции **solve** просто выражение. Подразумевается уравнение, что это выражение равно 0.

```
solve(a*x+b,x)
```

$$\left[-\frac{b}{a}\right]$$

Квадратное уравнение имеет 2 решения.

```
solve(a*x**2+b*x+c,x)
```

$$\left[\frac{1}{2a}\left(-b+\sqrt{-4ac+b^2}\right),\quad -\frac{1}{2a}\left(b+\sqrt{-4ac+b^2}\right)\right]$$

Система линейных уравнений.

```
solve([a*x+b*y-e,c*x+d*y-f],[x,y])
```

$$\left\{x:\frac{-bf+de}{ad-bc},\quad y:\frac{af-ce}{ad-bc}\right\}$$

Функция **roots** возвращает корни многочлена с их кратностями.

```
roots(x**3-3*x+2,x)
```

$$\{-2:1,\quad 1:2\}$$

Функция **solve_poly_system** решает систему полиномиальных уравнений, строя их базис Грёбнера.

```
p1=x**2+y**2-1
p2=4*x*y-1
solve_poly_system([p1,p2],x,y)
```

$$\left[\left(4\left(-1-\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(-\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad -\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right),\right.$$
$$\left(-4\left(-1+\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(\sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad \sqrt{-\frac{\sqrt{3}}{4}+\frac{1}{2}}\right),$$
$$\left(4\left(-1-\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(-\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad -\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right),$$
$$\left.\left(-4\left(-1+\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\left(\sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}+1\right),\quad \sqrt{\frac{\sqrt{3}}{4}+\frac{1}{2}}\right)\right]$$

Ряды

```
exp(x).series(x,0,5)
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \mathcal{O}\left(x^5\right)$$

Ряд может начинаться с отрицательной степени.

```
cot(x).series(x,n=5)
```

$$\frac{1}{x} - \frac{x}{3} - \frac{x^3}{45} + \mathcal{O}\left(x^5\right)$$

И даже идти по полу целым степеням.

```
sqrt(x*(1-x)).series(x,n=5)
```

$$\sqrt{x} - \frac{x^{\frac{3}{2}}}{2} - \frac{x^{\frac{5}{2}}}{8} - \frac{x^{\frac{7}{2}}}{16} - \frac{5x^{\frac{9}{2}}}{128} + \mathcal{O}\left(x^5\right)$$

```
log(gamma(1+x)).series(x,n=6).rewrite(zeta)
```

$$-\gamma x + \frac{\pi^2 x^2}{12} - \frac{x^3\zeta\left(3\right)}{3} + \frac{\pi^4 x^4}{360} - \frac{x^5\zeta\left(5\right)}{5} + \mathcal{O}\left(x^6\right)$$

Подготовим 3 ряда.

```
sinx=series(sin(x),x,0,8)
sinx
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}\left(x^8\right)$$

```
cosx=series(cos(x),x,n=8)
cosx
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \mathcal{O}\left(x^8\right)$$

```
tanx=series(tan(x),x,n=8)
tanx
```

$$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \mathcal{O}\left(x^8\right)$$

Произведения и частные рядов не вычисляются автоматически, к ним надо применить функцию **`series`**.

```
series(tanx*cosx,n=8)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}\left(x^8\right)$$

```
series(sinx/cosx,n=8)
```

$$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \mathcal{O}\left(x^8\right)$$

А этот ряд должен быть равен 1. Но поскольку **sinx** и **cosx** известны лишь с ограниченной точностью, мы получаем 1 с той же точностью.

```
series(sinx**2+cosx**2,n=8)
```

$$1 + \mathcal{O}\left(x^8\right)$$

Здесь первые члены сократились, и ответ можно получить лишь с меньшей точностью.

```
series((1-cosx)/x**2,n=6)
```

$$\frac{1}{2} - \frac{x^2}{24} + \frac{x^4}{720} + \mathcal{O}\left(x^6\right)$$

Ряды можно дифференцировать и интегрировать.

```
diff(sinx,x)
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \mathcal{O}\left(x^7\right)$$

```
integrate(cosx,x)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}\left(x^9\right)$$

Можно подставить ряд (если он начинается с малого члена) вместо переменной разложения в другой ряд. Вот ряды для **sin(tan(x))** и **tan(sin(x))**

```
st=series(sinx.subs(x,tanx),n=8)
st
```

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{55x^7}{1008} + \mathcal{O}\left(x^8\right)$$

```
ts=series(tanx.subs(x,sinx),n=8)
ts
```

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{107x^7}{5040} + \mathcal{O}\left(x^8\right)$$

```
series(ts-st,n=8)
```

$$\frac{x^7}{30} + \mathcal{O}\left(x^8\right)$$

В ряд нельзя подставлять численное значение переменной разложения (а значит, нельзя и строить график). Для этого нужно сначала убрать $\mathcal{O}$ член, превратив отрезок ряда в многочлен.

```
a=sinx.removeO()
```

```
a.subs(x,0.1)
```

0.0998334166468254

Производные

```
a=x*sin(x+y)
diff(a,x)
```

$$x\cos\left(x+y\right) + \sin\left(x+y\right)$$

```
diff(a,y)
```

$$x\cos\left(x+y\right)$$

Вторая производная по x и первая по y

```
diff(a,x,2,y)
```

$$-x\cos\left(x+y\right) + 2\sin\left(x+y\right)$$

Можно дифференцировать выражения, содержащие неопределённые функции.

```
a=x*f(x**2)
b=diff(a,x)
b
```

$$2x^2 \frac{d}{d\xi_1} f(\xi_1)\Big|_{\xi_1=x^2} + f(x^2)$$

Что это за «зверь» такой получился?

```
print(b)
```

```
2*x**2*Subs(Derivative(f(_xi_1), _xi_1), (_xi_1,), (x**2,)) + f(x**2)
```

Функция **Derivative** представляет не вычисленную производную. Её можно вычислить методом doit.

```
a=Derivative(sin(x),x)
Eq(a,a.doit())
```

$$\frac{d}{dx}\sin(x) = \cos(x)$$

Интегралы

```
integrate(1/(x*(x**2-2)**2),x)
```

$$\frac{1}{4}\log(x) - \frac{1}{8}\log(x^2-2) - \frac{1}{4x^2-8}$$

```
integrate(1/(exp(x)+1),x)
```

$$x - \log(e^x+1)$$

```
integrate(log(x),x)
```

$$x\log(x) - x$$

```
integrate(x*sin(x),x)
```

$$-x\cos(x) + \sin(x)$$

```
integrate(x*exp(-x**2),x)
```

$$-\frac{e^{-x^2}}{2}$$

```
a=integrate(x**x,x)
a
```

$$\int x^x\,dx$$

Получился неопределённый интеграл.

```
print(a)
```

```
Integral(x**x, x)
```

```
a=Integral(sin(x),x)
Eq(a,a.doit())
```

$$\int \sin(x)\,dx = -\cos(x)$$

Определённые интегралы.

```
integrate(sin(x),(x,0,pi))
```

$$2$$

oo - это ∞.

```
integrate(exp(-x**2),(x,0,oo))
```

$$\frac{\sqrt{\pi}}{2}$$

```
integrate(log(x)/(1-x),(x,0,1))
```

$$-\frac{\pi^2}{6}$$

Суммирование рядов

```
summation(1/n**2,(n,1,oo))
```

$$\frac{\pi^2}{6}$$

```
summation((-1)**n/n**2,(n,1,oo))
```

$$-\frac{\pi^2}{12}$$

```
summation(1/n**4,(n,1,oo))
```

$$\frac{\pi^4}{90}$$

Не вычисленная сумма обозначается Sum.

```
a=Sum(x**n/factorial(n),(n,0,oo))
Eq(a,a.doit())
```

$$\sum_{n=0}^{\infty} \frac{x^n}{n!} = e^x$$

Пределы

```
limit((tan(sin(x))-sin(tan(x)))/x**7,x,0)
```

$$\frac{1}{30}$$

Это простой предел, он считается разложением числителя и знаменателя в ряды. А если в x=0, то появляется особая точка.

Найдем односторонние пределы.

```
limit((tan(sin(x))-sin(tan(x)))/(x**7+exp(-1/x)),x,0,'+')
```

$$\frac{1}{30}$$

```
limit((tan(sin(x))-sin(tan(x)))/(x**7+exp(-1/x)),x,0,'-')
```

$$0$$

Дифференциальные уравнения

```
t=Symbol('t')
x=Function('x')
p=Function('p')
```

Первого порядка.

```
dsolve(diff(x(t),t)+x(t),x(t))
```

$$x(t) = C_1 e^{-t}$$

Второго порядка.

```
dsolve(diff(x(t),t,2)+x(t),x(t))
```

$$x(t) = C_1 \sin(t) + C_2 \cos(t)$$

Система уравнений первого порядка.

```
dsolve((diff(x(t),t)-p(t),diff(p(t),t)+x(t)))
```

$$[x(t) = C_1 \sin(t) + C_2 \cos(t), \quad p(t) = C_1 \cos(t) - C_2 \sin(t)]$$

Линейная алгебра

```
a,b,c,d,e,f=symbols('a b c d e f')
```

Матрицу можно построить из списка списков.

```
M=Matrix([[a,b,c],[d,e,f]])
M
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
M.shape
```

$$(2, \ 3)$$

Матрица-строка.

```
Matrix([[1,2,3]])
```

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix}$$

Матрица-столбец.

```
Matrix([1,2,3])
```

$$\begin{bmatrix}1\\2\\3\end{bmatrix}$$

Можно построить матрицу из функции.

```
def g(i,j):
    return Rational(1,i+j+1)
Matrix(3,3,g)
```

$$\begin{bmatrix}1 & \frac{1}{2} & \frac{1}{3}\\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4}\\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5}\end{bmatrix}$$

Или из неопределённой функции.

```
g=Function('g')
M=Matrix(3,3,g)
M
```

$$\begin{bmatrix}g(0,0) & g(0,1) & g(0,2)\\ g(1,0) & g(1,1) & g(1,2)\\ g(2,0) & g(2,1) & g(2,2)\end{bmatrix}$$

```
M[1,2]
```

$$g(1,2)$$

```
M[1,2]=0
M
```

$$\begin{bmatrix}g(0,0) & g(0,1) & g(0,2)\\ g(1,0) & g(1,1) & 0\\ g(2,0) & g(2,1) & g(2,2)\end{bmatrix}$$

```
M[2,:]
```

$$\begin{bmatrix}g(2,0) & g(2,1) & g(2,2)\end{bmatrix}$$

```
M[:,1]
```

$$\begin{bmatrix}g(0,1)\\g(1,1)\\g(2,1)\end{bmatrix}$$

```
M[0:2,1:3]
```

$$\begin{bmatrix}g(0,1) & g(0,2)\\ g(1,1) & 0\end{bmatrix}$$

Единичная матрица.

```
eye(3)
```

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

Матрица из нулей.

```
zeros(3)
```

$$\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$$

```
zeros(2,3)
```

$$\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$$

Диагональная матрица.

```
diag(1,2,3)
```

$$\begin{bmatrix}1 & 0 & 0\\0 & 2 & 0\\0 & 0 & 3\end{bmatrix}$$

```
M=Matrix([[a,1],[0,a]])
diag(1,M,2)
```

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & a & 1 & 0\\0 & 0 & a & 0\\0 & 0 & 0 & 2\end{bmatrix}$$

Операции с матрицами.

```
A=Matrix([[a,b],[c,d]])
B=Matrix([[1,2],[3,4]])
A+B
```

$$\begin{bmatrix}a+1 & b+2\\c+3 & d+4\end{bmatrix}$$

```
A*B,B*A
```

$$\left(\begin{bmatrix}a+3b & 2a+4b\\c+3d & 2c+4d\end{bmatrix}, \begin{bmatrix}a+2c & b+2d\\3a+4c & 3b+4d\end{bmatrix}\right)$$

```
A*B-B*A
```

$$\begin{bmatrix}3b-2c & 2a+3b-2d\\-3a-3c+3d & -3b+2c\end{bmatrix}$$

```
simplify(A**(-1))
```

$$\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

```
det(A)
```

$$ad - bc$$

Собственные значения и векторы

```
x=Symbol('x',real=True)
```

```
M=Matrix([[(1-x)**3*(3+x),4*x*(1-x**2),-2*(1-x**2)*(3-x)],
          [4*x*(1-x**2),-(1+x)**3*(3-x),2*(1-x**2)*(3+x)],
          [-2*(1-x**2)*(3-x),2*(1-x**2)*(3+x),16*x]])
M
```

$$\begin{bmatrix} (-x+1)^3(x+3) & 4x\left(-x^2+1\right) & (-x+3)\left(2x^2-2\right) \\ 4x\left(-x^2+1\right) & -(-x+3)(x+1)^3 & (x+3)\left(-2x^2+2\right) \\ (-x+3)\left(2x^2-2\right) & (x+3)\left(-2x^2+2\right) & 16x \end{bmatrix}$$

```
det(M)
```

$$0$$

Значит, у этой матрицы есть нулевое подпространство (она обращает векторы из этого подпространства в 0). Базис этого подпространства.

```
v=M.nullspace()
len(v)
```

$$1$$

Оно одномерно.

```
v=simplify(v[0])
v
```

$$\begin{bmatrix} -\frac{2}{x-1} \\ \frac{2}{x+1} \\ 1 \end{bmatrix}$$

Проверим.

Проверим.

```
simplify(M*v)
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Собственные значения и их кратности.

```
M.eigenvals()
```

$$\left\{0 : 1, \quad -\left(x^2+3\right)^2 : 1, \quad \left(x^2+3\right)^2 : 1\right\}$$

Если нужны не только собственные значения, но и собственные векторы, то нужно использовать метод **`eigenvects`**. Он возвращает список кортежей. В каждом из них нулевой элемент - собственное значение, первый - его кратность, и последний - список собственных векторов, образующих базис (их столько, какова кратность).

```
v=M.eigenvects()
len(v)
```

3

```
for i in range(len(v)):
    v[i][2][0]=simplify(v[i][2][0])
v
```

$$\left[\left(0, \quad 1, \quad \left[\begin{bmatrix} -\frac{2}{x-1} \\ \frac{2}{x+1} \\ 1 \end{bmatrix}\right]\right), \quad \left(-\left(x^2+3\right)^2, \quad 1, \quad \left[\begin{bmatrix} \frac{x}{2}+\frac{1}{2} \\ \frac{x+1}{x-1} \\ 1 \end{bmatrix}\right]\right), \quad \left(\left(x^2+3\right)^2, \quad 1, \quad \left[\begin{bmatrix} \frac{x-1}{x+1} \\ -\frac{x}{2}+\frac{1}{2} \\ 1 \end{bmatrix}\right]\right)\right]$$

Проверим.

```
for i in range(len(v)):
    z=M*v[i][2][0]-v[i][0]*v[i][2][0]
    pprint(simplify(z))
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Жорданова нормальная форма

```
M=Matrix([[Rational(13,9),-Rational(2,9),Rational(1,3),Rational(4,9),Rational(2,3)],
          [-Rational(2,9),Rational(10,9),Rational(2,15),-Rational(2,9),-Rational(11,15)],
          [Rational(1,5),-Rational(2,5),Rational(41,25),-Rational(2,5),Rational(12,25)],
          [Rational(4,9),-Rational(2,9),Rational(14,15),Rational(13,9),-Rational(2,15)],
          [-Rational(4,15),Rational(8,15),Rational(12,25),Rational(8,15),Rational(34,25)]])
M
```

$$\begin{bmatrix} \frac{13}{9} & -\frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \frac{2}{3} \\ -\frac{2}{9} & \frac{10}{9} & \frac{2}{15} & -\frac{2}{9} & -\frac{11}{15} \\ \frac{1}{5} & -\frac{2}{5} & \frac{41}{25} & -\frac{2}{5} & \frac{12}{25} \\ \frac{4}{9} & -\frac{2}{9} & \frac{14}{15} & \frac{13}{9} & -\frac{2}{15} \\ -\frac{4}{15} & \frac{8}{15} & \frac{12}{25} & \frac{8}{15} & \frac{34}{25} \end{bmatrix}$$

Метод M.jordan_form() возвращает пару матриц, матрицу преобразования P и собственно жорданову форму J: $M=PJP^{-1}$

```
P,J=M.jordan_form()
J
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1-i & 0 \\ 0 & 0 & 0 & 0 & 1+i \end{bmatrix}$$

```
P=simplify(P)
P
```

$$\begin{bmatrix} -2 & \frac{10}{9} & 0 & \frac{5i}{12} & -\frac{5i}{12} \\ -2 & -\frac{5}{9} & 0 & -\frac{5i}{6} & \frac{5i}{6} \\ 0 & 0 & \frac{4}{3} & -\frac{3}{4} & -\frac{3}{4} \\ 1 & \frac{10}{9} & 0 & -\frac{5i}{6} & \frac{5i}{6} \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Проверим.

```
Z=P*J*P**(-1)-M
simplify(Z)
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

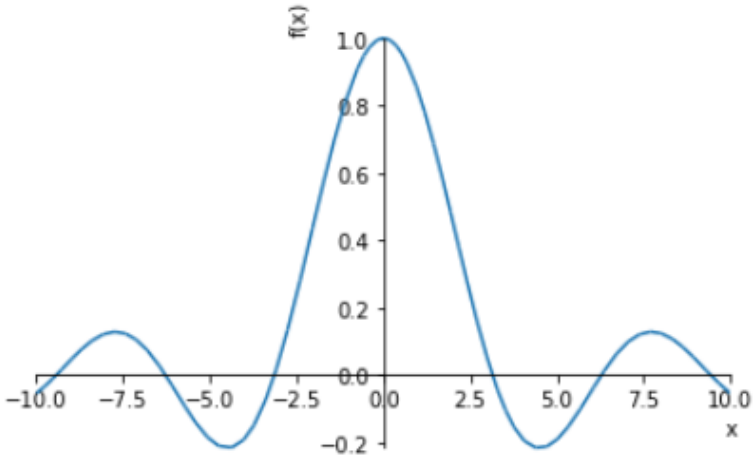
Графики

SymPy использует matplotlib. Однако он распределяет точки по x адаптивно, а не равномерно.

%matplotlib inline

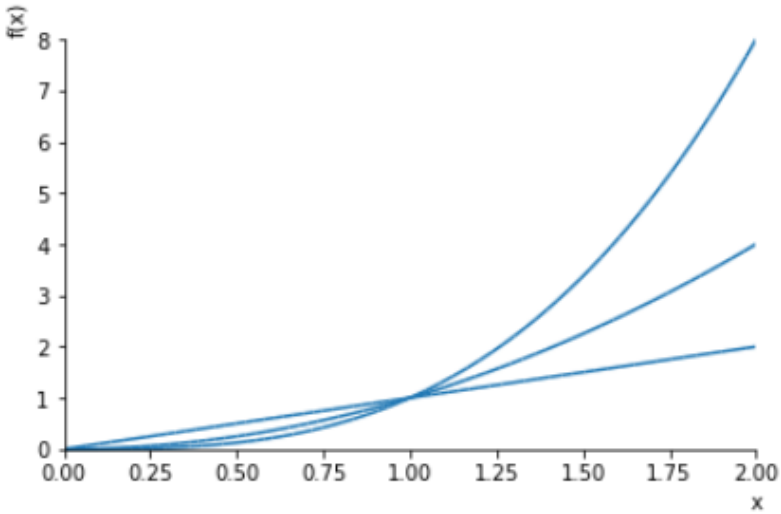
Одна функция.

plot(sin(x)/x,(x,-10,10))



Несколько функций.

plot(x,x**2,x**3,(x,0,2))

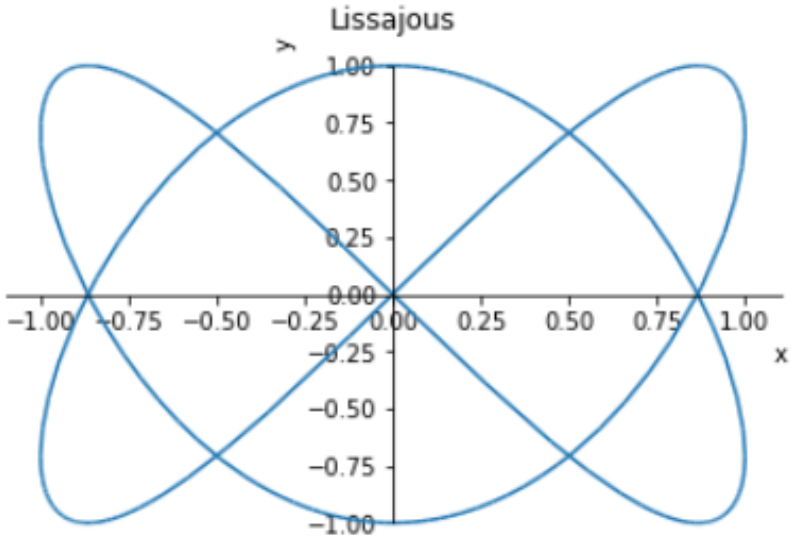


Другие функции надо **импортировать** из пакета **sympy.plotting.**

```
from sympy.plotting import
     (plot_parametric,plot_implicit,
        plot3d,plot3d_parametric_line,
                  plot3d_parametric_surface)
```

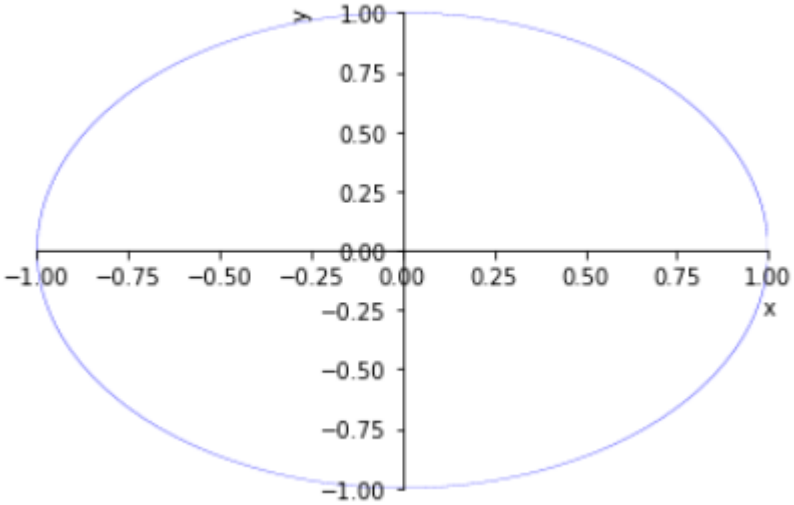
Параметрический график - фигура Лиссажу.

```
t=Symbol('t')
plot_parametric(sin(2*t),cos(3*t),(t,0,2*pi),
          title='Lissajous',xlabel='x',ylabel='y')
```



Неявный график - окружность.

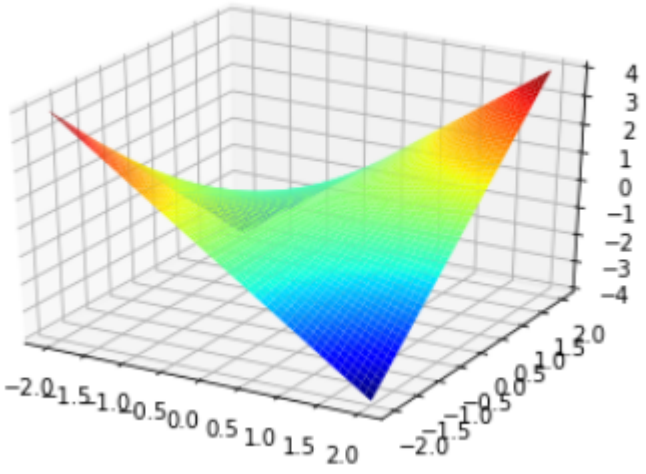
```
plot_implicit(x**2+y**2-1,(x,-1,1),(y,-1,1))
```



Поверхность.

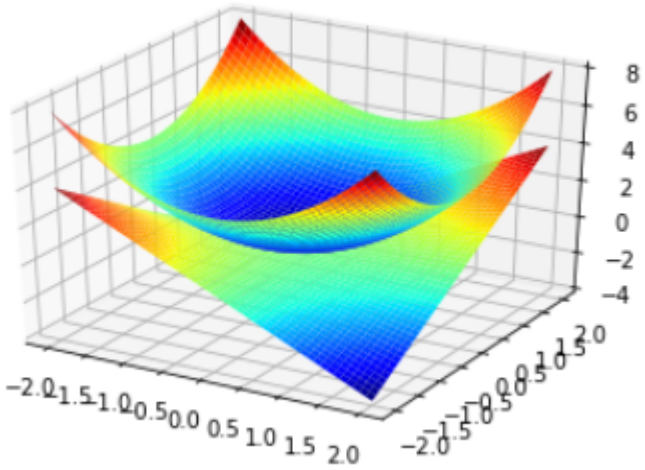
Если она строится не inline, а в отдельном окне, то её можно вертеть мышкой.

```
plot3d(x*y,(x,-2,2),(y,-2,2))
```



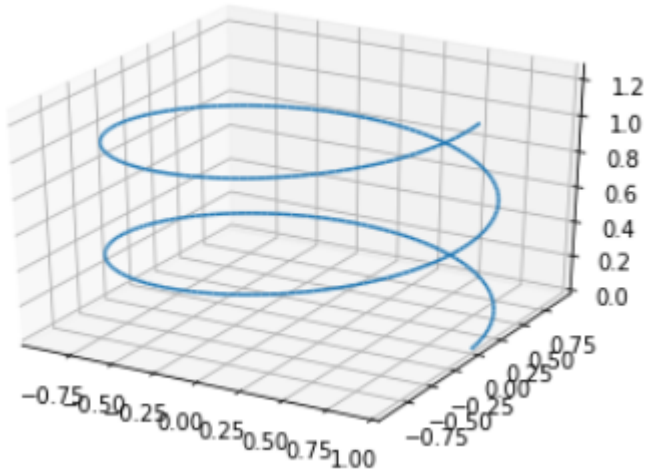
Несколько поверхностей.

```
plot3d(x**2+y**2,x*y,(x,-2,2),(y,-2,2))
```



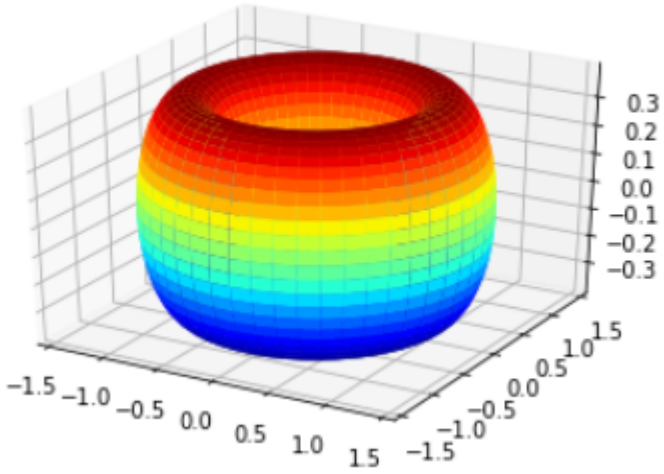
Параметрическая пространственная линия - спираль.

```
a=0.1
plot3d_parametric_line(cos(t),sin(t),a*t,(t,0,4*pi))
```



Параметрическая поверхность - тор.

```
u,v=symbols('u v')
a=0.4
plot3d_parametric_surface((1+a*cos(u))*cos(v),
                          (1+a*cos(u))*sin(v),a*sin(u),
                          (u,0,2*pi),(v,0,2*pi))
```



(По https://www.inp.nsk.su/~grozin/python/b25_sympy.html)

XXX. Список использованных источников

Ниже перечислены адреса сайтов, которые послужили основой для написания данного пособия.

1. https://pythonworld.ru/numpy
2. https://docs.scipy.org/doc/numpy/reference/generated/numpy.set_printoptions.html
3. https://docs.scipy.org/doc/numpy/reference/
4. https://docs.scipy.org/doc/numpy/reference/routines.math.html
5. https://pythonworld.ru/samouchitel-python
6. http://old.pynsk.ru/posts/2015/Nov/09/matematika-v-python-preobrazovanie-fure/
7. https://github.com/whitehorn/Scientific_graphics_in_python
8. https://metanit.com/python/
9. http://nbviewer.jupyter.org/github/whitehorn/Scientific_graphics_in_python/tree/master/
10. http://jenyay.net/Programming/Python
11. https://habrahabr.ru/company/mailru/blog/228379/)/
12. https://github.com/sympy/sympy/wiki/Pretty-Printing
13. http://www.asmeurer.com/sympy_doc/dev-py3k/tutorial/tutorial.ru.html
14. http://www.realcoding.net/articles
15. http://docplayer.ru/51061131-Vvedenie-v-nauchnyy-python.html
16. https://ru.wikiversity.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B8_%D0%BD%D0%B0%D1%83%D1%87%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8

%D1%8F_%D0%BD%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA
%D0%B5_Python/%C2%A75
17. https://eax.me/python-matplotlib
18. http://jenyay.net/Matplotlib/Widgets
19. https://matplotlib.org/2.0.0/examples/animation/animate_decay.html
20. http://www.asmeurer.com/sympy_doc/dev-py3k/tutorial/tutorial.ru.html
21. http://www.sympy.org/en/index.html