Доля П.Г.
Харьковский Национальный Университет
Факультет математики и информатики
2016 г.

Введение в научный Python.

Python представляет собой интерпретируемый объектно-ориентированный язык и интерактивную среду для разработки программ. С его помощью можно разрабатывать приложения с графическим интерфейсом, работать с базами данных, создавать Web-сайты и делать многое другое. Язык программирования Python обладает ясным и понятным синтаксисом и хорош для программирования математических вычислений. В некотором смысле Python это мощный калькулятор. Даже если вы только поверхностно с ним знакомы, вы можете использовать его для выполнения невероятных вещей. Python реализован практически во всех операционных системах, и большинство его модулей распространяется бесплатно.

В этом пособии мы бегло познакомимся с основными конструкциями языка и наиболее часто используемыми функциями из стандартной библиотеки. Мы не будем пытаться охватить всё – недостающие факты вы легко найдете в справочной документации. Основное внимание будет уделено рассмотрению пакетов, используемых в научных вычислениях – `numpy`, `scipy`, `matplotlib` и `sympy`. В них реализованы классические численные алгоритмы решения уравнений, задач линейной алгебры, вычисления определенных интегралов, аппроксимации, решения дифференциальных уравнений и их систем. Пакет `matplotlib` обладают хорошо развитыми возможностями визуализации двумерных и трехмерных данных. Основой пакетов `NumPy` и `SciPy` являются численные расчеты, но и символьные вычисления, основанные на библиотеке `SymPy`, Python умеет делать хорошо. Решение уравнений и систем, интегрирование и дифференцирование, вычисление пределов, разложение в ряд и суммирование рядов, поиск решения дифференциальных уравнений и систем, упрощение выражений – вот далеко не полный перечень «аналитических» возможностей пакета `sympy`.

Пособие предназначено в первую очередь для знакомства с математическими возможностями Python. Читайте текст и выполняйте примеры. Во многих случаях все пояснения дает сам код решения задачи. Надеемся, что по окончании выполнения последнего примера вы начнете применять `Scientific Python` для решения ваших задач.

Большая часть, изложенного в пособии материала, доступна студентам младших курсов физико–математических факультетов университетов, а также студентам технических вузов, прослушавшим курс высшей математики. Сложности могут возникнуть только при чтении параграфов, содержащих решение прикладных задач, поскольку они предполагают некоторое знакомство с соответствующими областями знаний.

Оглавление

1. Введение

Вначале вы должны установить программную среду на свой компьютер. Полагаем, что вы уже это сделали. Если нет, то ее можно взять с сайта разработчиков https://www.python.org/downloads/. Выберите на этой странице желаемую версию Python и скачайте инсталляционный пакет, соответствующий вашей операционной системе (он бесплатный). Запустите скачанный файл и следуйте инструкциям мастера установки. Затем вам потребуется установить научные библиотеки. Их коллекция называется `SciPy` (Scientific Library for Python) и установка этой библиотеки потребует от вас некоторых усилий.

Другой способ установить «научный» Python состоит в использовании бесплатного дистрибутива Anaconda (https://www.continuum.io/downloads). Это самый простой способ установить сразу Python и научные библиотеки. Кроме всего прочего, вы получите неплохую интегрированную оболочку `Spyder` предназначенную для разработки и выполнения программ.

Полагаем, что вы установили пакет Anaconda или Python с научными библиотеками numpy, scipy, matplotlib и sympy, или обе среды сразу (для обучения последний вариант более предпочтительный).

Далее мы будем полагать, что вы работаете в одной из версий ОС Windows. С обычным Python–ом можно работать по–разному. Один способ состоит в использовании командной строки, когда команды и программы запускаются на выполнение из консольного окна. Однако для изучения языка, а также создания и редактирования текстов программ, лучше использовать среду разработки IDLE, поставляемую вместе с дистрибутивом Python. Она состоит из интерпретатора команд, называемого Python Shell, и специального текстового редактора IDLE для Python. В окне интерпретатора Python Shell можно выполнять те же команды, что и в консоли.

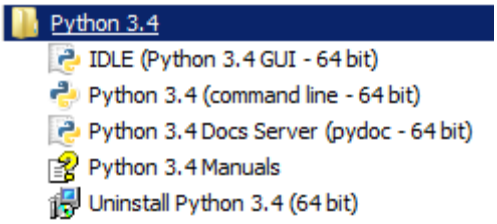
Кроме интерпретатора Python Shell среды IDLE, многие разработчики пользуются интерактивной оболочкой IPython. Она предоставляет расширенный список возможностей по сравнению со стандартным Python shell. Другой удобной средой для выполнения научных исследований является IPython Notebook (переименованной в Jupyter Notebook). Она выполняется в web–браузерах и по своим функциональным возможностям напоминает оболочку системы Mathematica. Последние версии IPython и Jupyter Notebook вы найдете среди программ, установленных вместе с пакетом Anaconda.

При разработке больших Python программ, состоящих из нескольких файлов и классов, использование интерпретатора командной строки не всегда удобно. Большинство программистов пользуется какой–либо интегрированной средой разработки. Одной из наиболее распространенных программ является Spyder – бесплатная среда разработки и выполнения Python программ, созданная специально для выполнения научных расчетов. Она содержит редактор, предназначенный для разработки Python программ, а также оба интерпретатора Python Shell и IPython, любой из которых можно использовать для выполнения команд и программ. Spyder входит в состав пакета Anaconda. Установив пакет Anaconda, вы получите все библиотеки и инструменты, необходимые для работы с нашим пособием.

Имейте в виду, что новые версии языка Python и оболочки, предназначенные для разработки его программ, выпускаются довольно регулярно. Поэтому ко времени чтения вами нашего пособия наверняка появятся новые версии языка и его оболочек. В нашем пособии мы будем опираться на стандарт языка Python 3.4.

1.1 Первые шаги. Числа, переменные, операции, выражения.

Если вы установили стандартный Python в Windows, то у вас в меню Пуск–Все Программы должна появиться папка примерно следующего содержания.



Если запустить программу `Python 3.4(command line – 64 bit)`, то Python запустится внутри консольного окна.

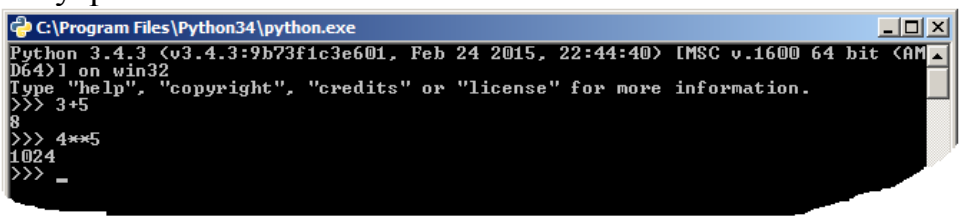


Рис. 1.1 Внешний вид Python в окне консоли.

В нем можно вводить команды и наблюдать результаты вычислений. Например, после знака >>> введите `3+5` и нажмите клавишу Enter. Вы увидите:

```
>>> 3+5
8
```

Значок '>>>' является символом приглашения на ввод команд. Инструкции нужно вводить справа от него (без пробелов). Для выполнения команды следует нажать клавишу Enter. Если инструкция возвращает значение, то на следующей строке сразу отобразится результат. Затем отображается приглашение для ввода новой команды.

В стандартный дистрибутив Python входит интегрированная среда разработки IDLE, в которой редактировать и выполнять программы удобнее, чем в консоли. Вы можете запустить среду разработки IDLE, пройдя по следующему пути:

`Пуск → Все программы → Python 3.X → IDLE(Python 3.X GUI).`

На экране появляется окно с заголовком Python 3.X Shell (вместо 3.X будет стоять номер вашей версии). Оно предназначено для ввода команд и отображения результатов вычислений. На первых порах мы будем работать в нем.

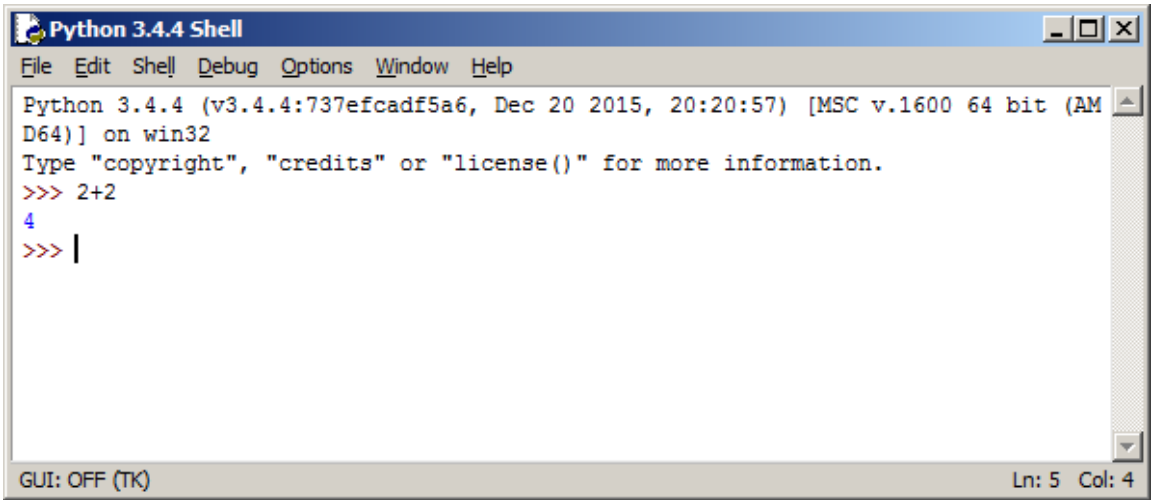


Рис. 1.2 Внешний вид окна Python Shell

Аналогично предыдущему, наберите в окне Python Shell после символа >>> команду 2+2 и нажмите клавишу Enter. Вы увидите:

```
>>> 2+2
4
```

Здесь также значок >>> является символом приглашения на ввод команд. Возможности Python Shell по выполнению Python программ аналогичны консольному варианту, но бо̀льшие при редактировании и создании программ.

Если вы установили на свой компьютер только пакет Anaconda, то среды разработки IDLE в нем нет, но имеется интегрированная среда разработки и выполнения программ Spyder. Мы сейчас кратко расскажем, как ее использовать вместо IDLE.

В Windows Spyder вы найдете в меню Все программы–Anaconda3(64-bit)–Spyder. Его главное окно разделено на несколько панелей. Если вы не меняли настроек, то слева у вас будет расположено окно редактора, а в правой нижней части будут расположены одна над другой панели двух консолей: Python и IPython (Interactive Python). Команды можно выполнять в любой из них. Нас сейчас интересует консоль Python, работа в которой подобна работе в Python Shell.

Настроим Spyder для работы преимущественно с Python консолью. Для этого в правой нижней части окна Spyder найдите и захватите мышью строку панели с заголовком Console. Перетащите панель влево вверх, расположив ее поверх редактора. Затем, если нужно, переместите правую границу окна консоли. Окно Spider примет следующий вид.

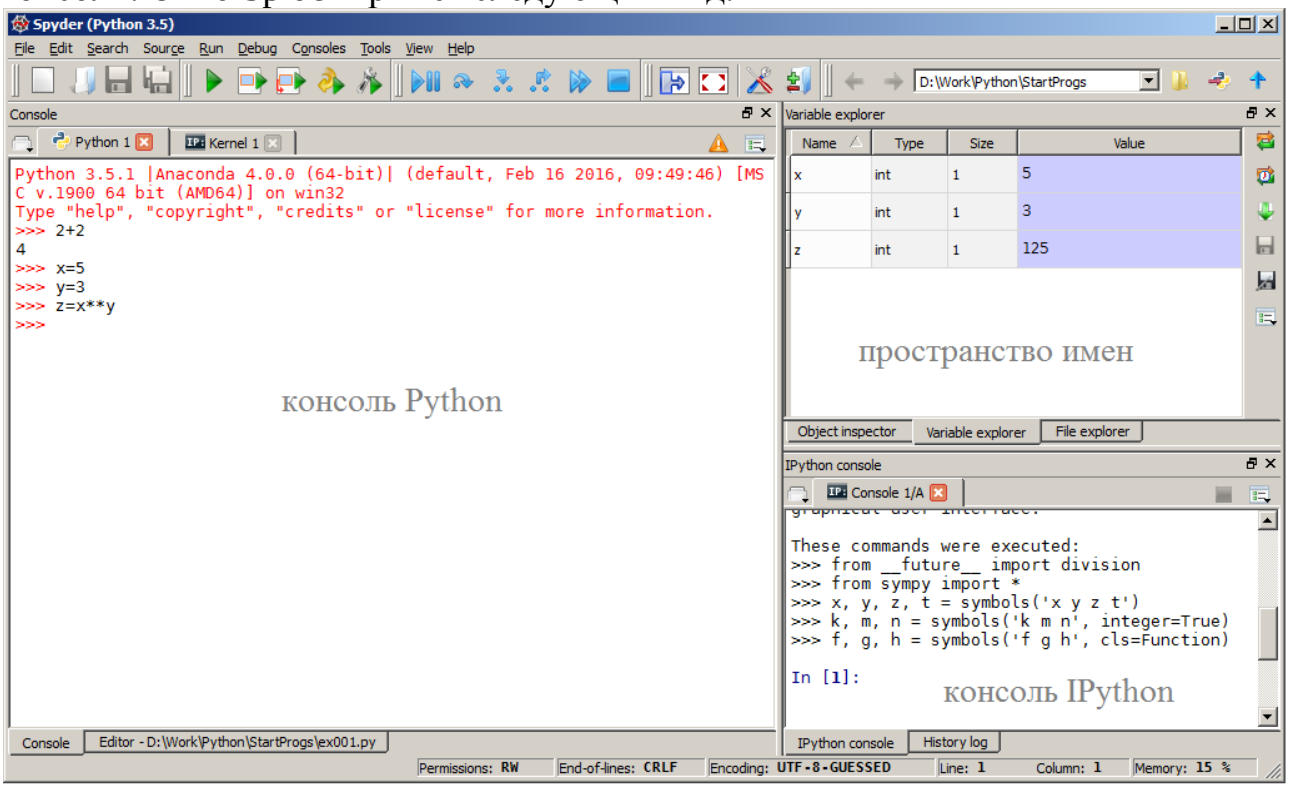


Рис. 1.3 Внешний вид окна Spyder

Не обращайте пока внимание на другие панели Spyder. О них мы расскажем тогда, когда вы больше будете знать о языке программирования Python.

В этом параграфе мы будем предполагать, что вы работаете в интегрированной среде IDLE, но если ее у вас нет, то будем полагать, что примеры вы выполняете в консоли Spyder. Для инструкций языка Python это не имеет значения. Небольшие различия между Python Shell и консолью Spyder имеются при редактировании команд. В тех местах этого параграфа, в которых описываются способы ввода инструкций (а не сами инструкции), мы будем подразумевать команды редактирования Python Shell. Если вы пользуетесь консолью Spyder, то вы должны самостоятельно проверить, что эти команды работоспособны, или по справочной системе узнать их аналоги. В любой из двух описанных нами сред после ввода команды следует нажимать клавишу Enter. Если инструкция возвращает значение, то на следующей строке отображается результат, а затем появляется приглашение >>> для ввода новой команды.

Начнем знакомиться с языком Python. В нашем пособии команды (то, что мы вводим) будем записывать шрифтом Arial, а результат их выполнения – шрифтом Courier New.

Обычно одна команда Python вводится на отдельной строке после знака приглашения '>>>'. Но можно набирать несколько команд в одной строке, тогда их нужно разделять символом ';' (точка с запятой).

```
>>>x=5;y=10; x+y     # Три инструкции на одной строке
15
```

Текст, стоящий после символа #, является комментарием. Комментарии предназначены для вставки пояснений в текст программы, и интерпретатор полностью их игнорирует. В языке Python имеется только однострочный комментарий, который начинается символом #.

Строки выводятся функцией print(...).

```
>>> print("Hello world!")
Hello world!
```

Строка – это набор символов, заключенных в двойные или одинарные кавычки. Можно просто ввести строку и нажать клавишу Enter для получения результата:

```
>>>"Привет, мир!"
'Привет, мир!'
```

Результирующая строка отобразится в кавычках. Этого не произойдет, если выводить строку с помощью функции print (...).

Учитывая возможность получить результат сразу после ввода команды, окно Python Shell можно использовать в качестве многофункционального калькулятора.

```
>>> 12*45+30
570
```

Результат вычисления последней инструкции сохраняется в переменной '_'(одно подчеркивание). Это позволяет производить дальнейшие расчеты без ввода предыдущего результата. Вместо него достаточно ввести символ подчеркивания.

```
>>> 134*4
536
>>> _*6
3216
>>> _/4
804.0
```

Перенести курсор в одну из верхних секций интерпретатора нельзя. Для повторения одной из предыдущих команд можно использовать клавишу <↑> (или <↓>), нажав ее нужное количество раз. В текущей командной строке будут по очереди отображаться предыдущие (следующие) команды. Когда появится нужная команда, вы можете ее отредактировать и нажать клавишу Enter.

Для математических операций в выражениях используются следующие обозначения:

- * – это умножение; 34*89 это 34 умножить на 89;
- ** – обозначает возведение в степень; 2**3 это 2 в третьей степени. Степень может быть дробной:
  ```
  >>>3**0.5
  1.7320508075688772
  ```
  Возведение целого числа в целую степень даёт целое число, если показатель степени ≥0, и число с плавающей точкой, если он <0.
- / – это деление; 15/6 это 15 делить на 6 с результатом 2.5. Деление целых чисел всегда даёт результат с плавающей точкой, даже если они делятся нацело (в версиях Python 3.X).
- // – целочисленное деление; 15//6 вернет 2.
- % – это операция вычисления остатка от деления.
  ```
  >>> 5%2
  1
  >>> 5.4%2.1    # остаток от деления
  1.2000000000000002
  ```
- ( ) – (круглые скобки) служат для указания порядка вычислений: (6+9)/4 это 6+9 деленное на 4. Не используйте квадратные или фигурные скобки для указания порядка вычислений. Кроме этого, круглые скобки используются для указания аргументов функций, а также для создания кортежей. Кортежем является любая последовательность элементов, разделенных запятыми и заключенная в круглые скобки.
- [ ] – (квадратные скобки) обозначают список; [1,2,3,4] - это список из четырех чисел. К элементам списка можно обращаться по индексу. Нумерация индексов начинается с нуля.
  ```
  >>> x=[10,20,30,40]
  >>> x[2]
  30
  >>> x[0]
  10
  ```

- Для запоминания результатов вычислений удобно использовать переменные. При определении переменной используется знак равенства **=**. Слева от знака равенства стоит имя переменной, а справа – ее значение или выражение для вычисления. Одно значение можно присвоить сразу нескольким переменным:
  ```
  >>>x = y = z = 0
  ```
  Имя переменной (идентификатор) не должно совпадать с именами ключевых слов (операторов и функций) Python. Имя должно начинаться с буквы, может содержать буквы, цифры и символ подчеркивания. Недопустимо включать в имена переменных пробелы и специальные знаки. Регистр букв учитывается при именовании переменных. При попытке использовать переменную, которой не присвоено никакого значения, генерируется ошибка.
- _ (подчеркивание) представляет результат последнего вычисления. Эта переменная используется только для чтения и ей нельзя присвоить значение явно.
- == – проверка на равенство; a ==5 возвращает `True`, если `a` равно `5`.
- != – проверка на неравенство; a!=5 возвращает `True`, если `a` не равно `5`.
- <,<=,>,>= обозначают неравенства; можно использовать привычные из математики сравнения, вроде $1 < x < 2$:
  ```
  >>>x=3.4
  >>>1<x<5
  True
  >>>1<x<3
  False
  ```

Набранная формула, как правило, не заканчивается никаким символом. Если выполнялось присваивание, то результат операции сразу не отображается.

```
>>> z=1<x<8
>>> z
True
```

С логическими значениями `True` и `False` можно выполнять логические операции and, or, not.

```
>>> True and False
False
>>> not(True or False)
False
```

Логические (булевы) значения `True` и `False` можно использовать в арифметических выражениях, в которых они интерпретируются как `1` и `0`.

```
>>> a=4
>>> b=True
>>> a+b
5
>>>(3+True)**0.5          # извлечение квадратного корня из 4
2.0
```

Допустимо следующее выражение

```
>>> (3*(4>3)+(5!=6))**0.5 # извлечение квадратного корня из 4
2.0
```
Однако помните, что арифметические операции имеют приоритет перед логическими. Сравните предыдущий пример со следующим.
```
>>> (3*(4>3)+5!=6)**0.5   # извлечение квадратного корня из 1
1.0
```
В языке Python встроенных функций немного. Большинство надо импортировать (загружать из специальных файлов, называемых модулям). Элементарные математические функции, такие как `sin`, `cos`, `log` и другие, собраны в модуле `math`. Чтобы функции стали доступными, модуль следует импортировать в рабочее пространство исполнительной системы командой `import имя_модуля`. Затем для вызова функции нужно использовать команду `имя_модуля.имя_функции(...)`.
```
>>> import math
>>> math.exp(1.0)
2.718281828459045
```
Используя функцию `dir(имя_модуля)` можно узнать имена функций, которые доступны из этого модуля.
```
>>> dir(math)
['__doc__', '__loader__', ..., 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', '...', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```
Имена с двумя подчеркиваниями являются внутренними. Все остальные – это функции, а также константы такие, как π и e.
```
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
>>> math.log(math.e)
1.0
```
Можно (и рекомендуется) импортировать только те функции, которые вы собираетесь использовать. Для этого нужно указать их имена в команде
`from имя_модуля import имя_функции, имя_функции,...`
В этом случае перед именем функции (или константы) имя модуля указывать не требуется. Например,
```
>>>from math import sin,pi
>>>pi
3.141592653589793
>>>sin(pi/2)
>>>1.0
```
Для импорта всех функций модуля можно использовать команду
```
>>>from math import *
```

При импорте функции из модуля ей можно назначить другое имя. В этом случае одновременно с добавлением функции создается ее новое имя. Например

```
>>> from math import factorial as f
>>> f(5)
120
```

Python умеет работать с комплексными числами. Для записи мнимой части используется **j** (а не **i**). При этом операция умножения перед мнимой единицей не используется. Например, запись x=1-2*j приведет к ошибке.

```
>>> x=3+4j
>>> y=4-7j
>>> z=x*y
>>> z
(40-5j)
>>> z/y
(3+4.000000000000001j)
>>> x**2
(-7+24j)
```

Прежде, чем продолжать дальше, скажем несколько слов о переменных и объектах Python. Каждая переменная хранит информацию о каком–либо объекте. Каждый объект относится к какому-нибудь типу данных. Типы в Python называются классами, и пользователь имеет возможность создавать собственные классы. Фактически класс представляет собой коллекцию данных и функций, которые называются атрибутами и методами. Атрибут – это переменная, метод – это функция. В языке Python все является объектами: числа, списки, функции, модули и т.д. Объект, созданный на основе некоторого класса, называется экземпляром класса. Экземпляры одного класса отличаются один от другого значениями своих атрибутов. Для доступа к объекту используются переменные. Переменная создается при присваивании ей значения с помощью знака равенства '='. Во время присваивания в переменной сохраняется ссылка на объект (адрес объекта в памяти компьютера). При доступе к атрибутам и методам используется точка, которая разделяет имя объекта/переменной и имя атрибута или метода. Удалить переменную из пространства имен исполнительной системы (очистить область памяти, занимаемую переменной, и удалить ссылку на эту область) можно с помощью команды `del`:

```
del Имя1[, ... , ИмяN].
```

Например,

```
>>> x = 10; x
10
>>> del x; x
Ошибка!!!
```

Комплексная переменная z, созданная выше, тоже является объектом, и она имеет свои атрибуты и методы. Например, в выражении `z.real`, `real` — это атрибут объекта `z`, а `z.conjugate()` – метод. Следующие команды вычисляют

вещественную и мнимую часть комплексной переменной z, а также сопряженное значение. Обычно атрибут возвращает уже хранящееся в объекте значение, а метод вычисляет что – то новое.

```
>>> z.real
40.0
>>> z.imag
-5.0
>>> z.conjugate()
(40+5j)
```

Другие функции, предназначенные для работы с комплексными переменными, собраны в модуле cmath:

```
>>> from cmath import sqrt
>>> sqrt(z)
(6.336848141682612-0.39451789661100817j)
```

В Python пользователь может создавать свои функции. Создание функции начинается со слова def, за которым следует имя функции, аргументы в круглых скобках и двоеточие. Затем на новой строке с отступом начинаются инструкции. Оператор return значение возвращает из функции значение и является командой выхода.

```
>>> def f(x):
        return x**2

>>> f(3)
9
```

Здесь была использована многострочная команда, признаком которой является символ ':' (двоеточие). После ввода двоеточия и нажатия клавиши Enter курсор переходит на следующую строку. В Python Shell автоматически вставляется отступ (в консоли Spyder вставляется троеточие и отступ следует делать самостоятельно) и интерпретатор ожидает дальнейшего ввода (продолжения многострочной команды). Чтобы сообщить о конце ввода многострочной команды, необходимо дважды нажать клавишу Enter.

Многострочные команды встречаются при программировании циклов, в условных операторах и в ряде других случаев. Следующий пример демонстрирует использование оператора цикла.

```
>>> for n in range(1,3):
        print(n)
1
2
```

Результаты работы программы можно вывести с помощью функции print(), общий формат которой имеет следующий вид:

```
print([Объекты][,sep=' '][,end ='\n'] [,file=sys.stdout])
```

Здесь элементы, указанные в квадратных скобках, являются необязательными. Параметр sep задает строку–разделитель между объектами, а параметр end задает строку (или один символ), которая выводится после последнего объекта. Функция print() преобразует свой аргумент в строку и отображает его в окне

исполнительной системы. После вывода строки автоматически добавляется символ перевода строки (если не указано другое значение в параметре end).

```
>>> print ("Строка 1"); print ("Строка 2")
Строка 1
Строка 2
```

Если необходимо вывести результат на той же строке, то объекты указываются через запятую.

```
>>> print("Строка 1", "Строка 2")
Строка 1 Строка 2
```

Между выводимыми строками автоматически вставляется пробел. С помощью параметра sep можно указать другой символ–разделитель.

```
>>>print("Строка1", "Строка2", sep="")
Строка 1Строка 2
```

Здесь мы вывели строки без пробела между ними. После вывода данных в конце добавляется символ новой строки. Если последующий вывод должен продолжиться в той же строке, то в параметре end следует указать другой символ, например, пробел.

```
>>>print("Строка 1", "Строка 2", end=" "); print ("Строка 3")
Строка 1 Строка 2 Строка 3
```

Если необходимо вывести большой блок текста, то его следует разместить между утроенными кавычками. В этом случае текст сохраняет свое форматирование.

```
>>> print("""Строка 1
Строка 2
Строка 3""")
Строка 1
Строка 2
Строка 3
```

До сих пор все наши примеры состояли из нескольких строк, и все команды записывались и выполнялись в окне Python Shell. Опишем некоторые его особенности, связанные с вводом и выполнением команд.

Если ввести несколько первых букв инструкции и нажать комбинацию клавиш Ctrl+Пробел, то будет отображен список, из которого можно выбрать нужное имя. Если при открытом списке дальше вводить буквы, то список имен будет фильтроваться, и показываться будут идентификаторы, начинающиеся с этих букв. Выбирать идентификатор необходимо с помощью клавиш управления курсором '↑' и '↓'. После выбора не следует нажимать клавишу Enter, иначе это приведет к выполнению команды. Просто вводите инструкцию дальше, а список закроется.

Для вставки последней введенной инструкции Python Shell можно использовать комбинацию клавиш Alt+P (это не работает в консоли Spyder). Каждое следующее нажатие этих комбинаций клавиш будет вставлять предыдущую инструкцию, а Alt–N – следующую. Для еще более быстрого повторного ввода инструкции следует предварительно ввести ее первые буквы.

В этом случае перебираться будут только инструкции, начинающиеся с этих букв. Комбинация клавиш Ctrl – z отменяет в Python Shell последний ввод.

Для того, чтобы получить справку о функции, достаточно написать `help(имя_функции)` или `help(имя_модуля.имя_функции)`.

Обычный командный режим работы (ввод одной или нескольких строк) не очень удобен. Поэтому в Python предусмотрена возможность записывать инструкции в текстовый файл и затем выполнять его. Имя этого файла должно иметь расширение `py` (или `pyw` для программ с оконным интерфейсом). Среда IDLE содержит встроенный текстовый редактор, специально предназначенный для создания Python программ. Для примера создадим и выполним программу, которая выводит текст "Привет, мир!". В Python Shell для создания файла с программой в меню File выберите пункт New File. В открывшемся окне редактора введите текст

```
print ("Привет, мир !")
```

Сохраните файл с именем, например, `hello.py`.

Запускается программа на выполнение выбором в окне редактора пункта меню Run – Run Module или нажатием клавиши `F5`. Результат выполнения будет отображен в окне Python Shell. Запустить программу можно также с помощью двойного щелчка мыши по имени файла. В этом случае результат отобразится в консоли Windows. Однако после вывода окно консоли сразу закроется и вы, скорее всего, ничего не успеете заметить. Чтобы предотвратить автоматическое закрытие окна, в программу необходимо добавить вызов функции `input()`, которая будет ожидать нажатия клавиши Enter и не даст программе сразу завершиться.

Аналогично создается и выполняется программа в Spyder. Для этого активируйте окно редактора Spyder, щелкнув мышью внизу по закладке Editor (на рис. 1.3 она видна слева внизу). Откроется окно редактора с пустым файлом. Введите в нем текст программы, сохраните файл, и запустите на выполнение командой меню Run – Run (или `F5`).

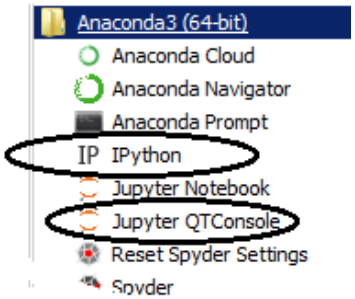
1.2 Среды разработки и выполнения Python программ.

Мы уже говорили, что в стандартный комплект поставки Python входит интегрированная среда разработки IDLE, в которой редактировать и выполнять программы удобнее, чем в консоли. Краткое (и не полное) описание ее возможностей было приведено в предыдущем параграфе. Однако жизнь не стоит на месте и в последние годы появилось много новых интегрированных сред, предназначенных для создания, редактирования и выполнения Python программ. Здесь мы опишем некоторые из них.

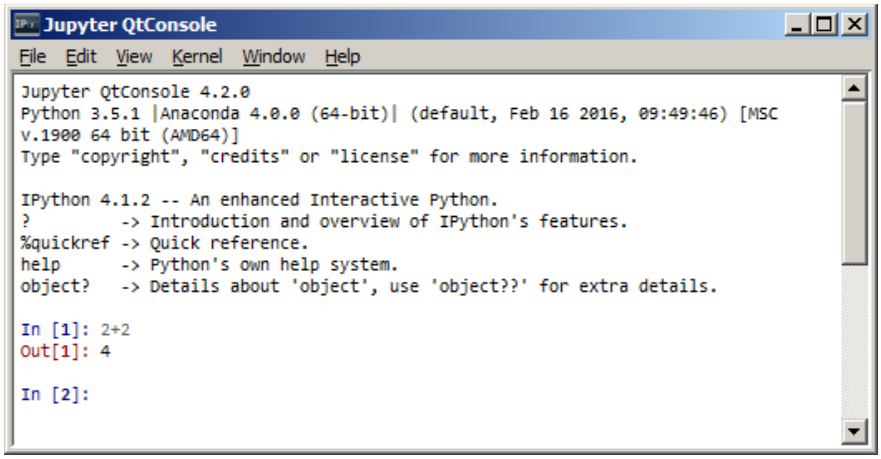
IPython и **Jupyter QtConsole**. IPython (Interactive Python, улучшенная консоль Python) предоставляет расширенный список возможностей по редактированию команд. Вместо символов приглашения консоли '>>>', делящих сессию работы с Python на части, IPython делит документ на последовательность ячеек. Некоторые из них содержат код, другие текст и графику. Ячейки с кодом

помечаются метками `In[n]`. Введя в такой ячейке команду и нажав клавишу Enter, вы отправляете код интерпретатору Python. Результат появляется в поле вывода, которое помечается меткой `Out[n]`.

Еще одна усовершенствованная консоль включена в состав пакета Anaconda с названием Jupyter QtConsole. Если вы установили пакет Anaconda, то обе программы вы найдете в меню `Все программы – Anaconda3 (64-bit)`



В интерпретаторе Jupyter QtConsole команды выполняются нажатием комбинации клавиш Shift – Enter (для однострочных инсрукций можно также использовать Enter). Отличие в редактировании команд в обеих консолях небольшое и мы кратко опишем работу с программой Jupyter QtConsole. Запустите ее.



Всё, что можно было делать в Python Shell, можно делать и здесь: выполнять команды Python, пользоваться справкой, импортировать модули. Но есть несколько дополнительных возможностей. Рассмотрим некоторые из них.

1. Быстрый просмотр информации по объекту. Добавив вопросительный знак к имени объекта, вы получите информацию о его типе, классе, пространстве имён, а также строку документации.

```
In[1]: s = 'строка'
In[2]: s?
```

Более полную информацию можно получить, добавив два вопросительных знака.

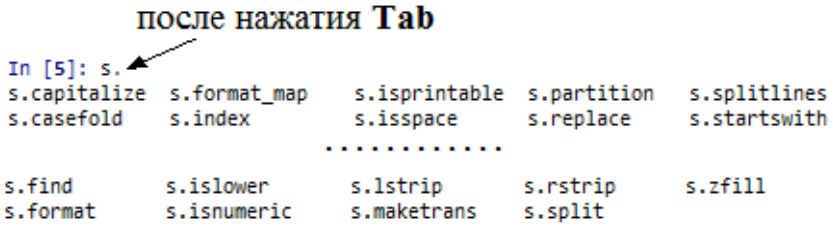
Команда `pdef fun` (или `%pdef fun`) подскажет, какие аргументы нужны для вызова функции или метода `fun`.

```
In[3]: pdef list
```

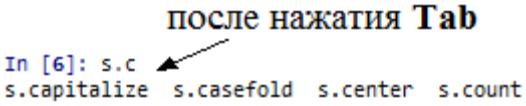
Посмотреть документацию по функции можно командой `pdoc fun`. Вывести информацию по функции можно командой `pinfo fun`. Часто доступен

исходный код функции. Чтобы его посмотреть, можно использовать команду `psource fun`.

2. Функции автодополнения команд. Введя имя объекта, точку и, нажав клавишу Tab, вы получите список всех атрибутов и методов объекта.



Введя первую букву желаемого атрибута или метода и опять нажав Tab, вы получите сокращенный список только тех атрибутов и методов, которые начинаются с этой буквы. Например, введите после точки букву `'c'` и вы увидите следующее:



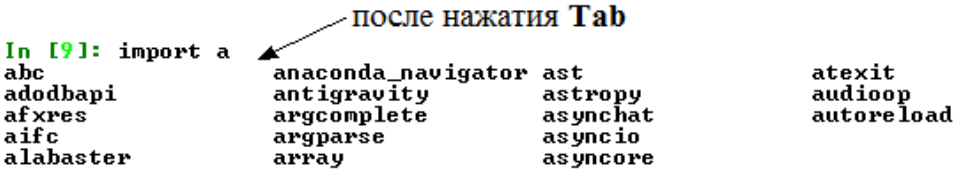
Если ввести следующую букву `'e'` и нажать Tab, то ввод будет завершен и появится инструкция `s.center`, которая представляет имя метода `s.center(width)`. Если в качестве аргумента ввести число, например `20`, то метод создаст строку длиной 20 символов и в ее середине поместит строку `s`.

Если сразу ввести несколько первых букв команды, а вариант автозавершения один, то нажатие Tab автоматически отобразит соответствующую команду в поле ввода.

Автодополнение работает почти везде, где оно имеет смысл. Например, команда

import a<Tab>

отобразит список модулей, имена которых начинаются с буквы `'a'`, и которые могут быть импортированы.



3. Встроенные «магические» команды. Многие из них выполняют те же функции, которые имеются в консоли. Например, команда `%cd` покажет текущий каталог, а команда `%cd 'dir'` сделает текущей папку `'dir'`. Команда `%load имя_файла` загрузит код в окно консоли. В следующем примере после выполнения команды `%load 'hello.py'` нужно еще дважды нажать Enter, чтобы завершить выполнение кода загруженного файла.

```
In[1]: %cd 'D:\Work\Python\StartProgs'
D:\Work\Python\StartProgs
In[2]: %load 'hello.py'
In[3]: # load 'hello.py'
  ...: print("Привет мир!")
```

```
   ...: input()
   ...:
Привет мир!
Out[3]: ''
In[3]:
```

Другие магические команды служат для расширения возможностей IPython. Обычно такие команды начинаются с символа '%' (процент), за которым следует имя команды. Но если нет конфликта с глобальными именами, то команды можно вводить без процента. Команда

%del имя_переменной

удаляет переменную из пространства имен IPython. Команда

%reset

(без аргументов) очищает пространство имен. Команда

%who

выводит список имен доступных переменных. У нее есть опции, например, чтобы вывести переменные только типа `int`, следует написать:

%who int

Команда

%whos

печатает имена, типы и значения всех объектов из пространства имен.

Чтобы переменная была доступна после перезапуска IPython, ее нужно сохранить в профайле командой `%store имя_переменной`. Чтобы посмотреть все сохраненные переменные, надо выполнить команду `store` без параметров. Для восстановления переменных, которые были ранее сохранены в профайле, надо выполнить команду

%store –r

Проверить, что переменные восстановились, можно командой `%whos`. Для очистки всего, что находится в профайле, используется команда

%store –z

При следующем запуске IPython пространство имен будет пустым.

Полный список магических команд можно получить с помощью команды

%lsmagic

Для получения справки по конкретной магической команде достаточно поставить знак вопроса в конце ее имени.

%run?

Обратите внимание, что в Jupyter QtConsole справка открывается в дополнительном окне, расположенном поверх командного окна интерпретатора. Чтобы вернуться в окно документа следует нажать клавишу Esc. В стандартной консоли Ipython справка отображается в ее окне.

Если напечатать `%` (процент) и нажать клавишу Tab, то система автодополнения также выведет полный список магических команд. Он весь не поместится в окне, но вы можете передвигаться по нему, нажимая клавишу Tab.

Опишем еще несколько часто используемых магических команд.

Команда `%run` имя_файла запускает файл на выполнение.

```
In[1]: %cd 'D:\Work\Python\StartProgs'
D:\Work\Python\StartProgs
In[2]: run hello.py
Привет мир!
```

Команда `%save имя_файла номера_строк` сохраняет в файл команды входных полей, номера которых перечислены.

```
In[3]: a=3; b=4; print(a**b)
81
In[4]: %save 'testsave.py' 3
The following commands were written to file `testsave.py`:
a=3; b=4; print(a**b)
```

Файл сохраняется в текущем каталоге. Вот пример сохранения нескольких строк кода.

```
In[10]: from math import *
In[11]: a=3; b=4; c=a**b
In[12]: x=pi/6; y=sin(x)
In[13]: print(c,y)
81 0.49999999999999994
In[14]: %save 'testsave2.py' 10-13
The following commands were written to file `testsave2.py`:
from math import *
a=3; b=4; c=a**b
x=pi/6; y=sin(x)
print(c,y)
```

Здесь в файл `testsave2.py` были сохранены команды входных ячеек `In[10] – In[13]`.

Команда `%%writefile имя_файла` сохраняет содержимое (код) своей ячейки в файл.

```
In[13]: %%writefile 'testsave3.py'
   ...x=6;
   ...y=5;
   ...z=x*(y+2)
Writing testsave3.py
```

Кстати, для того, чтобы создать в Jupyter QtConsole такую многострочную ячейку, в конце первой строки нужно нажать комбинацию клавиш Ctrl – Enter, а последующие строки завершать нажатием клавиши Enter. Для завершения ввода и выполнения кода ячейки следует нажать комбинацию клавиш Shift – Enter.

У команды `%%writefile` имеется опция `'-a'` или `'--append'`, которая позволяет добавлять код ячейки в уже существующий файл. Добавление выполняется без вставки вспомогательных символов (пробелов или переводов строк) так, что первая инструкция ячейки «пристыковывается» к последним символам файла. Чтобы этого не происходило, перед сохраняемым кодом можно вставлять пустую строку.

```
In[15]: %%writefile -a 'testsave3.py'
        ...
        ...z=x*(y+2**x)
Appending to testsave3.py
```

Обратите внимание на то, что магическая команда `%%writefile` содержит впереди два символа процента.

Имеется большое количество других магических команд. С ними вы можете познакомиться самостоятельно по справочной системе.

4. Механизм истории команд. Кроме традиционных стрелок ↑ и ↓ (стрелка вверх и вниз), последовательно пролистывающих введенные ранее команды, можно просмотреть список предыдущих команд, используя магическую команду `%history`. Кстати, команды ↑ и ↓ (стрелка вверх и вниз) можно использовать после введения нескольких букв. Тогда пролистывание будет выполняться только по предыдущим командам, которые начинались с этих букв.

Если последняя команда возвращала результат, то обратиться к нему (результату) можно, используя символ `'_'` (одно подчеркивание). Можно использовать `'__'` и `'___'` (двойное и тройное подчеркивание) для обращения к предпоследнему и пред-предпоследнему результатам.

```
In[1]: from math import sqrt
In[2]: a=3
In[3]: b=4
In[4]: a**b
Out[4]: 81
In[5]: sqrt(_)                  # одно подчеркивание
Out[5]: 9.0
In[6]: sqrt(sqrt(__))           # два подчеркивания
Out[6]: 3.0
```

Переменная `'_i'` (одинарное подчеркивание + `i`) содержит строку последней команды.

```
In[7]: a**b
Out[7]: 81
In[8]: len(_i)
Out[8]: 4
```

Помимо прочего, метка ячейки ввода `In[n]` содержит строку своей команды, а метка `Out[n]` содержат результат. Доступ к результату поля вывода `Out[n]` позволяет использовать его значение в последующих командах.

```
In[9]: Out[8]**3
Out[9]: 64
```

Команда `%recall` повторяет предыдущую команду, а инструкция `%recall n` повторяет команду из поля ввода с номером `n` (из ячейки `In[n]`).

```
In[10]: %recall 7
In[11]: a**b
```

Команда %recall n1-n2 повторяет команды из полей ввода In[n1] – In[n2].

```
In[11]: a=3
In[12]: b=4
In[13]: c=a**b+1
In[14]: print(c)
82
In[15]: % recall 11-14
In[16]: a=3
     ... b=4
     ... c=a**b+1
     ... print(c)
```

5. Дополнительные возможности.

Магическая команда %pprint включает и выключает режим «красивой» печати. Она полезна при совместном использовании с командой %precision N, которая задает точность при отображении вещественных чисел (N – количество десятичных цифр после десятичной точки).

```
In[17]: %pprint              # включить режим «красивой» печати
Pretty printing has been turned ON
In[18]: from math import pi
In[19]: %precision 0         # отображать 0 десятичных цифр
Out[19]: '%.0f'
In[20]: pi
Out[20]: 3                   # приближение числа π
In[21]: a=pi**2; a           # вычисление с процессорной точностью, но
Out[21]: 10                  # отображение 0 десятичных цифр
In[22]: %precision 2         # отображать 2 десятичных цифры
Out[22]: '%.2f'
In[23]: a           # показать число с точностью 2 – х десятичных цифр
Out[23]: 9.87
In[24]: %precision           # включить точность по умолчанию
Out[24]: '%r'
In[25]: a                    # показать число с точностью по умолчанию
Out[25]: 9.869604401089358
In[26]: pi**2                # сравнить результаты
Out[26]: 9.869604401089358
```

В приведенном примере значение a вычислялось как π^2 при установленной точности 0 десятичных цифр, отобразилось значение 10, но в памяти вычисление выполнялось с процессорной точностью. Поэтому для печати значения переменной a при другой точности пересчет не потребовался.

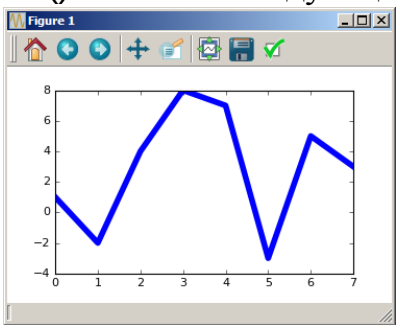
Опции %i и %e включают режимы отображения целой части и экспоненциальной формы представления чисел.

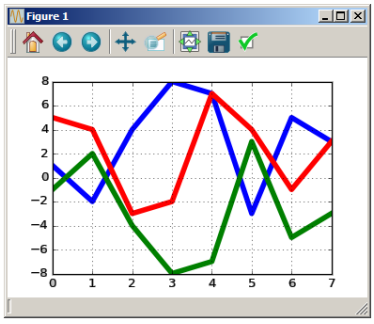
```
In[27]: %precision %i
Out[27]: '%i'
In[28]: a
Out[28]: 9
In[29]: %precision %e
Out[29]: '%e'
In[30]: a
Out[30]: 9.869604e+00
In[31]: %pprint              # выключить режим «красивой» печати
Pretty printing has been turned OFF
```

Имеется еще одна важная магическая команда `%matplotlib`, которая управляет способом отображения графических команд модуля matplotlib. Ее поведение может немного различаться для различных версий консоли IPython.

Построим график в стандартной IPython консоли.

```
In[1]: from matplotlib.pylab import *
In[2]: plot([1,-2,4,8,7,-3,5,3],linewidth=3);
In[3]: show()              # следующее окно слева
```





Инструкция `plot()` строит график в памяти, а команда `show()` отображает его в отдельном окне. При этом приглашение на ввод следующей команды не появится, пока вы не закроете графическое окно. Ели перед приведенным выше кодом выполнить команду

```
In[4]: %matplotlib qt
```

то инструкцию `show()` можно не использовать, графическое окно откроется автоматически и в документе появится метка `In[n]` новой ячейки. Можно выполнить еще несколько графических команд и их результат отобразится в том же (открытом) графическом окне.

```
In[5]: plot([-1,2,-4,-8,-7,3,-5,-3], linewidth=3)
In[6]: plot([5,4,-3,-2,7,4,-1,3], linewidth=3)
In[7]: grid(True)
```

Будет построено несколько кривых и добавлены элементы оформления (см. предыдущий рисунок справа). Пока графическое окно открыто все графические команды работают с ним.

График можно сохранить в файл, используя командную кнопку Save the figer этого окна.

Поведение среды Jupyter QtConsole после команды %matplotlib qt такое же, как у стандартной консоли IPython. Однако теперь доступна еще инструкция:

%matplotlib inline

Она включает режим отображения графики прямо в текущем документе. Конечно, для того, чтобы графические функции были доступны, их следует импортировать.

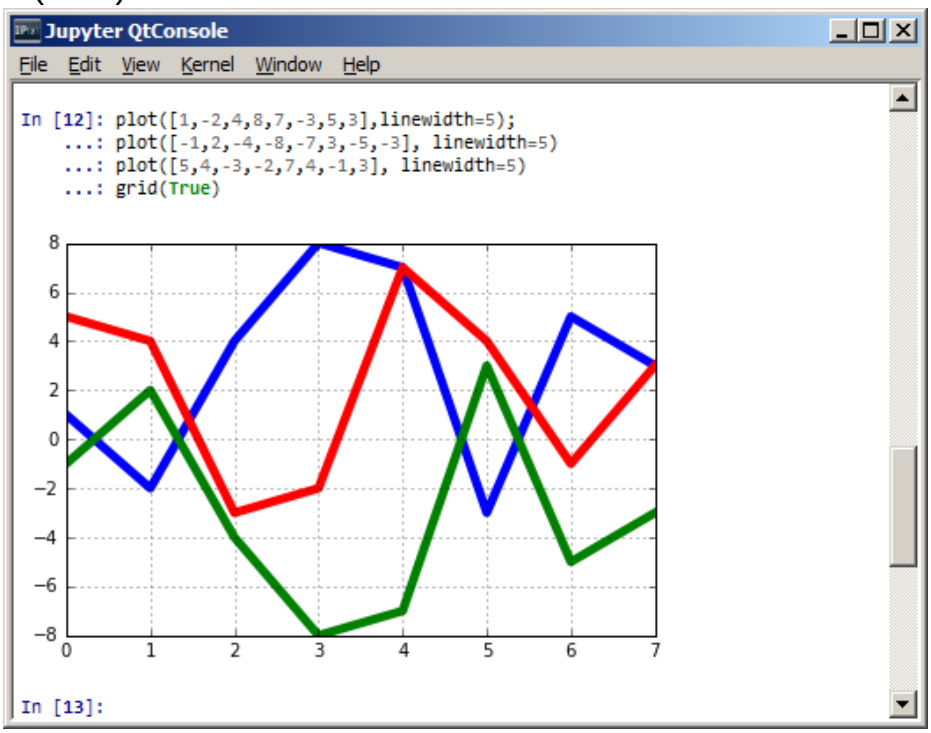
from matplotlib.pylab import *

Тогда команда

plot([1,-2,4,8,7,-3,5,3],linewidth=5);

построит график ломаной прямо в документе. Если в ячейке ввести несколько графических команд, то результат их работы отобразится на одном графике.

```
In[12]:plot([1,-2,4,8,7,-3,5,3],linewidth=5);
   ...plot([-1,2,-4,-8,-7,3,-5,-3], linewidth=5)
   ...plot([5,4,-3,-2,7,4,-1,3], linewidth=5)
   ...grid(True)
```



Вы можете переключаться из режима `'qt'` в режим `'inline'` и обратно. В режиме `'qt'` команда `show()` показывает график в отдельном окне, а в режиме `'inline'` никак себя не проявляет – все графики будут встроенными в документ.

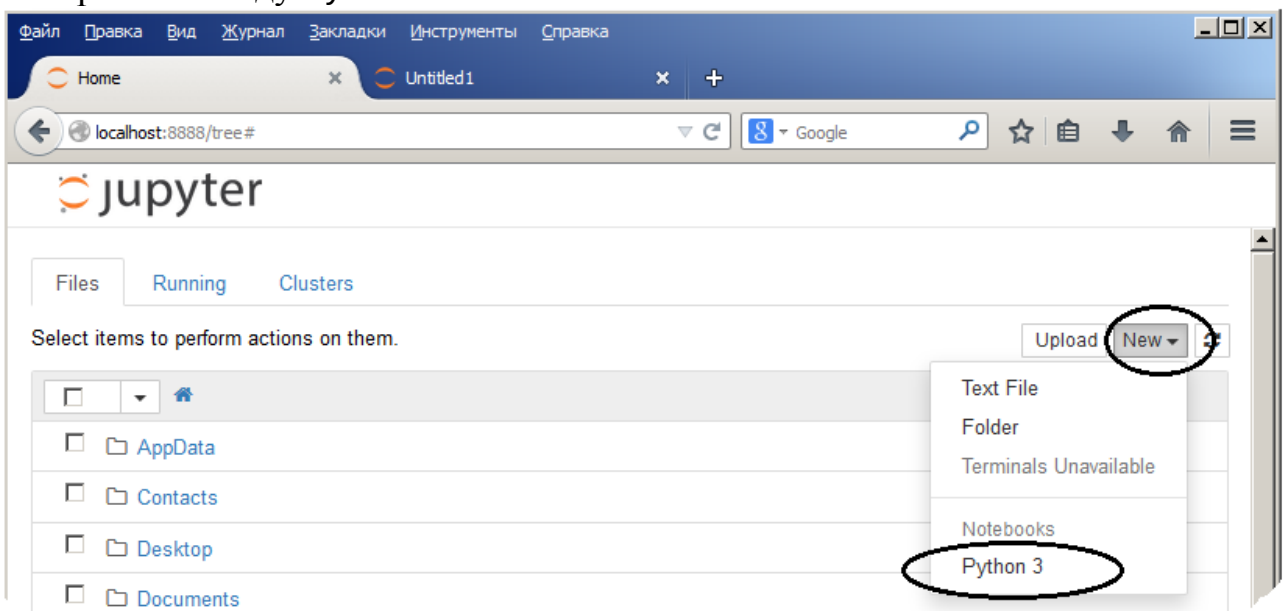
Кроме опций `'qt'` и `'inline'` у магической команды `%matplotlib` могут быть и другие опции, название которых можно узнать из справочной системы. Однако их «работоспособность» зависит от версии среды IPython и от ее настроек.

Jupyter Notebook популярная бесплатная интерактивная оболочка. Она позволяет объединить код, текст и графику в одном файле. Прежнее ее название – IPython Notebook. Его изменили, чтобы подчеркнуть совместимость

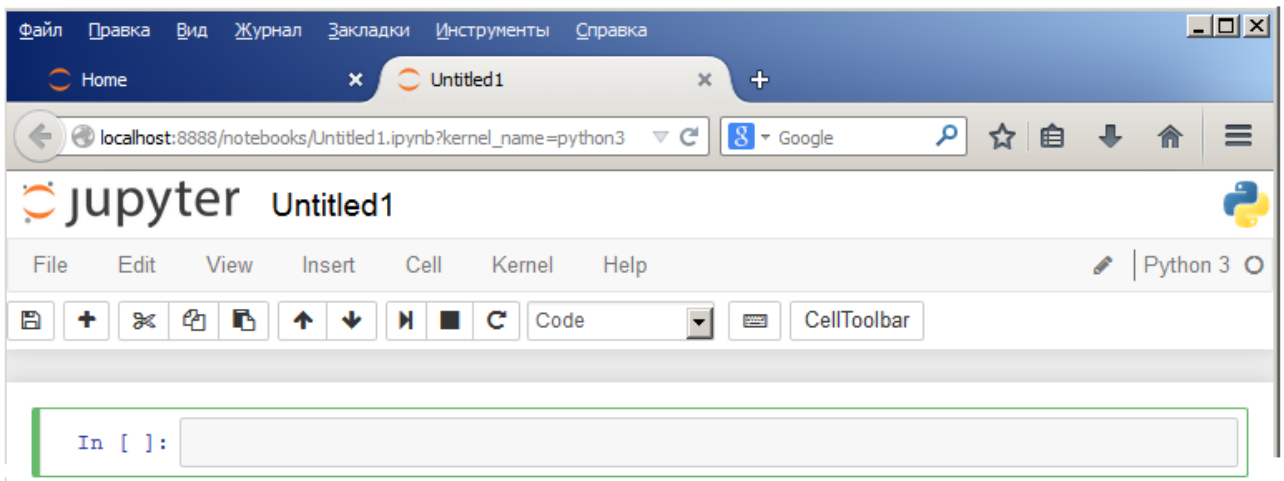
не только с Python, но и другими языками программирования. Jupyter также включен в состав пакета Anaconda.

Если вы установили пакета Anaconda, то Jupyter Notebook найдете в меню `Все программы – Anaconda3 (64-bit)-Jupyter Notebook`. При выполнении этой команды появляется окно с черным фоном и открывается браузер с закладкой, в которой находится система навигации по файлам и папкам. Черное окно – это сама программа, которая выполняет вычисления. Мы им пользоваться не будем, но и закрывать его нельзя, иначе Jupyter в браузере не будет работать. Лучше всего его свернуть.

На открывшейся закладке браузера справа вверху нажмите кнопку New и выберите команду Python 3.



В браузере открывается новая закладка с рабочим документом Jupyter.



В его верхней части расположены меню и элементы управления документом, а ниже – единственная пока секция ввода. Введите в ней `2+2` и нажмите кнопку ▶|. Вместо кнопки можно нажать комбинацию клавиш Shift – Enter. Ниже нашей секции появится поле с результатом, а рядом – метки `In[1]` и `Out[1]`, идентифицирующие поля ввода и вывода. Кроме того, появится вторая секция. В ней можно ввести новую команду, но можно отредактировать код в первой

секции и выполнить его заново. Если время выполнения кода большое, то вы успеете увидеть звездочку внутри метки `In[*]`, которая означает, что система находится в процессе вычислений.

Введите во второй секции команду

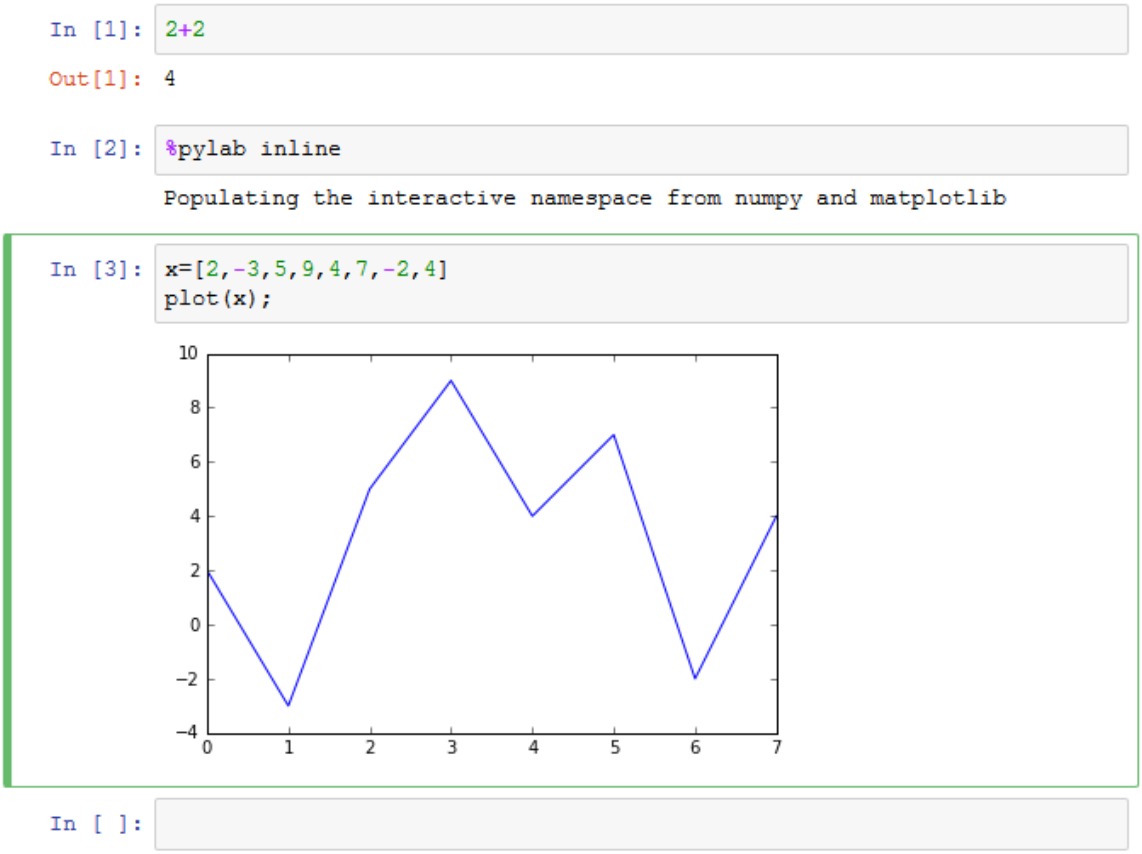
```
%pylab inline
```

Она является указанием среде Jupyter встраивать в его документы графику, генерируемую библиотекой Matplotlib.

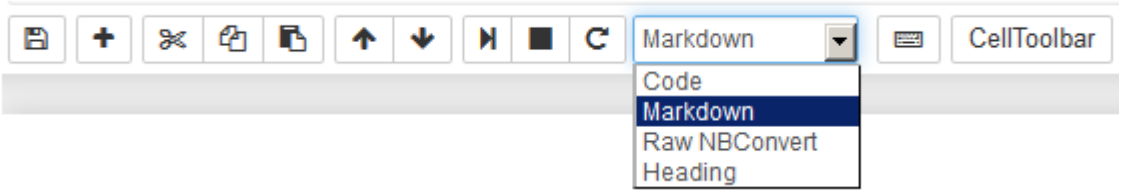
В следующей секции введите две команды (после первой строчки нажмите Enter).

```
x=[2,-3,5,9,4,7,-2,4]
plot(x);
```

Нажмите комбинацию клавиш Shift – Enter. Код выполнится и будет построен график ломаной. Результирующий вид документа приведен на следующем рисунке.

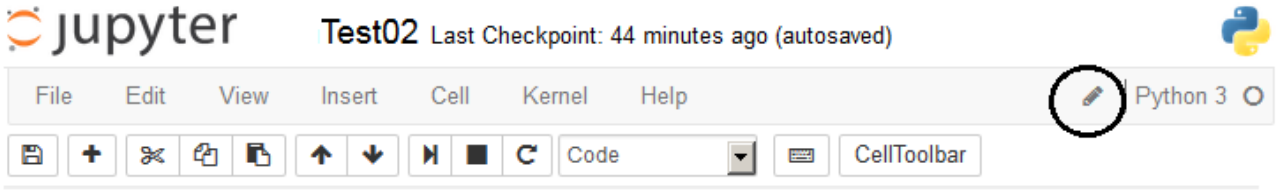


Документы Jupyter Notebook состоят из секций двух типов: ячеек с кодом и текстовых ячеек. Выше были приведены примеры секций/ячеек с кодом. Чтобы в секции вводить и форматировать обычный текст, вы должны установить курсор внутрь секции и на панели инструментов в выпадающем списке выбрать тип ячейки `Markdown`.



Markdown – это язык разметки текста. В секции такого типа с помощью простых команд–тегов (наподобие HTML) можно форматировать текст. Команда Shift – Enter переводит текстовую секцию из режима редактирования в режим отображения и обратно. Изучение языка Markdown не входит в нашу задачу, однако для простых операций форматирования этого не требуется.

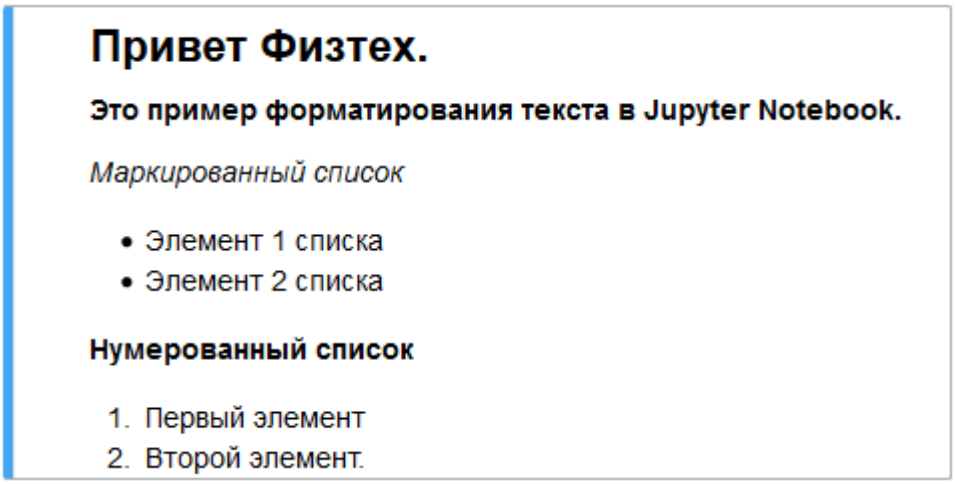
Перейдите в новую секцию документа, и выберите для нее тип Markdown. Справа вверху должен появиться символ «карандаш», который означает, что вы находитесь в режиме редактирования.



Введите в секции следующий текст

```
## Привет Физтех.
#### Это пример форматирования текста в Jupyter Notebook.
*Маркированный список*
* Элемент 1 списка
* Элемент 2 списка

**Нумерованный список**
1. Первый элемент
2. Второй элемент.
```

Здесь символы решеток, звездочек и двойных звездочек являются командами форматирования языка Markdown. Они управляют способом отображения последующего текста. Нажмите комбинацию клавиш Shift – Enter и вы получите отформатированную ячейку следующего вида.



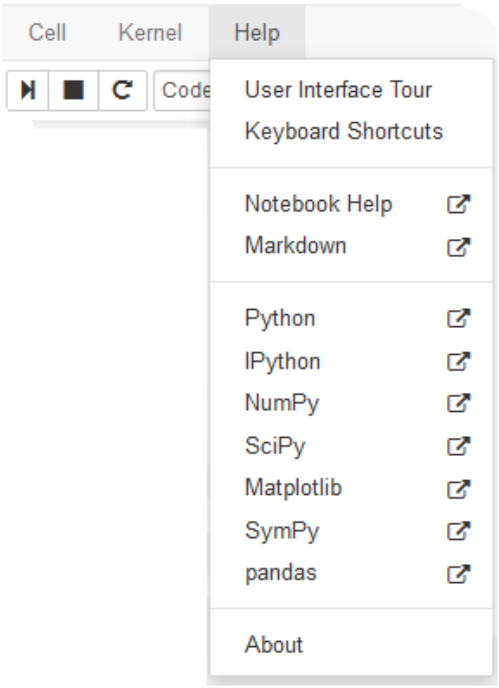
В приведенном выше коде, символы ## и #### означают представление текста, идущего следом, в виде заголовков второго (две решетки) и четвертого уровней (четыре решетки). Всего может быть 6 уровней заголовков. Между решетками и текстом должен быть пробел. Выделение и сильное выделение (полужирное начертание) текста выполняется заключением его между звездочками

Маркированный список
или двойными звездочками
Нумерованный список
Элементы маркированного списка должны начинаться символом '*' (звездочка), за которым следует пробел, знаком '-' (минус) или '+' (плюс) тоже с последующими пробелами. Элементы нумерованного списка предваряются цифрами, причем цифра в начале строки не имеет значения, а элементы нумеруются по порядку.

В «Markdown» ячейках можно выполнять и более сложное форматирование, помещать в них изображение, видео и любые другие данные, которые в состоянии отображать современный браузер.

Сделаем еще несколько замечаний. Команда Shift – Enter выполняет код секции и переводит курсор в следующую ячейку, которая открывается для редактирования. Команда Ctrl – Enter делает то же самое, но не переводит фокус на другие ячейки. Команды Esc – A и Esc – B вставляют новые ячейки выше или ниже текущей. Помимо этого, вы можете возвращаться к предыдущим ячейкам и выполнять их код. При этом используются последние (во времени) значения переменных (объектов). Если строку кода закончить символом ';' (точка с запятой), то код этой строки выполняется, но результат в поле вывода не отображается. Это, в частности, полезно для графических функций, которые кроме построения графиков, выводят служебную информацию.

Справка в Jupyter Notebook открывается из меню Help. Она содержит список быстрых клавиш (Keyboard Shortcuts), описание оболочки (Notebook Help), справочную информацию по языкам Markdown и Python, а также описание среды выполнения IPython, известной нам как Jupyter. Из этого меню также можно получить справку по множеству научных и технических библиотек Python.



Для того, чтобы получить информацию о функции можно поставить знак вопроса, набрать имя функции, и выполнить код ячейки.

```
?print
```

В текущем окне браузера ниже окна документа откроется панель с информацией об объекте. Знак вопроса можно написать и после имени функции.

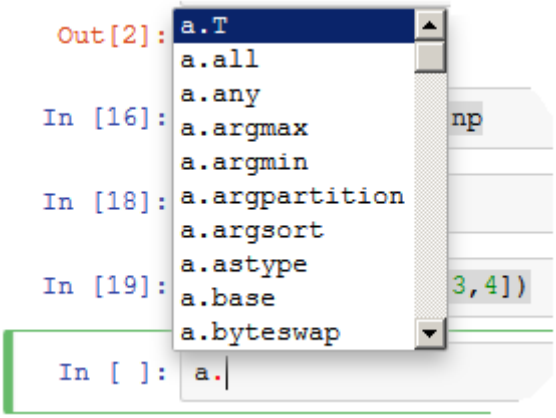
```
import numpy as np
np.linspace?
```

Это сработает для любой переменной, функции, объекта или метода. Если отображенной информации недостаточно, то используйте два вопросительных знака. Исполнительная система найдет файл с исходным кодом, и ту часть в нём, где запрашиваемый объект определен.

Для просмотра списка доступных методов объекта достаточно ввести его имя, поставить точку (и, если надо, несколько первых букв имени), и нажать клавишу TAB. Например, создадим массив **a** командой

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

В следующей секции поставьте букву **a**, точку, и нажмите TAB. Вы увидите список атрибутов и методов этого объекта (массива).



У самой программы Jupyter Notebook имеются собственные инструкции, которые начинаются со знака '%' (процент) и называются «магическими» командами. О многих из них мы говорили при описании возможностей консоли IPython. Одну из них, которая приказывает встраивать в документ графику Matplotlib, мы использовали выше:

```
%pylab inline
```

Завершение работы браузера или закрытие вкладки с Jupyter документом не останавливает работу сервера исполнительной системы. Для полной остановки следует закрыть консольное (черное) окно, ассоциированное с сервером. Оно было у нас все время свернуто или находилось на заднем плане. Завершить работу можно также из меню File текущего Jupyter документа, если выбрать в нем команду Close and Halt.

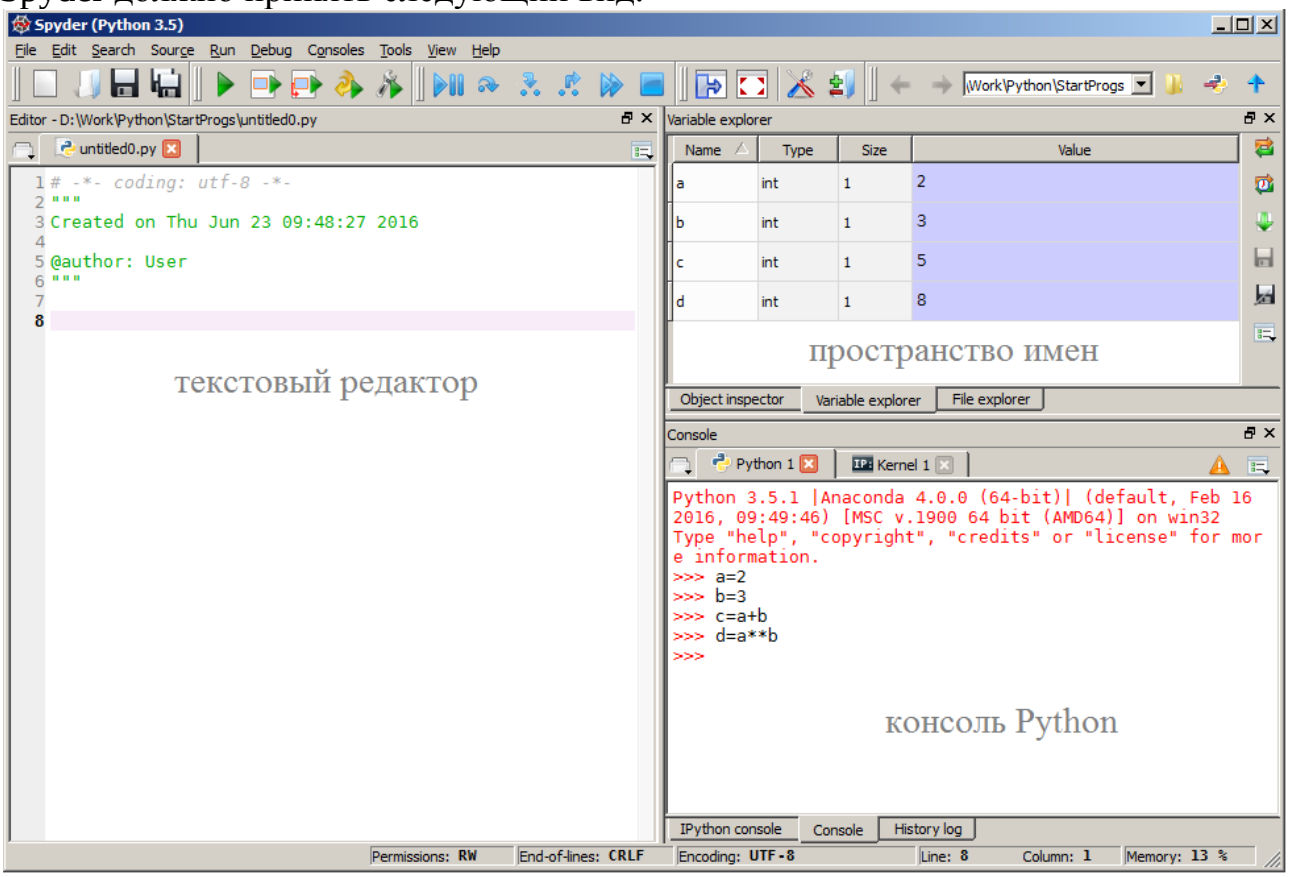
Spyder – интегрированная среда разработки и выполнения Python программ. Название Spyder расшифровывается как Scientific PYthon Development EnviRonment. Программа создавалась специально для выполнения научных расчетов и напоминает систему математических вычислений MatLab. Spyder

включен в состав дистрибутива Anaconda и содержит редактор, предназначенный для разработки Python программ, а также интерпретаторы Python Shell и IPython, любой из которых можно использовать для выполнения команд и программ.

Если вы установили пакета Anaconda, то Spyder вы найдете в меню Все программы–Anaconda3(64-bit)-Spyder. Его главное окно разделено на несколько панелей. Если вы не меняли настроек, то слева у вас откроется окно редактора кода, а в правой нижней части будут расположены одна над другой панели двух консолей: Python и IPython. Команды можно выполнять в любой из них. Пусть это будет стандартная консоль Python. Введите в ней следующие команды:

```
>>> a=2
>>> b=3
>>> c=a+b
>>> d=a**b
```

В правом верхнем углу перейдите на закладку **Variable explorer**. Окно Spyder должно принять следующий вид.



Панель **Variable explorer** отображает пространство имен в виде таблицы имен глобальных переменных, их типов, размеров и значений. При этом поле значений в этой таблице можно редактировать. Сделайте двойной щелчок мыши по полю **Value** переменной c и измените значение на 10.6. Затем в консоли выполните команду

```
>>> c
10.6
```

Вы видите, что тип и значение переменной изменились.

Все команды, введенные в консоли, выполняются в отдельном процессе, который содержит свое пространство имен, и пользователь в любой момент может процесс прервать.

Чтобы выполнить программу (файл) из консоли, следует использовать функцию `runfile('имя_файла.py')`. Например, создайте в текстовом редакторе файл со следующим кодом:

```
# файл ex001.py
print("Простая программа")
```

Сохраните его под именем `ex001.py`. Затем в консоли подайте команду

```
>>> runfile('ex001.py')
Простая программа
```

Файл, открытый в редакторе, можно выполнить с помощью кнопки  или клавиши `F5`. Однако не все файлы имеет смысл открывать в редакторе.

Чтобы получить справку по какой–либо функции (или объекту) в консоли следует выполнить команду `help(имя_функции)`. Например,

```
>>> help(list)
```

Также в этой консоли действует автозавершение команд. Оно осуществляется вводом нескольких первых букв имени функции или объекта с последующим нажатием комбинации клавиш Ctrl – пробел. Затем в выпадающем списке с помощью клавиш управления курсором (↓ и ↑) выбирается нужное имя и нажимается клавиша Tab. В командной строке появляется выбранное вами имя. Если это имя функции, то следует ввести открывающую скобку, аргументы и закрывающую скобку. В принципе, ввод и редактирование команд в этой консоли выполняется аналогично Python Shell, и мы на этом останавливаться не будем.

Теперь активизируйте вкладку **`IPython console`**. Эта консоль содержит собственное пространство имен. Поэтому панель **`Variable explorer`** очистится (она показывает пространство имен активного интерпретатора). У этой консоли функциональные возможности такие же, как у Jupyter QtConsole. Ввод, редактирование и выполнение команд выполняется точно так же. В ней доступны все магические команды IPython, и имеется такая же возможность отображать в документе графику.

Обратимся к текстовому редактору. Он используется для ввода текстов программ и умеет по–разному подсвечивать код. Как сказано выше, для выполнения программы (всего кода, набранного в редакторе) следует использовать кнопку  или клавишу `F5`.

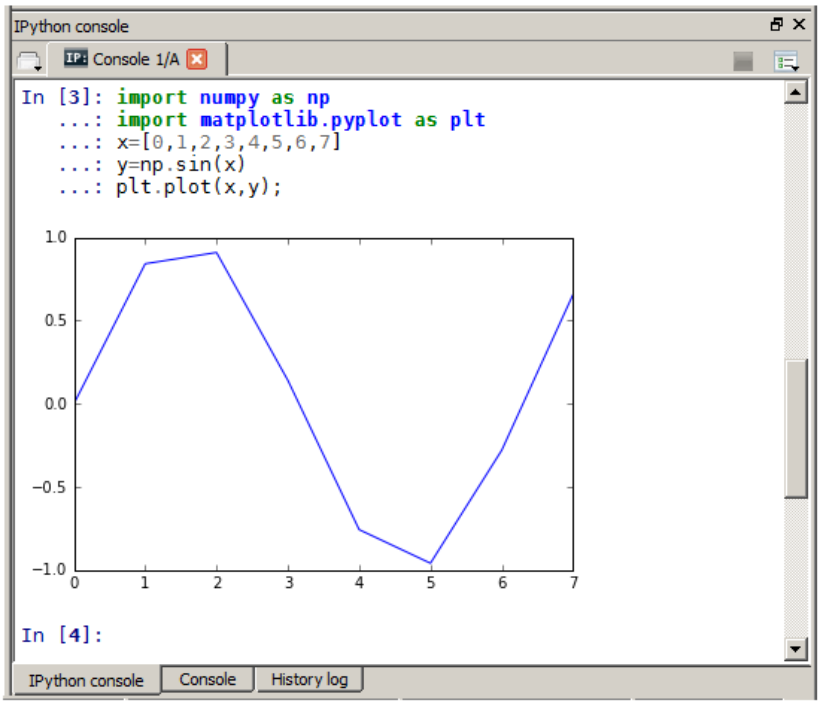
Кроме того, редактор обладает еще одной особенностью – можно выполнять только часть кода (мы не упоминали, но в редакторе IDLE также реализована эта возможность). Вернитесь в текстовый редактор и в файле `ex001.py` введите еще одну строку: `print("Привет универу!")`. Выделите только текст этой строки и нажмите клавишу `F9`. В активной консоли (в нашем случае в IPython) в строке ввода отобразится и будет выполнена выделенная команда.

```
In[1]: print("Привет универу!")
Привет универу!
```

Аналогично можно выполнять целые блоки кода. Введите в том же файле следующий код:

```
import numpy as np
import matplotlib.pyplot as plt
x=[0,1,2,3,4,5,6,7]
y=np.sin(x)
plt.plot(x,y);
```

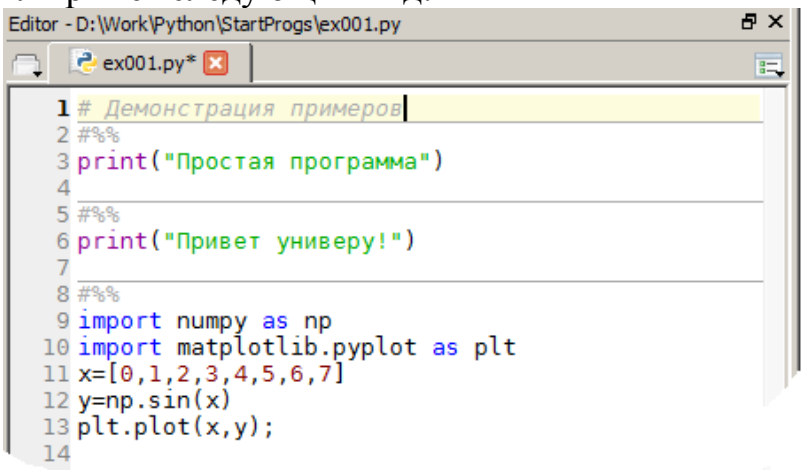
Выделите эту часть кода и нажмите клавишу F9. Код выполнится, и панель IPython примет вид



Для экономии места мы привели изображение только одной панели, а не всего окна Spyder. Таким образом, в текстовом редакторе можно любой код вводить по частям, и частями же выполнять его. Для разделения кода на части в редакторе можно использовать команду

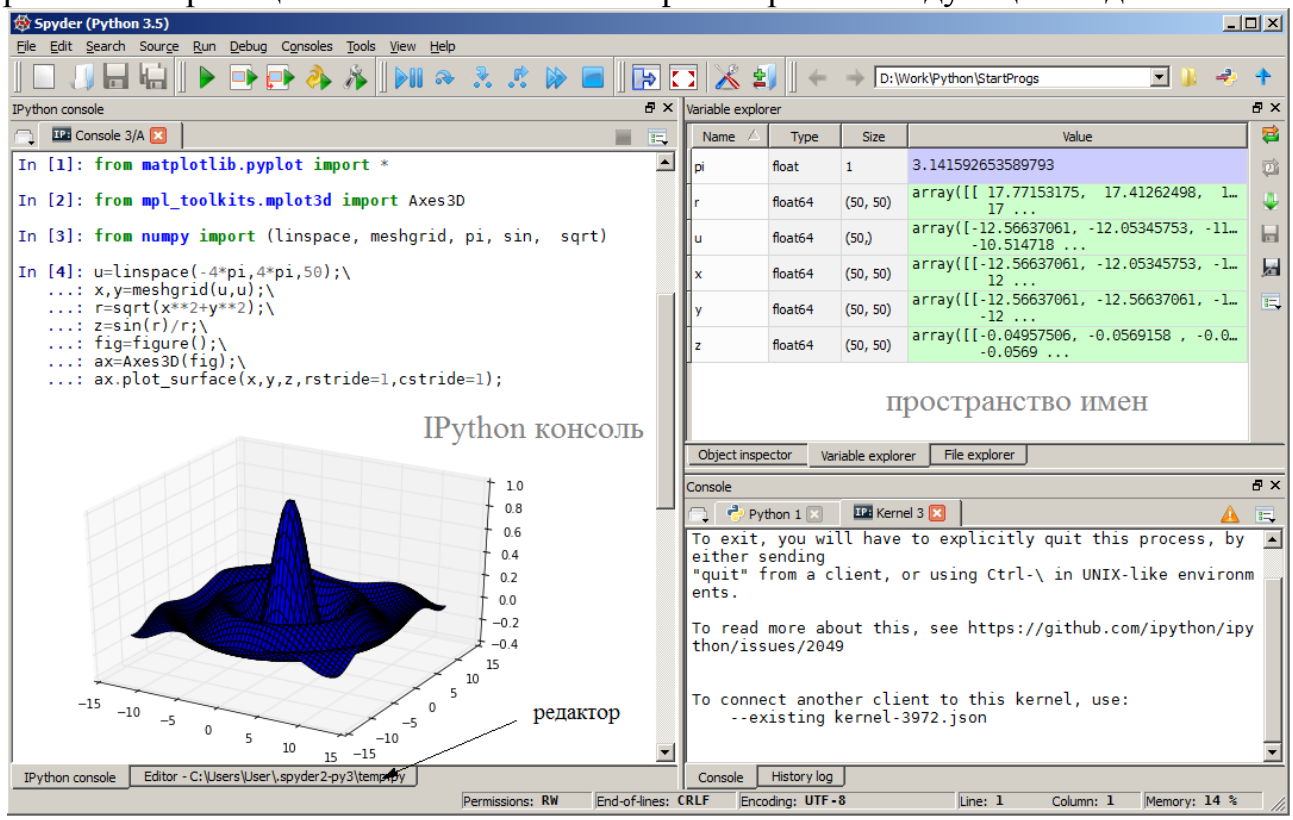
```
#%%
```

Например, если в тексте файла ex001.py несколько раз вставить такую команду, то файл примет следующий вид:



Однако команда '#%%' создает только видимое разбиение на блоки, проводя только разделительные линии. Никакого реального разбиения файла на ячейки не происходит.

Можно настроить среду Spyder для работы преимущественно с IPython консолью. Для этого захватите мышью заголовок панели IPython и перетащите его, расположив консоль IPython поверх редактора. Затем, если нужно, растяните границы окна консоли. Окно Spider примет следующий вид.



Здесь в панели IPython мы привели пример построения графика функции двух переменных. Ниже приведен его код.

```
In[1]: from matplotlib.pyplot import *
In[2]: from mpl_toolkits.mplot3d import Axes3D
In[3]: from numpy import (linspace, meshgrid, pi, sin,  sqrt)
In[4]: u=linspace(-4*pi,4*pi,50);\
       x,y=meshgrid(u,u);\
       r=sqrt(x**2+y**2);\
       z=sin(r)/r;\
       fig=figure();\
       ax=Axes3D(fig);\
       ax.plot_surface(x,y,z,rstride=1,cstride=1);
```

В конце каждой строки стоит символ '\' (обратный слэш), который является признаком продолжения строки. Точки, которые видны на рисунке и стоят слева в начале каждой строки, появляются автоматически после нажатия клавиши Enter. Можно обойтись без символа продолжения '\'. Если в IPython Spyder вы желаете набрать несколько однострочных команд в одной секции, то после первой строки нужно нажать комбинацию клавиш Ctrl – Enter. Последующие строки следует завершать нажатием клавиши Enter. Для

завершения ввода всей многострочной инструкции и выполнения кода секции следует два раза нажать Enter.

Ввод многострочной команды в IPython консоли выполняется так же, как обычно, но отступы нужно делать от многоточий,

```
In[8]:for n in range(1,5):
   ...:    print(n, end=" ")
   ...:
```

и аналогично, дважды нажимать Enter для завершения команды.

```
1 2 3 4
```

Чтобы перетащить IPython консоль обратно, надо захватить мышью верхнюю строку ее панели, в которой написано IPython console, и переместить, например, на прежнее место.

Надеемся, что с другими панелями Spyder вы разберетесь самостоятельно, когда лучше познакомитесь с языком программирования Python.

2. Основные языковые конструкции Python

В этой главе мы продолжим изучения языка программирования Python. При этом будем выполнять примеры в интерпретаторе Python Shell или в консоли Spyder. На это будут указывать символы приглашения '>>>', которые мы будем использовать при записи инструкций. Однако вы уже познакомились с другими интегрированными средами разработки, и примеры можете выполнять в любой из них.

2.1 Типы данных и операторы управления

Типы данных. Описания типов переменных в Python нет. Это означает, что при присваивании переменной значения интерпретатор автоматически относит переменную к одному из типов данных. Вот перечень основных типов:

- `bool` – логический тип данных;
- `int` – целые числа;
- `float` – вещественные числа;
- `complex` – комплексные числа;
- `str` – unicode–строки;
- `list` – списки;
- `tuple` – кортежи;
- `dict` – словари;
- `function` – функции;

Есть и другие, реже используемые, типы. Для определения типа переменной предназначена функция `type()`.

```
>>> a=45
>>> type(a)
<class 'int'>
```

```
>>> L = [1, 3, 5]
>>> type(L)
<class 'list'>
```
Имена типов по совместительству являются функциями, преобразующими в этот тип объекты других типов (если преобразование имеет смысл).
```
>>>float(2)
2.0
>>>int(-2.9)      # отбрасывание дробной части
-2
>>> bool(23)     # любое число, отличное от нуля, преобразуется в True
True
>>> bool(0)      # ноль преобразуется в False
False
>>> x="321"      # преобразование строки в целое
>>> y=int(x); y
321
>>> y+100
421
>>> z='34.123'
>>> w=float(z); type(w)     # преобразование строки в вещественное число
<class 'float'>
```

Операторы управления вычислительным процессом. Обычно программа состоит из текстового файла, имеющего расширение имени `*.py` (или `*.pyw` для программ с оконным интерфейсом), который можно создавать в любом текстовом редакторе. В Python Shell вы можете использовать встроенный редактор, который вызывается из его меню командой File – New File. В Spyder тоже имеется свой редактор, окно которого обычно занимает его левую половину. Независимо от того, какой редактор используется, при написании кода программ следует придерживаться определенных правил.

Команды программы вводятся последовательно, строка за строкой. Строка должна иметь вид такой, как если бы она была записана в командной строке интерпретатора. В текстовом файле программы все операторы одного уровня должны иметь один и тот же отступ или не иметь его вовсе. Вложенные операторы (тела функций, условных операторов, операторов цикла) должны иметь одинаковый отступ, если они не являются вложенными во вложенные операторы. Особенность языка Python состоит в том, что в нем отсутствуют ограничительные символы для выделения инструкций внутри блока. Вложенность определяется величиной отступа. Например, в следующем примере тело цикла состоит из двух операторов и они должны иметь одинаковый отступ (обычно длину табуляции).
```
i=1
while i<5:
   print(i,end=" ")
   i+=1
1 2 3 4
```

Инструкции, перед которыми расположено одинаковое количество пробелов, являются телом блока (в данном случае телом цикла). Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В языке Python принято использовать четыре пробела для выделения инструкций внутри блока.

Приведенную программу вы можете ввести и выполнить в командном окне интерпретатора, если будете руководствоваться правилом, что после ввода двоеточия и нажатия клавиши Enter, следующую строку следует набирать с отступом, а концом ввода блока (концом ввода многострочной команды) является двойное нажатие клавиши Enter.

Инструкции программы выполняются по порядку, начиная с первой. Порядок выполнения может быть изменен с помощью специальных операторов цикла или условного выбора. Оператор цикла позволяет выполнить одни и те же инструкции, которые называются телом цикла, многократно. В языке Python используются два цикла: `for` и `while`. Цикл `for` применяется для перебора элементов последовательности и имеет следующий формат:

```
for Элемент in Последовательность:
    тело цикла
[else:
    необязательный блок, выполняемый, если
    не использовался оператор break
]
```

Под термином «последовательность» здесь понимаются такие объекты как кортежи, списки, строки, словари и другие, для которых определены средства последовательного перебора их элементов.

Простейший вид цикла – это цикл по элементам списка.

```
>>> L=[10,20,30,40,50]
>>> for x in L:
    print(x**2,end=" ")
100 400 900 1600 2500
```

Очень часто используются циклы по диапазонам целых чисел.

```
for i in range(5):
   print(i,end=" ")
0 1 2 3 4
```

В Python 3 функция `range()` создает специфический объект, поддерживающий перебор его элементов в цикле. Элементы этого объекта всегда целые числа. Объект `range` позволяет получать значения элементов по индексу, отсчитываемому от 0. Функция `range()` имеет следующий формат:

```
range([start,] stop[,step])
```

Если параметр `start` не указан, то по умолчанию используется 0. Если параметр `step` не указан, то используется значение 1. Элемент с индексом `start` включается в последовательность, а элемент с индексом `stop` – нет.

```
>>> r=range(5)  # содержит элементы: 0,1,2,3,4
>>> r[3]
3
```

Чтобы из такого объекта получить список, его следует передать аргументом в функцию list().

```
>>> m=range(3,8)
>>> list(m)
[3, 4, 5, 6, 7]
```

Цикл while применяется в тех случаях, когда заранее неизвестна последовательность. Он имеет следующий формат:

```
while условие:
        тело цикла
        изменение переменных, используемых в условии
[else:
        необязательный блок, выполняемый, если
        не использовался оператор break
]
```

Тело цикла while выполняется до тех пор, пока истинно условие (логическое выражение), указанное в его начале. Перед началом цикла while переменным, используемым в логическом выражении, должны быть присвоены какие–либо начальные значения.

```
>>> L=[1,3,5,2,-8,4]
>>> i=0
>>> while L[i]>0:       # пока элементы списка положительны
        print(L[i]**3,end=" ")
        i+=1
1 27 125 8
while L:                # пока список не пуст
   print(L[0],end=" ")
   L=L[1:]              # удаление из списка первого элемента
1 3 5 2 -8 4
L
[ ]
>>> L=[1,3,5,2,-8,4]; i=5
>>> while i:            # печать списка в обратном порядке
        print(L[i],end=" ")
        i-=1
4 -8 2 5 3
```

Условие в последнем операторе while не содержит сравнения. На каждой итерации цикла из значения переменной (счетчика) i вычитается единица. Как только значение i станет равным нулю, цикл остановится, поскольку число 0 в логическом выражении эквивалентно значению False.

Если тело цикла состоит из одной инструкции, то допустимо поместить ее на одной строке с основной инструкцией.

```
for i in range(1,11): print(i)
```

Если инструкция является слишком длинной, то ее можно перенести на следующую строку несколькими способами. Можно в конце строки написать

символ \ (слэш), после которого должен следовать символ перевода строки. Другие символы (в том числе и комментарии) недопустимы.

```
>>> x=78+34\
  +25
>>> x
137
```

Точно также эти команды вы можете набрать в текстовом редакторе (естественно без символов приглашения >>>).

Для продолжения инструкции на следующую строку можно поместить выражение внутри круглых скобок.

```
>>> x=(15+25
      +35)
>>> print(x)
75
```

Несколько позже мы поясним понятие кортежа, списка и словаря, определение которых помещается внутри круглых, квадратных и фигурных скобок. При задании этих объектов такие скобки можно использовать аналогично круглым, и определение кортежа, списка и словаря можно размещать на нескольких строках.

В некоторых примерах этого параграфа (и последующих) мы не будем приводить символы приглашения >>>. Все примеры вы можете набирать в текстовом редакторе и выполнять файл примера в интерпретаторе целиком, либо можете каждую команду набирать в командной строке интерпретатора и выполнять их последовательно. Для коротких примеров это не существенно, однако примеры, содержащие более 10 строк кода, мы рекомендуем набирать в виде программ и выполнять как единое целое.

Рассмотрим одну особенность работы циклов `for`. Попробуем написать цикл, который меняет элементы списка, например, возводит каждый элемент списка в квадрат.

```
>>> lst=[1,2,3,4]
>>> for elem in lst:
      elem=elem**2
>>> print(lst)
[1, 2, 3, 4]
```

Список `lst` не изменился. Дело в том, что переменная `elem` на каждом шаге цикла содержит лишь копию значения текущего элемента списка. Изменить таким способом элементы списка нельзя. Чтобы изменить элементы списка, цикл должен выполняться по какому–либо другому списку, например, по списку индексов.

```
>>> for i in range(len(lst)):
      lst[i]=lst[i]**2
>>> print(lst)
[1, 4, 9, 16]
```

Для работы с циклом for полезна функция enumerate(список). Она на каждой итерации возвращает пару объектов: значение индекса i и значение соответствующего элемента списка.

```
>>>for i,x in enumerate(lst):
      print(i+1,"  ",x)
1     1
2     4
3     9
4     16
```

Оператор ветвления if позволяет в зависимости от значения логического выражения выполнить один или другой участок кода. Он имеет следующий формат:

```
if логическое_выражение 1:
      блок операторов; выполняется, если логическое_выражение1  истинно
[elif логическое_выражение 2:
      блок операторов; выполняется, если логическое_выражение 2  истинно
]
[else:
      блок операторов; выполняется, если предыдущие
      логические выражения ложны
]
```

Участков кода elif может быть несколько.

```
>>> x=25
>>> if x%2==0:
      print(x," - четное число")
else:
      print(x," - нечетное число")
25  - нечетное число
```

Если блок операторов состоит из одной команды, то ее можно разместить на одной строке с заголовком оператора if.

```
>>> x=34
>>> if x%2==0: print(x," - четное число")
else: print(x," - нечетное число")
34  - четное число
```

Внутри циклов часто используются операторы continue и break. Оператор continue позволяет перейти к следующей итерации цикла до завершения выполнения всех команд тела цикла

```
>>> for i in range(1,10):
      if 4<i<8: continue
      print(i,end=" ")
1 2 3 4 8 9
```

Оператор break прерывает цикла досрочно.

```
>>> i=1
>>> while True:
      if i>5: break
```

```
    print(i,end=" ")
    i+=1
1 2 3 4 5
```

Операторы сравнения используются в логических выражениях. Перечислим эти операторы: == (равно), != (не равно), <,<=,>,>= (меньше, меньше или равно, больше, больше или равно), in (проверка того, является ли элемент членом последовательности), is (проверка того, ссылаются ли две переменные на один и тот же объект). При этом в логическом выражении можно указывать сразу несколько условий:

```
>>> x=5
>>> 1<x<9
True
```

Несколько логических выражений можно объединить в одно с помощью операторов and (логическое И) и or (логическое ИЛИ).

```
>>> x1=5; x2=3
>>> x1==x2 or x1!=x2
True
```

Кроме того, значение логического выражения можно инвертировать с помощью оператора not.

```
>>> x=True
>>> y=not x
>>> y
False
```

2.2 Списки, кортежи, словари и строки

В этом параграфе мы будем говорить о типах Python, которые могут хранить последовательности данных: о списках, кортежах и строках. Есть еще типы bytes и bytearray, которые также могут хранить последовательности данных, но о них мы поговорим в другом месте. Используя термин последовательность, мы будем иметь в виду любой из этих типов.

Любой набор чисел или других объектов, заключенный в квадратные скобки, интерпретируется как список, и над ним можно выполнять преобразования естественные для такого типа данных. Списки могут содержать объекты любых типов и в одном списке могут быть объекты разных типов. Списки являются изменяемыми объектами, т.е. элементы списка можно изменить с помощью операции присваивания.

```
>>> L=[1,2,3]
>>> L[1]=11
>>> L
[1, 11, 3]
```

Кортеж (tuple) – это нумерованный набор объектов, заключенный в круглые скобки, элементы которого нельзя менять. Кортеж, по сути – это неизменяемый список. Можно получить элемент кортежа по индексу, но изменить его нельзя:

```
>>> K=(4,5,6)
>>> print(K[1])
```

```
5
>>> K[1]=11
Ошибка!
```
Если кортеж единственный объект в правой части присваивания, то скобки ставить не обязательно.
```
>>> t=1,2,3
>>> t
(1, 2, 3)
```
Работать с кортежами можно так же, как со списками. Нельзя только изменять их.

Многие функции, в частности функции модуля `math` не могут принимать в качестве аргумента списки и кортежи. Чтобы применить какую–либо функцию `f` ко всем элементам списка (кортежа) и получить список результатов, следует задействовать встроенную функцию `map(f,список)`, где `f` это имя функции (без аргументов). В результате возвращается «итератор», объект, который можно преобразовать в список с помощью функции `list()`.
```
>>> import math as m
>>> list(map(m.sin,[0,m.pi/6,m.pi/4, m.pi/2]))
[0.0,0.49999999999999994, 0.7071067811865475, 1.0]
>>> def f(x): return x**2
>>> list(map(f, [0, 1, 2, 3, 4]))
[0, 1, 4, 9, 16]
```
Списки и кортежи являются упорядоченными последовательностями элементов и поддерживают обращение к элементам по индексу. Нумерация элементов начинается с 0. В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца последовательности, или точнее, индекс будет определяться как разность длины и индекса (с учетом знака).
```
>>> L
[1, 11, 3]
>>> L[-1]          # последний элемент
3
>>> L[len(L)-1] # последний элемент
3
```
Здесь функция `len()` вычисляет длину (количество элементов) списка.

Перебрать элементы списка можно с помощью цикла `for`.
```
>>> L=[10,20,30,40,50,60]
>>> for elem in L:
        print(elem,end=" ")
10 20 30 40 50 60
```
К спискам применимы операции конкатенации (оператор +), повторения (оператор *), проверки на вхождение (оператор `in`).
```
>>> L2=[4,5,6]
>>> L+L2          # конкатенация (соединение) списков
[1, 11, 3, 4, 5, 6]
```

```
>>> L*3          # повторение списка L три раза
[1, 11, 3, 1, 11, 3, 1, 11, 3]
>>> 11 in L      # проверка того, что число 11 имеется в списке
True
>>> 12 in L
False
>>> K2=(7,8,9)
>>> K+K2         # конкатенация кортежей
(4, 5, 6, 7, 8, 9)
>>> K*2          # повторение кортежа K два раза
(4, 5, 6, 4, 5, 6)
>>> 5 in K       # проверка того, что число 5 содержится в кортеже
True
```

Допустимы операции составного присваивания типа `список+=список` или `список*=целое_число`.

```
>>> L=[10,20,30]
>>> L+=[1,2,3];L
[10, 20, 30, 1, 2, 3]
>>> L=[10,20,30,40]
>>> L*=2; L
[10, 20, 30, 40, 10, 20, 30, 40]
```

Перед использованием списка или кортежа его нужно создать. Самый простой способ – это перечислить все элементы списка внутри квадратных скобок.

```
>>> lst=['one',2, "three",4]
>>> lst
['one', 2, 'three', 4]
```

Функция `list()` позволяет преобразовать любую последовательность, указанную в ее аргументе, в список. Если параметр не указан, то создается пустой список.

```
>>> list("Hello")                  # преобразуем строку в список
['H', 'e', 'l', 'l', 'o']
>>> list()                         # создаем пустой список
[]
>>> list((10,11,12,13,14,15))      # преобразуем кортеж в список
[10, 11, 12, 13, 14, 15]
```

У списков есть методы, которые позволяют их модифицировать. Метод `append()` добавляет элемент в конец списка.

```
>>> L=[10,20,30,40]; L.append(100); L
[10, 20, 30, 40, 100]
```

Используя этот метод в цикле, можно строить список поэлементно

```
>>> L=[ ]
>>> for i in range(10):
        L.append(i**2)
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Но более элегантно такой список создается с помощью генератора списка (list comprehension).

```
>>> [i**2 for i in range(10)]       # квадратные скобки обязательны
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Здесь цикл `for` находится внутри квадратных скобок. Другой особенностью является то, что инструкция, выполняемая внутри цикла, находится перед циклом.

Генераторы списков могут иметь сложную структуру, например, состоять из нескольких вложенных циклов `for` или содержать условный оператор `if`.

```
>>> [ [ a, b ]  for a in range(3) for b in range(2)]
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
```

В генерируемый список можно включать не все элементы

```
>>> [ elem**2  for elem in range(10) if elem != 5 ]
[0, 1, 4, 9, 16, 36, 49, 64, 81]
```

При присваивании одного списка другому создается новая ссылка на тот же список, а не его копия.

```
>>> L=[10,20,30,40,50,60]
>>> M=L; M[2]=33; M
[10, 20, 33, 40, 50, 60]
>>> L
[10, 20, 33, 40, 50, 60]
```

В следующей команде операция `is` проверяет, являются ли `M` и `L` одним и тем же объектом (разными именами, которые ссылаются на одну и ту же область памяти).

```
>>> M is L
True
```

Чтобы изменять списки M и L независимо, нужно присвоить переменной M не список L, а его копию. Тогда это будут два различных списка, содержащие в начальный момент одни и те же элементы. Для этого используется команда M=L[:], где L[:] – это подсписок списка L от начала до конца, а подсписок всегда копируется.

```
>>> L=[10,20,30,40,50,60]; M=L[:]; M
[10, 20, 30, 40, 50, 60]
>>> M is L
False
```

Операция '==' может проверить, совпадают ли списки поэлементно.

```
>>> M==L
True
>>> M[2]=33
>>> M==L
False
```

Другой способ создания копии списка состоит в использовании функции `list()`.

```
>>> L=[10,20,30,40,50,60]
>>> M=list(L)
```

```
>>> M is L        # результат  False  показывает, что L и M разные объекты
False
```
Однако следует помнить, что функция `list()` и операция среза (: двоеточие) создают лишь поверхностную копию, т.е. копию внешнего одномерного списка. Если списки вложенные, то для создания полной копии следует воспользоваться функцией `deepcopy()` из модуля `copy`.
```
>>> import copy
>>> L=[[1,2,3],[4,5,6]]
>>> M=copy.deepcopy(L)  # или M=list(L)
>>> M[1][1]=555
>>> M
[[1, 2, 3], [4, 555, 6]]
>>> L       # использование list() приводит к изменению элемента L[1][1]
[[1, 2, 3], [4, 5, 6]]
```
Функция `deepcopy()` создает полную копию любого объекта Python, а не только списка.

Последовательности (списки, строки, кортежи и т.д.), кроме обращения к элементам по индексу, поддерживают операцию получения среза, которая возвращает фрагмент последовательности. Операция имеет следующий формат:

`объект[начало : конец : шаг]`

Все параметры являются необязательными. Если параметр `Начало` не указан, то используется значение 0. Если параметр `Конец` не указан, то возвращается фрагмент до конца последовательности. Если параметр `Шаг` не указан, то используется значение 1. Значения параметров могут быть отрицательными.
```
>>> L=[10,20,30,40,50,60,70,80,90]
>>> L[2:5]
[30, 40, 50]
>>> L[1:9:2]
[20, 40, 60, 80]
```
Заметим, что символ с индексом, указанным в параметре `Конец`, не входит в возвращаемый фрагмент. Фактически, в параметрах `начало` и `конец` указываются не номера элементов, а номера «промежутков» между ними, где нулевым промежутком считается позиция перед первым элементом последовательности.

Вот еще несколько примеров использования операции среза.
```
>>> L[ : ]  # Возвращается фрагмент от позиции 0 до конца списка
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```
Здесь начало не указано (значит 0), конец не указан (значит до конца), шаг не указан (значит 1), т.е. `L[:]` возвращает список целиком, а точнее, возвращает копию списка (можно использовать запись `L[::]`).
```
>>> L=[10,20,30,40,50,60]; M=L[:]; M[2]=33; M
[10, 20, 33, 40, 50, 60]
>>> L                    # список L не изменился
[10, 20, 30, 40, 50, 60]
```

```
>>> L[ : : -1]          # список в обратном порядке
[60, 50, 40, 30, 20, 10]
>>> M=[11]+L[1:]       # M отлично от L значением первого элемента списка
>>> M
[11, 20, 30, 40, 50, 60]
>>> M=L[:-1]; M        # удаление последнего элемента списка
[10, 20, 30, 40, 50]
>>> M=L[-1:]; M        # получить последний элемент списка, а точнее
                       # получить фрагмент от len(L)-1 до конца списка
[60]
>>> L[:4]
[10, 20, 30, 40]
>>> L[4:]
[50, 60]
>>> L[:4]+L[4:]
[10, 20, 30, 40, 50, 60]
```

Можно заменить какой-нибудь подсписок новым списком (в том числе другой длины)

```
>>> L[2:4]=['a','b','c'];L      # заменить второй и третий элементы
[10, 20, 'a', 'b', 'c', 50, 60]
>>> L[2:4]=[ ]; L               # удалить второй и третий элементы
[10, 20, 'c', 50, 60]
>>> L[2:3]=[1,2,3];L            # заменить второй элемент элементами из списка
[10, 20, 1, 2, 3, 50, 60]
```

Некоторые из этих операций могут быть записаны в другой форме.

```
>>> L=[1,2,3,4,5,6,7,8]
>>> del L[2:4]; L
[1, 2, 5, 6, 7, 8]
```

Элементами списка могут быть другие списки

```
>>> L=[[1,2,3],[4,5,6]]
>>> L[1][1]
5
```

Поскольку выражение в скобках может располагаться на нескольких строках, то последнее присваивание можно записать так:

```
>>> L=[[1,2,3],
       [4,5,6]]
```

Количество вложений списков не ограничено.

Вложенные списки можно создавать с помощью генераторов списков, а затем, если нужно, менять или добавлять элементы.

```
>>> lst=[ [ ]  for a in range(3)]
>>> lst
[[], [], []]
>>> lst[0].append(100)
>>> lst
[[100], [], []]
>>> lst[1].append(222)
```

```
>>> lst
[[100], [222], []]
```
Как мы видели, переменная может ссылаться на список. Но и слева от операции присваивания может стоять список переменных.
```
>>> [x,y,z]=[[1,2],[3,4],[5,6]]
>>> x
[1, 2]
>>> y
[3, 4]
>>> z
[5, 6]
```
Вообще, по обе стороны оператора присваивания могут стоять последовательности, содержащие одинаковое количество элементов.
```
>>> x,y="12"
>>> x
'1'
>>> y
'2'
>>> [x,y,z]=["Hello",[1,2,3],[[1,2],[3,4]]]
>>> x
'Hello'
>>> y
[1, 2, 3]
>>> z
[[1, 2], [3, 4]]
```
Существует возможность сохранения в элементе левого списка лишних элементов правого, если количество элементов справа от знака присваивания больше количества элементов левого списка. Для этого перед именем переменной левого списка указывается звездочка(*).
```
>>> [x,y,*z]=[1,2,3,4]
>>> print(x,y,z)
1 2 [3, 4]
>>> [x,*y,z]=["Hello",[1,2,3],123,[[1,2],[3,4]]]
>>> x
'Hello'
>>> y
[[1, 2, 3], 123]
>>> z
[[1, 2], [3, 4]]
```
Переменная, перед которой указана звездочка, всегда содержит список. Если для этой переменной не хватило значений, то ей присваивается пустой список.
```
>>> [x, y, *z]= [10, 20]
>>> z
[]
```
Звездочку можно указать только перед одной переменной, иначе возникает неоднозначность, и интерпретатор выводит сообщение об ошибке.

Списки являются объектами и, следовательно, они имеют методы.

```
>>> L=[1,2,3,4,5,6,7,8]
>>> L.insert(3,0); L          # вставить на 3-ю позицию ноль
[1, 2, 3, 0, 4, 5, 6, 7, 8]
>>> L.append(10); L           # добавить число 10 в конец списка
[1, 2, 3, 0, 4, 5, 6, 7, 8, 10]
```

Метод `extend(Последовательность)` добавляет в конец списка элементы последовательности (другого списка, кортежа и т.д.). Метод изменяет текущий список и ничего не возвращает.

```
>>> L.extend([12,13,14]); L
[1, 2, 3, 0, 4, 5, 6, 7, 8, 10, 12, 13, 14]
```

Метод `remove(x)` удаляет первый элемент в списке, имеющий значение `x`.

```
>>> L=[1,7,3,4,5,7,5,3]
>>> L.remove(7); L
[1, 3, 4, 5, 7, 5, 3]
```

Метод `pop(i)` удаляет `i`-ый элемент из списка и возвращает его. Если индекс не указан, удаляется последний элемент

```
>>> x=L.pop()
>>> L
[1, 3, 4, 5, 7, 5]
>>> x
3
>>> L.pop(2)
4
>>> L
[1, 3, 5, 7, 5]
```

Метод `index(x,[start[,end]])` возвращает положение первого элемента `x` в диапазоне от `start` до `end`.

```
>>> L.index(5)
2
```

Метод `count` возвращает количество элементов списка со значением `x`.

```
>>> L.count(5)
2
```

Есть еще метод `sort()` – сортировка списка, `reverse()` – обращение порядка элементов в списке, `clear()` – очистка списка, `copy()` – поверхностная копия списка. Заметим, что многие описанные методы изменяют сам список.

Функции `min(список)` и `max(список)` возвращают наименьший и наибольший элементы.

Функция `zip(...)` принимает набор последовательностей (списков, кортежей и т.д.) и возвращает итератор по кортежам, где i–й кортеж содержит i–й элемент каждой из последовательностей.

```
>>> a = [1,2]
>>> b = [3,4]
>>> list(zip(a,b))            # преобразуем итератор в список
[(1, 3), (2, 4)]
```

Здесь функция `zip()` соединяет списки парами. Итератор завершается, когда исчерпана кратчайшая из последовательностей.

```
>>> list(zip([1, 2, 4], [4, 5], [5, 7]))
[(1, 4, 5), (2, 5, 7)]
>>> c = [5,6]
>>> tuple(zip(a,b,c))        # преобразуем итератор в кортеж
((1, 3, 5), (2, 4, 6))
```

Мы уже упоминали функцию `map(fun,list)`. Она применяет функцию `fun` к элементам списка `list` и возвращает итератор, который можно преобразовать в список.

```
>>> def f(x):
      return x*x

>>> list(map(f, [0.5,0.9,1.9]))
[0.25, 0.81, 3.61]
```

В функцию `map(fun,...)` можно передать несколько списков. Тогда функция `fun` должна принимать столько аргументов, сколько передано списков в функцию `map`.

```
>>> def f(x, y):
      return x*y

>>> a = [1,3,4]
>>> b = [3,4,5]
>>> list(map(f, a, b))
[3, 12, 20]
```

Функция `filter(fun,list)` возвращает список только из тех элементов списка `list`, для которых функция `fun` возвращает истину.

```
>>> filter(lambda x: 2<x<=5, [0,1,2,3,4,5,6,7])
<filter object at 0x0000000003688E48>
>>> list(_)        # преобразовать в список результат последней секции
[3, 4, 5]
```

Все, что говорилось в этом параграфе о списках, относится и к кортежам, за исключением того, что кортеж – это список, доступный только для чтения. Поэтому все функции и методы, не меняющие содержимое, можно применять и к кортежам. Некоторые особенности мы опишем здесь.

Функция `tuple(последовательность)` позволяет преобразовать любую последовательность в кортеж. Если параметр не указан, то создается пустой кортеж.

```
>>> L=[1,2,3,4,5,6,7,8]
>>> t=tuple(L); t
(1, 2, 3, 4, 5, 6, 7, 8)
```

Чтобы задать кортеж из одного элемента, необходимо в конце последовательности внутри круглых скобок указать запятую. Именно запятые формируют кортеж, а не круглые скобки.

```
>>> t=(5,); type(t)
<class 'tuple'>
>>> r=(5); type(r)
<class 'int'>
>>> q=3,"str",[1,2]
>>> type(q)
<class 'tuple'>
```
Последний пример подтверждает, что не скобки формируют кортеж, а запятые.

В левой части присваивания можно написать несколько переменных через запятую, а в правой кортеж. Результатом является одновременное присваивание значений нескольким переменным.
```
>>>x,y,z=10,20,30
>>>y
20
```
При этом, сначала вычисляется кортеж в правой части, исходя из старых значений переменных (до присваивания). Потом одновременно всем переменным левой части присваиваются новые значения из этого кортежа. Так можно обменять значения нескольких переменных.
```
>>>x,y,z=z,x,y
>>>x,y,z
(30, 10, 20)
```
Строки. Одним из наиболее часто используемых типов данных является строка. Строки в Python являются упорядоченными последовательностями символов и относятся к неизменяемым типам данных. Иными словами, можно получить символ строки по индексу, но изменить его нельзя. Поэтому большинство строковых методов в качестве значения возвращают новую строку, а не модернизируют текущую.

Строку можно задать, используя одинарные или двойные кавычки.
```
>>> s1="Hello"
>>> s2='world'
>>> s1+s2
'Helloworld'
```
В языке Python никакого отличия между строкой в апострофах и строкой в кавычках нет. Если строка содержит кавычки, то ее лучше заключить в апострофы и наоборот.

Чтобы расположить строку в нескольких строчках, следует перед символом перевода строки указать символ \ (слэш).
```
>>> print("Привет \
мир")
Привет мир
```
Строки могут содержать специальные (управляющие) символы, такие как \n (символ новой строки), \t (табуляция) и некоторые другие. Если перед строкой поместить модификатор r, то специальные символы внутри строки будут выводиться как есть.

```
>>> print(r"Привет\nмир")
Привет\nмир
```
Иначе они (специальные символы) управляют отображением сроки.
```
>>> print("Привет\nмир")
Привет
мир
```
Сложение строк (оператор +) означает конкатенацию (соединение), а умножение на целое число (с любой стороны) – повторение строки несколько раз.
```
>>> 3*s1
'HelloHelloHello'
```
Строка, размещенная между утроенными кавычками, сохраняет свое форматирование.
```
>>> s="""Привет
мир"""
>>> s
'Привет\nмир'
>>> print(s)
Привет
мир
```
Строки, разделённые только пробелами, автоматически объединяются в одну строку (неявная конкатенация).
```
>>> print('Эта\n'
    'строка\n'
    'слипается')
Эта
строка
слипается
```
Операция `in` проверяет, содержится ли символ (или подстрока) в строке.
```
>>> 'M' in s1
False
>>> 'ell' in s1
True
>>> "a" not in s1
True
```
Длину строки можно получить с помощью функции `len()`.
```
>>> len(s)
5
```
Функция `len()` применима не только к строкам, но и к спискам, словарям и многим другим типам, про объекты которых разумно спрашивать, какая у них длина.

Функция `str(объект)` возвращает строковое представление объекта.
```
>>> a=5+3j
>>> str(a)
'(5+3j)'
```

В Python нет специального типа char, его роль играют строки длины 1. В Python 3 строки содержат символы в Unicode и, следовательно, могут содержать одновременно русские, латинские, греческие буквы и даже иероглифы.

```
>>> s='Текст + \u03C0 \u03A3 \u2154 \u2660 \u263A \u222B';print(s)
Текст + π Σ ⅔ ♠ ☺ ∫
```

Функция ord(символ) возвращает код символа, а функция chr(код) возвращает символ (строку длины 1).

```
>>> ord('D')            # код английской буквы D
68
>>> c=ord('Б'); c       # код русской буквы Б
1041
>>> chr(1043)           # строка из одного символа
'Г'
```

Символы в строке индексируются с 0.

```
>>> s='Mathematics'
>>> s[0]
'M'
```

Отрицательные индексы используются для отсчёта с конца.

```
>>> s[-4]
't'
```

Можно перебрать элементы строки в цикле.

```
>>> for i in range(len(s)): print(s[i], end=" ")
M a t h e m a t i c s
```

Можно просто указать строку в качестве последовательности цикла.

```
>>> for ch in s: print(ch, end=" ")
M a t h e m a t i c s
```

Можно выделить подстроку, указав диапазон индексов. Подстрока включает начальный символ диапазона, и не включает конечный.

```
>>> s[2:5]
'the'
```

Шаг диапазона может быть отрицательным

```
>>> s='Mathematics'
>>> s[::-1]        # строка в обратном порядке (отрицательный шаг)
'scitamehtaM'
>>> 'H'+s[1:]      # замена первого символа
'Hathematics'
```

Строки являются неизменяемым типом данных. Создав строку, нельзя изменить в ней один или несколько символов.

```
>>> s="Hello"
>>> s[1]='N'
Ошибка!
```

У объектов строк имеется множество методов. Например, метод upper() возвращает строку, целиком состоящую из прописных букв. Метод lower() возвращает строку из строчных букв.

```
>>> s3=s.upper(); s3
'HELLO'
```
Метод split() возвращает список, элементами которого являются все слова строки. По умолчанию расщепление производится по пустым промежуткам (пробелам, табуляциям и символам конца строки).
```
>>> a="Это тестовое предложение"
>>> L=a.split(); L
['Это', 'тестовое', 'предложение']
```
Метод строка.join(список_строк) создает строку из всех элементов списка, между которыми вставляется «базовая» строка.
```
>>> s4=" ".join(L); s4          # слова объединяются через пробел
'Это тестовое предложение'
>>> s5="+++".join(L); s5       # слова объединяются через '+++'
'Это+++тестовое+++предложение'
```
Метод lstrip() удаляет все пустые промежутки (пробелы, табуляции и символы конца строки) в начале строки; метод rstrip() – в конце; метод strip() – с обеих сторон.
```
>>> s="  Привет родной ВУЗ  "
>>> s.lstrip()
'Привет родной ВУЗ  '
>>> s.rstrip()
'   Привет родной ВУЗ'
>>> s.strip()
'Привет родной ВУЗ'
```
Имеются методы, которые проверяют, содержит ли строка символы какого либо типа.
```
>>> s="123"
>>> s.isdigit()   # возвращает True, если все символы цифры
True
>>> s.isalpha() # возвращает False, если хотя бы один символ не буква
False
>>> s="привет"
>>> s.isalpha() # возвращает True, если все символы буквы
True
>>> s.islower() # возвращает True, если все буквы строчные
True
>>> s.isupper() # возвращает True, если все буквы прописные
False
```
При написании программ возникают ситуации, когда нужно создать строку, подставив в ее определенные места некоторые данные. Это можно сделать, используя оператор '%' или метод format(). Оператор '%' имеет следующий формат:
```
Строка_со_спецификаторами_форматирования  % данные
```
Например,
```
>>> string="Peter"
```

```
>>> 'Hello, %s' % string
'Hello, Peter'
```
Здесь в строке `'Hello, %s'` использован спецификатор `%s`, означающий, что вместо него будет выведена строка `string` (не путайте спецификатор формата с оператором форматирования, для обозначения которых используется одинаковый символ `'%'`).

Строка со спецификаторами форматирования может содержать два типа объектов: обычные символы, которые копируются в результирующую строку без изменений, и спецификаторы формата. Спецификаторы формата начинаются символом `'%'` (процент), и преобразуют очередной аргумент к строковому представлению, зависящему от этого спецификатора.

```
>>> 'Нам %d лет' % 11            # %d – вывод целой части числа
'Нам 11 лет'
>>> import math
>>> 'Число e =%e' % math.e       # %e – вывод числа с плавающей точкой
'Число e =2.718282e+00'
>>> 'Число e =%f' % math.e       # %f – вывод числа в обычном формате
'Число e =2.718282'
```
Можно задать ширину поля вывода, указав количество символов для отображения данных.
```
>>> 'Число e =%12f' % math.e
'Число e =    2.718282'
>>> '%10f' % 15
' 15.000000'
```
Строка форматирования может содержать несколько различных спецификаторов форматирования.
```
>>> "Золотое сечение это деление какой-либо величины \
   приблизительно в отношении %d к %d или %5.3f" % (62,38,1.618)
'Золотое сечение это деление какой-либо величины приблизительно в отношении 62 к 38 или 1.618'
```
Метод `format()` выполняет функции, аналогичные оператору форматирования `%`, и имеет следующую форму:

```
Строка = Строка_со_спецификаторами_формата.format(данные)
```
В параметре `Строка_со_спецификаторами_формата` для указания места отображения данных используются фигурные скобки `{ }`.
```
>>> string="Peter"
>>> "Hello, { }".format(string)
'Hello, Peter'
>>> s="Число pi ={ }".format(math.pi)
>>> s
'Число pi =3.141592653589793'
```
В параметре `строка_со_спецификаторами_формата` внутри фигурных скобок можно указывать спецификаторы форматирования
```
>>> s="Число pi ={:e}".format(math.pi); s
'Число pi =3.141593e+00'
```

Описание используемых при форматировании строк, всех допустимых форматов, их ключей и флагов, потребует много места. Мы предлагаем читателю самостоятельно познакомиться с ними тогда, когда в этом возникнет необходимость.

Словарь – это набор объектов, доступ к которым осуществляется по ключу, а не по индексу, как в списках или массивах. Ключом может быть любой неизменяемый объект, например, число, строка или кортеж. Элементами словаря могут быть объекты произвольного типа данных. Чтобы получить элемент словаря, нужно указать ключ, который использовался при сохранении значения. Словари не являются последовательностями, и многие функции, используемые для работы с последовательностями, к ним не применимы. Словари относятся к изменяемым типам данных. Поэтому вы можете не только получить значение по ключу, но и изменить его.

Создать словарь можно с помощью функции `dict(...)`. Она имеет несколько форматов, основным из которых является следующий:

```
dict(<Ключ1>=<Значение1>[, ... , <КлючN>=<ЗначениеN>])
```

```
>>> d1 = dict(a=5, b=8); d1
{'b': 8, 'a': 5}
```

Если параметры не указаны, то создается пустой словарь.

```
>>> d0=dict(); d0
{}
```

Аргументом функции `dict(...)` может быть другой словарь (создается поверхностная копия словаря).

```
>>> d2=dict(d1);d2
{'b': 8, 'a': 5}
```

Аргументом функции `dict(...)` может быть список кортежей с парой элементов `(ключ, значение)`.

```
>>> d = dict([("a", 5), ("b", 8)]); d      # Список кортежей
{'b': 8, 'a': 5}
```

Аргументом функции `dict(...)` может быть список списков с парой элементов `[ключ, значение]`.

```
>>> d = dict ( [ ["a", 5], ["b", 8]]); d   # Список списков
{'b': 8, 'a': 5}
```

Создать словарь можно с помощью операции присваивания, перечислив справа от знака равенства все его элементы внутри фигурных скобок. Ключ и значение разделяется двоеточием, а пары (`ключ:значение`) разделяются запятыми.

```
>>> d = { "a": 5, "b": 8 }; d
{'b': 8, 'a': 5}
```

Заполнить словарь можно поэлементно. При этом ключ указывается внутри квадратных скобок.

```
>>> d = {}        # Создаем пустой словарь
>>> d["a"]=5      # Добавляем ключ "a"
>>> d["b"]=8      # Добавляем ключ "b"
```

```
>>> d
{'b': 8, 'a': 5}
```
Как и для списков, при создании словаря в переменной сохраняется ссылка на объект, а не сам объект.

Для создания поверхностной копии можно воспользоваться методом `copy( )` или функцией `dict()`.
```
>>> d3=d.copy(); d3
{'b': 8, 'a': 5}
>>> d3 is d              # сравнение объектов
False
>>> d3["b"]=25
>>> d,d3                 # Изменилось значение b в словаре d3
({'b': 8, 'a': 5}, {'b': 25, 'a': 5})
>>> d4=dict(d)
>>> d4["b"] = 55
>>> d,d4
({'b': 8, 'a': 5}, {'b': 55, 'a': 5})
```
Для создания полной копии словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy`.
```
>>> import copy
>>> d4=copy.deepcopy(d)
```
Чтение значений элементов словаря выполняется с помощью квадратных скобок, в которых указывается ключ.
```
>>> d['a']
5
>>> d['a'],d['b']
(5, 8)
```
Если элемента в словаре нет (нет соответствующего ключа) то генерируется сообщение об ошибке. Проверить наличие ключа можно с помощью оператора `in`.
```
>>> 'a' in d
True
>>> 'c' in d
False
```
Используя ключ, можно изменить элемент словаря. Если элемента с указанным ключом в словаре не окажется, то он будет добавлен.
```
>>> d['a']=1000
>>> d['c']='KhNu'
>>> d
{'a': 1000, 'b': 8, 'c': 'KhNu'}
```
Функция `len()` возвращает количество элементов (ключей) словаря.
```
>>> len(d)
3
```
Оператор `del` удаляет элементы словаря.
```
>>> del d['b']
```

```
>>> d
{'a': 1000, 'c': 'KhNu'}
```
Перебрать все элементы словаря можно с помощью цикла `for`, поскольку словари поддерживают итерацию.
```
>>> for k in d:
      print("d['{0}'] = {1}".format(k,d[k]),end=' ')

d['a'] = 1000 d['c'] = KhNu
```
Словари являются неупорядоченными наборами. Чтобы вывести элементы с сортировкой по ключам, нужно получить набор ключей с помощью метода `keys()`, преобразовать полученный объект в список, отсортировать его, а затем выводить элементы, используя отсортированный список ключей.
```
>>> d={'x':200, 'c':5, 'z':1000, 'a':10, 'y': 33}
>>> k=list(d.keys())  # получение списка ключей
>>> k.sort()          # сортировка списка ключей
>>> for key in k:
      print("d['{0}'] = {1}".format(key,d[key]),end=' ')

d['a'] = 10 d['c'] = 5 d['x'] = 200 d['y'] = 33 d['z'] = 1000
```
При сортировке ключей вместо метода `sort()` можно использовать функцию `sorted()`.
```
>>> for key in sorted(d.keys()):
      print("d['{0}'] = {1}".format(key,d[key]),end=' ')
```
Функции `sorted()` можно сразу передавать объект словаря, а не объект `dict_keys()`, возвращаемый методом `keys()`.
```
>>> for key in sorted(d):
      print("d['{0}'] = {1}".format(key,d[key]),end=' ')
```
У объектов `dict_keys()` есть одна особенность: они поддерживают операции на множествах.
```
>>> d1,d2={'a':10, 'z':1000,'b':'Hello'},{'y':50, 'b':'Hello','x':(21,77)}
>>> d1.keys () | d2. keys ()          # Объединение
{'a', 'x', 'b', 'y', 'z'}
>>> d1.keys () - d2. keys ()          # Разность
{'a', 'z'}
>>> d1.keys () & d2. keys ()          # Пересечение
{'b'}
>>> d1.keys () ^ d2. keys ()          # исключающее ИЛИ
{'a', 'x', 'y', 'z'}
```
Метод `values()` возвращает объект `dict_values()`, содержащий все значения словаря.
```
>>> d2.values()
dict_values(['Hello', (21, 77), 50])
>>> [elem for elem in d2.values()]
['Hello', (21, 77), 50]
```

Метод items() возвращает объект dict_items, который содержит все ключи и значения элементов попарно в виде кортежей.

```
>>> d={'x':200, 'c':5, 'z':1000, 'a':10, 'y': 33}
>>> d.items()
dict_items([('y', 33),('a',10),('c',5),('x',200),('z',1000)])
>>> list(d.items())
[('y',33),('a',10),('c',5),('x',200),('z',1000)]
```

Метод get(ключ,значение) возвращает значение, соответствующее ключу (если ключ в словаре есть). Если ключа нет, то возвращается None или значение, указанное во втором аргументе.

```
>>> d.get('a'), d.get('f'), d.get('f',111)
(10, None, 111)
```

Метод pop(ключ,значение) удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, то возвращается значение второго аргумента.

```
>>> d.pop('a')
10
>>> d
{'y': 33, 'c': 5, 'x': 200, 'z': 1000}
```

Метод clear() очищает словарь (удаляет все элементы).

Метод update(...) вносит изменения в текущий словарь, добавляя в него элементы. Если элемент с указанным ключом есть в словаре, то его значение меняется.

```
>>> d.update(d=555, c=111); d
{'y': 33, 'x': 200, 'c': 111, 'z': 1000, 'd': 555}
>>> d.update({'c':222,'d':333}); d
{'y': 33, 'x': 200, 'c': 222, 'z': 1000, 'd': 333}
```

Возможны другие способы записи добавляемых пар в методе update().

Генераторы словарей в Python конструируются подобно генераторам списков, но вся конструкция заключается в фигурные скобки, и внутри перед циклом for указываются два значения через двоеточие. Левое от двоеточия значение становится ключом, а правое – значением элемента.

```
>>> keys=['a','x','b','z','y']
>>> vals=[2,5,7,1,3]
>>> {k:v for (k,v) in zip(keys,vals)}
{'y': 3, 'b': 7, 'x': 5, 'a': 2, 'z': 1}
```

Напомним, что функция zip() в качестве аргументов принимает два (или больше) списка и возвращает итератор (поддерживает итерацию и может быть преобразован в список), составленный из кортежей соответствующих элементов этих списков.

```
>>> list(zip(keys,vals))
[('a',2),('x',5),('b',7),('z',1),('y',3)]
>>> import random
>>> keys=['a','b','c','d','e']
>>> dr= {k: random.randrange(0,10) for k in keys}; dr
```

```
{'c': 9, 'e': 2, 'a': 6, 'b': 4, 'd': 9}
>>> {k: v for (k, v) in dr.items() if v% 2 == 0}  # выбор четных значений
{'e': 2, 'a': 6, 'b': 4}
```

Строки, списки, кортежи и словари широко используются в Python программах. Здесь мы описали только основные функции, методы и приемы работы с ними. Имеется еще много других функций и методов, не описанных нами. Вы можете познакомиться с ними самостоятельно по справочной системе.

2.3 Функции, модули и пакеты

Функция – это участок кода, который можно вызвать из любого места программы по его имени. В предыдущих параграфах мы много раз использовали встроенные функции. Например, функция `len()` возвращает количество элементов последовательности (списка, кортежа, строки и т.д.). Здесь мы рассмотрим, как создаются функции.

Функция определяется с помощью ключевого слова `def` следующим образом:

```
def имя_функции( [аргументы] ) :
    ["""  Строка документирования """]
    тело функции
    [return значение]
```

Имя функции должно быть уникальным идентификатором. В круглых скобках через запятую можно указать один или несколько аргументов. Если функция не принимает аргументов, то указываются пустые круглые скобки. После круглых скобок ставится двоеточие. Операторы тела функции смещаются на одинаковое количество пробелов. Концом функции считается команда, смещенная на меньшее количество пробелов.

Если тело функции не содержит команд, то ее тело должно состоять из одной команды `pass`. Вот пример функции, которая ничего не делает:

```
def func ():
    pass
```

Необязательная команда `return` возвращает значение из функции. Она может отсутствовать. Тогда выполняются все команды тела функции, и возвращается значение `None`.

Определение функции должно быть расположено перед ее вызовом. Обычно определение функций размещают в самом начале программы после подключения модулей. При вызове функции значения аргументов передаются внутри круглых скобок через запятую. Если функция не принимает аргументов, то указываются пустые скобки.

Переменные, используемые в теле функции, являются локальными. Изменение локальной переменной в теле функции не влечет изменения одноименной глобальной переменной.

```
>>> def f(x):
    y=5;              # локальная переменная
    return x*y
```

```
>>> y=22                # глобальная переменная
>>> f(4)
20
>>> y                   # глобальная переменная не изменилась
22
```
В этом примере в теле функции `f(x)` присваивание `y=5` не меняет глобальную переменную, а создает локальную переменную `y` со значением `5`. Изменить таким способом глобальную переменную внутри функции нельзя. Однако глобальные переменные внутри функции можно использовать.
```
>>> def g(x):
      return x*z         # использование глобальной переменной z
>>> z=2
>>> g(4)
8
```
Чтобы изменить глобальную переменную в теле функции, она должна быть объявлена глобальной внутри функции с помощью ключевого слова `global`.
```
>>> def f(x):
      global y     # объявление глобальной переменной
      y=5          # изменение глобальной переменной
      return x*y
>>> y=10          # глобальная переменная
>>> f(5)
25
>>> y             # глобальная переменная изменилась
5
```
Аргументы в функцию передаются по значению. Если объект относится к неизменяемому типу, то изменение его значения внутри функции не затронет его значения снаружи.
```
>>> def g(x):
      x+=2
      return x
>>> z=4;  g(z)
6
>>> z             # значение z не изменилось
4
```
Если объект относится к изменяемому типу, то изменение его значения внутри функции повлияет на него снаружи.
```
>>> def fun(x):
      x[0]=5
      return sum(x)
>>> L=[1,2,3]
>>> fun(L)
10
>>> L
[5, 2, 3]
```

Как видно, значение первого элемента списка изменилось. Однако можно в функцию передавать и копию объекта.

```
>>> L=[1,2,3]
>>> fun(L.copy())
10
>>> L
[1, 2, 3]
```

Копию можно передать также по–другому.

```
>>> fun(L[:])
10
>>> L
[1, 2, 3]
```

Тип переменной при вызове функции может не совпадать с типом аргумента при определении функции. Например, в следующем примере мы вызываем функцию `f()`, созданную нами выше, с аргументом, который является списком.

```
>>> f([1,2,3])
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Операция умножения списка на число определена и означает его многократное повторение. В результате, вместо арифметического умножения, функция выполнила операцию повторения списка.

Если значения аргументов, передаваемые в функцию, содержатся в кортеже или списке, то перед кортежем (списком) следует указать символ * (звездочка).

```
>>> def prod(x,y):
    """ произведение элементов"""        # строка документации
    return x*y
>>> L=[6,3]
>>> prod(*L)
18
```

Все в языке Python является объектом, и функции не являются исключением. Команда `def` создает объект, имеющий тип `function`, и сохраняет ссылку на него в идентификаторе, указанном после инструкции `def`. Можно сохранить ссылку на функцию в другой переменной, тем самым, создав ее новое имя.

```
>>> f=prod
>>> f(5,8)
40
```

Можно передать ссылку на функцию в качестве аргумента другой функции.

```
>>> def fun(f,x,y):
    return f(x,y)**2
>>> fun(prod,2,3)
36
```

Функция может иметь необязательные аргументы. Чтобы сделать аргумент необязательным, в определении функции ему нужно присвоить начальное значение.

```
>>> def prod(x,y=1):
    return x*y**2
```

```
>>> prod(5)
5
>>> prod(5,2)
20
```
Таким образом, если второй аргумент функции `prod()` не задан, то его значение будет равно 1.

При определении функции необязательные аргументы должны следовать после обязательных, иначе возникает неоднозначность.

До сих пор мы использовали позиционную передачу аргументов в функцию. В Python можно передать значения аргументов также по ключевому имени, например,
```
>>> prod(y=4,x=3)
48
```
Для этого при вызове функции следует указать имя аргумента, использованное при ее определении, и после знака равенства – значение. Последовательность указания аргументов в таком случае не существенна, а важны только их имена.

Значение изменяемого объекта, передаваемого аргументом, может сохраняться между вызовами функции. Для этого такому аргументу следует назначить значение по умолчанию.
```
>>> def fn(a=[ ]):
    a.append(5)
    return a
```
Теперь
```
>>> L=fn([11]); L
[11, 5]
>>> fn(L)
[11, 5, 5]
>>> fn(L)
[11, 5, 5, 5]
```
Если перед аргументом в определении функции указать символ * (звездочка), то функции можно вызывать с произвольным количеством аргументов. Для примера создадим функцию суммирования произвольного количества чисел.
```
>>> def summa(*x):
    s=0
    for i in x:
        s+=i
    return s
>>> summa(10,15,25)
50
>>> summa(2,4,6,8,10)
30
```
Перед аргументом со звездочкой можно указать несколько обязательных аргументов и аргументов по умолчанию.
```
>>> def summa(a,b=1,*x):
    s=a+b
    for i in x:
```

```
            s+=i
        return s
>>> summa(5)
6
>>> summa(5,10,2,3,4)
24
```

Если при определении функции некоторые аргументы указываются после аргумента со звездочкой, то при вызове функции их обязательно надо задавать по имени, т.е. в виде имя=значение. Иначе не ясно, где заканчивается список аргументов со звездочкой.

```
>>> def fun(*z,a):
        s=sum(z)
        s*=a
        return s
>>> fun(2,3,4,a=5)    # параметр a должен передаваться по имени
45
>>> fun(a=5)
0
```

Кроме обычных функций, язык Python позволяет использовать анонимные функции, которые называются лямбда–функциями. Они создаются с помощью ключевого слова lambda следующим образом:

```
 lambda [аргумент_1 [,... ,аргумент_N]]: результирующее значение
```

В качестве результирующего значения указывается выражение, результат выполнения которого будет возвращен функцией. У лямбда–функций нет имени, поэтому их и называют анонимными функциями. Лямбда–функция возвращает ссылку на объект–функцию, которую можно сохранить в переменной или передать в качестве аргумента в другую функцию. Вызываются лямбда–функции как обычные, с использованием круглых скобок, внутри которых, через запятую, перечисляются значения аргументов.

```
>>> f1=lambda: 100                 # функция без аргументов
>>> f1()
100
>>> f=lambda x,y: x**2+y**2        # функция с двумя аргументами
>>> f(1,2)
5
>>> (lambda x: x**2)(5)            # создание и вызов лямбда функции
25
```

Аргументы лямбда–функций могут быть необязательными. Для этого им в определении функции нужно присвоить значение по умолчанию.

```
>>> f=lambda x,y=2: x**y
>>> f(4)
16
>>> f(4,3)
64
```

В Python можно создавать рекурсивные функции, которые могут вызывать сами себя.

```
>>> def factorial(n):
      if n==0:
            return 1
      else:
            return n*factorial(n-1)
>>> factorial(5)
120
```

Python разрешает создавать вложенные функции, причем уровень их вложения может быть неограниченным. Вложенная функция получает собственную локальную область видимости и имеет также доступ к идентификаторам внешней функции.

```
>>> def f1(x,y):
      z=2
      def f2(x):
            return x**z
      return f2(x)*y
>>> f1(4,3)
48
```

У функций, как объектов, имеется множество атрибутов. Обращение к ним выполняется указанием названия атрибута через точку после имени функции. Например, атрибут __name__ (два подчеркивания) содержит имя функции, а атрибут __doc__ – строку документирования.

```
>>> prod.__doc__
' произведение элементов'
```

Список всех атрибутов функции можно получить с помощью команды

```
dir(имя_функции)
```

Модулем в языке Python называется произвольный `py`–файл. Любой файл может импортировать другой модуль, получая, таким образом, доступ к функциям и переменным внутри импортированного модуля.

Откройте текстовый редактор и создайте следующий файл.

```
#  модуль testmod.py
def plus(x,y):
    return x+y

def mult(x,y):
    return x*y

PiNumber=3.141592
```

Этот файл содержит две функции и одну константу. Сохраните его под именем `testmod.py`.

Теперь создайте файл, в который импортируйте файл `testmod.py`, и в котором будем вызывать его переменную и функции.

```
# файл test.py
import testmod
zp=testmod.plus(5,6)
```

```
print(zp)
zm=testmod.mult(4,8)
print(zm)
print(testmod.PiNumber)
```
Сохраните этот файл под именем `test.py` в той же папке, что и предыдущий файл.

После импортирования файла можно получить доступ к функциям и переменным, определенным внутри него. Доступ выполняется путем указания имени функции (или переменной) через точку после имени модуля: `testmod.plus()`,`testmod.mult()` и `testmod.PiNumber`. При импортировании название модуля не должно содержать расширения имени и путь к файлу.

Запустите файл `test.py` на выполнение любым способом (из интегрированного текстового редактора командой `F5` или двойным щелчком мыши по имени файла). Если ошибок не было, то результат работы программы должен отобразиться в окне исполнительной системы.

Теперь посмотрите на содержимое папки, в которой сохранены файлы. Внутри нее автоматически был создан каталог __pycache__ с файлом `testmod.cpython-35.pyc`. Он содержит откомпилированный код модуля `testmod.py`. Этот код создается при первом импортировании модуля, и при всех последующих подключениях модуля `testmod` будет выполняться откомпилированный код. Исходный файл не будет использоваться, даже если его код изменить (пока не выйти из исполнительной системы Python и снова не зайти). Для повторной компиляции и загрузки модуля в той же сессии Python следует использовать функцию `reload()` модуля `imp`.

```
>>>from imp import reload        # импортирование функции reload()
>>>reload(testmod)
```

При использовании функции `reload()` следует знать, что идентификаторы, импортированные с помощью инструкции `from`, перезагружены не будут. В таком случае вам нужно перезагрузить интерпретатор Python.

Обратите также внимание на то, что после первой загрузки и компиляции модуля его исходный файл `testmod.py` больше не требуется. Это значит, что вы можете поставлять клиентам свои модули в откомпилированном виде, скрывая, тем самым, их код.

В приведенном выше примере, импортируемый модуль и тестовый файл были размещены в общей папке. В этом случае нет необходимости настраивать пути поиска модулей, т.к. каталог с исполняемым файлом автоматически добавляется в начало списка каталогов, в которых выполняется поиск импортируемых модулей. Каждый из таких каталогов является строкой, и имеется возможность просмотреть список этих строк.

```
>>> import sys
>>> sys.path
['', 'D:\\ProgramFiles\\Anaconda3\\python35.zip', ...,
'D:\\ProgramFiles\\Anaconda3\\lib\\site-packages\\Pythonwin']
```

У вас будет свой длинный список.

При поиске модуля список путей просматривается от начала. Как только в каком–либо каталоге обнаруживается модуль с заданным именем, поиск заканчивается.

Список путей, кроме каталогов со стандартными библиотечными модулями, содержит каталоги, заданные в переменной окружения PYTHONPATH.

На время текущей сессии работы интерпретатора Python пути поиска можно расширить, используя методы и функции работы со списками. Например, каталог с нашим файлом `testmod.py` можно добавить в конец списка следующим образом:

```
>>> sys.path.append(r"D:\Work\Python\StartProgs") # ваш путь поиска
```

Его можно было бы добавить и в начало списка путей поиска.

```
>>>sys.path.insert(0, r"D:\Work\Python\StartProgs") # ваш путь поиска
```

Теперь вы можете вызывать функции и постоянные этого модуля из командной строки.

```
>>> import testmod
>>> testmod.plus(5,6)
11
>>> testmod.mult(4,8)
32
>>> testmod.PiNumber
3.141592
```

Обратите внимание на модификатор `r`, стоящий перед строкой каталога. Он нужен для того, чтобы специальные последовательности символов внутри строки не интерпретировались как спецсимволы. Если модификатор не использовать, т.е. использовать обычные строки, то необходимо удвоить каждый слэш в пути, например,

```
>>> sys.path.append("D:\\Work\\Python\\StartProgs") # ваш путь поиска
```

Каждый модуль является объектом и его имя можно присвоить другой переменной. После этого обращаться к атрибутам и методам модуля можно посредством этого нового имени (через точку).

```
>>> tm=testmod
>>> tm.mult(4,5)
20
```

Получить список всех атрибутов и методов модуля позволяет функция `dir()`.

```
>>> dir(testmod)
['PiNumber', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'mult',
'plus']
```

Атрибут `__name__` содержит имя выполняемого модуля. Обычно, запускаемый модуль, имеет имя `__main__`, а импортируемый – имя своего файла.

```
>>> tm.__name__
'testmod'
```

В коде выполняемого модуля к атрибуту `__name__` можно обращаться по имени без использования какого–либо объекта.

Использовать имя модуля в виде строки нельзя.

```
>>> md='testmod'
>>> import md
Ошибка!
```

Чтобы подключить модуль, название которого содержится в строке, следует использовать функцию `__import__(строка_имени_модуля)`.

```
>>> m =__import__(md)
>>> m.__name__
'testmod'
```

Для доступа ко всем идентификаторам модуля без использования его имени, используется команда:

```
from имя_модуля import *
```

Например,

```
>>> from testmod import *
>>> plus(8,9)
17
>>> mult(8,9)
72
```

В этом случае идентификаторы, названия которых начинаются с символа подчеркивания, не импортируются.

У модулей есть атрибут `__all__`. Он содержит список идентификаторов модуля, которые будут импортироваться, если будет использоваться форма `from имя_модуля import *`. Идентификаторы в этом списке указываются в виде строк. Рассмотрим пример.

Добавьте в конце модуля `testmod.py` строку

```
__all__=['plus','PiNumber']
```

и сохраните файл с тем же именем (первоначальный код модуля `testmod.py` приведен ранее). Закройте вашу исполнительную систему и снова откройте ее (в Spider вы можете перезагрузить исполнительную консоль). Затем выполните команды

```
>>> import sys
>>> sys.path.insert(0, r"D:\Work\Python\StartProgs")  # ваш путь поиска
>>> from testmod import *
>>> plus(6,7)
13
>>> mult(6,7)
Ошибка!
>>> PiNumber
3.141592
```

Как видите, функция `plus()` и переменная `PiNumber` загрузились, а функция `mult()` – нет. Можно посмотреть список всех загруженных идентификаторов

```
>>> print(list(vars().keys()))
[..., '__loader__', 'PiNumber', 'sys', '__IDLE_eventloop_set',
'__builtins__', '__spec__', '__name__', '__package__', 'plus',
'__doc__']
```

У вас будет свой список, но в нем будут присутствовать строки 'PiNumber' и 'plus'.

Модули в Python можно объединять в пакеты. Пакетом называется каталог с модулями, распределенными по подкаталогам, в каждом из которых должен присутствовать файл __init__.py. Он является файлом инициализации и выполняется один раз при первом обращении к модулям соответствующего подкаталога, и может быть пустым.

В вашем рабочем каталоге создайте следующую структуру подкаталогов и файлов в них:

```
main.py
dir1\
     __init__.py
     testmod.py
     dir2\
          __init__.py
          testmod.py
          test2.py
```

Пусть содержимое файлов __init__.py будет одинаковым.

```
# файл __init__.py
print("Инициализация из ",__name__)
```

Код модуля testmod.py приведен выше. Мы только переместим его в каталог dir1, и еще создадим копию в каталоге dir2. Код файла test2.py приведен ниже.

```
# файл test2.py
msg="Модуль {}".format(__name__)
```

Теперь выполните команды

```
>>> import sys
>>> sys.path.insert(0, r"D:\Work\Python\StartProgs")   # ваш путь поиска
>>> import dir1.testmod as m1      # инициализация из каталога  dir1
Инициализация из dir1
>>> m1.plus(7,9)            # функция plus() из модуля dir1\testmod.py
16
>>> from dir1 import testmod as m2
>>> print(m2.mult(7,9))     # функция mult() из модуля dir1\testmod.py
63
```

Импорт модулей из каталога ..\dir1\dir2 выполняется следующим образом.

```
>>> import dir1.dir2.testmod as m3   # инициализация из каталога dir1\dir2
Инициализация из dir1.dir2
>>> m3.plus(6,15)           # функция plus() из модуля dir1\dir2\testmod.py
21
>>> import dir1.dir2.test2 as m4
>>> m4.msg                  # переменная msg из модуля dir1\dir2\test2.py
'Модуль dir1.dir2.test2'
```

Как видите, при первом обращении к каталогу dir1 автоматически выполнился файл dir1__init__.py, а при первом обращении к каталогу dir1\dir2 – файл dir1\dir2__init__.py.

Таким образом, чтобы импортировать модуль из вложенного каталога пакета, необходимо указать путь к нему, перечислив имена каталогов через точку. Если используется инструкция import, то после имен каталогов через точку нужно указать имя модуля.

```
>>>import dirl.dir2.testmod
```

Использовать такой длинный идентификатор неудобно, поэтому в Python предусмотрена возможность создания псевдонимов, которые указываются после ключевого слова as.

```
>>> import dir1.dir2.test2 as m4
```

Инструкция from позволяет импортировать сразу несколько модулей пакета. Для этого внутри файла инициализации _init_.py в атрибуте __all__ необходимо указать список модулей, которые будут импортироваться с помощью команды from пакет import *. Например,

```
#  файл __init__.py из каталога dir1\dir2
__all__ = [ "testmod", "test2"]
```

Описанные правила импортирования касаются случая, когда импорт выполняется из основной программы или командной строки. Но иногда приходится выполнять импорт между модулями пакета. Чтобы импортировать модуль, расположенный в том же каталоге, перед названием модуля указывается точка.

```
from .module import *
```

Для импортирования модуля, расположенного в родительском каталоге, перед названием модуля указываются две точки.

```
from ..module import *
```

Если необходимо обратиться уровнем выше, то указываются три точки.

```
from ...module import *
```

Чем выше уровень, тем больше точек необходимо указать.

3. Массивы и линейная алгебра

В «научном» Python основу всех математических вычислений составляет пакет NumPy. Он представляет библиотеку типов и функций для работы с массивами и матрицами, и является мультимодульным пакетом. Перечислим некоторые из содержащихся в нем модулей: random (случайный), linalg (линейная алгебра), fft (Fast Fourier Transform – быстрое преобразование Фурье), polynomial (работа с полиномами) и многие другие.

Вначале модуль NumPy следует загрузить. При этом для него можно создать более короткое имя.

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

После этого к любому объекту модуля можно обращаться, используя это короткое имя.

Важно также знать версию установленной у вас библиотеки. Это можно сделать командой

```
>>> np.__version__
'1.10.4'
```

3.1 Массивы

Создание массивов. Основным объектом `NumPy` является массив элементов, как правило, чисел. Существует несколько способов создать массив. Один из них состоит в создании массива из списка с помощью функции `array()`.

```
>>> a=np.array([0.0,2.5,5.2])
>>> a
array([0.,2.5,5.2])
```

Функция `print(...)` из ядра Python, печатает массив в виде списка

```
>>> print(a)
[0. 2.5 5.2]
```

Вот как создается двумерный массив

```
>>> b=np.array([[1,2,3],[4,5,6]])
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
```

Для доступа к элементам массива используются индексы. Нумерация индексов начинается с нуля. Обе следующие записи допустимы.

```
>>> b[0,1]
2
>>> b[1][2]
6
```

Массивы – это объекты со своими атрибутами и методами. Атрибут `ndim` содержит размерность массива.

```
>>> b.ndim
2
```

Количество элементов по каждой из координат возвращает атрибут `shape`.

```
>>> b.shape
(2, 3)
```

Атрибут `size` содержит количество элементов в массиве, а функция `len` вычисляет размер массива по первой координате.

```
>>> b.size
6
>>> len(b)
2
```

Перебирать элементы массива можно в цикле, например, так:

```
>>> for i in a:
      print(i)
0.0
2.5
5.2
```

Модуль NumPy предоставляет несколько различных числовых типов: int16, int32, int64, float32, float64. Массивы могут содержать элементы любого из них, но все элементы массива будут одного типа. Тип элементов массива можно узнать, используя атрибут dtype.

```
>>> b.dtype
dtype('int32')
>>> a.dtype
dtype('float64')
```

В модуле имеются большое количество команд/функций создания массивов. Приведем здесь некоторые из них.

Функция zeros() генерирует массив со всеми нулями, а функция ones() генерирует массив со всеми единицами. Размерность и количество элементов задается в аргументах этих функций.

```
>>> c=np.zeros(5)
>>> print(c)
[0. 0. 0. 0. 0.]
>>> A=5*np.ones([2,3])
>>> print(A)
[[ 5.  5.  5.]
 [ 5.  5.  5.]]
```

Здесь умножение числа на массив соответствует умножению каждого элемента массива на число.

При генерировании массива можно задать тип его элементов.

```
>>> d=np.ones(3,dtype=np.int16))
>>> print(d)
[1 1 1]
```

Функция eye() генерирует единичную матрицу, размер которой передается через аргумент.

```
>>> I=np.eye(3); print(I)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Для создания одномерного массива последовательно идущих чисел в модуле numpy имеются функции arange() и linspace(). Функция arange(a,b,шаг) использует шаг для создания массива чисел, начинающихся значением a, и не превосходящих значение b.

```
>>> a=np.arange(0.5,9.3,2.5)
>>> print(a)
[0.5 3.0 5.5 8.0]
```

Массив N чисел, равномерно распределенных на отрезке [a,b], создается функцией linspace(a,b,N). Начальная и конечная точки включаются.

```
>>> np.linspace(0,5,6)
array([ 0.,  1.,  2.,  3.,  4.,  5.])
```

Квадратный единичный массив можно создать функцией `identity(N)`, где `N` размер массива `N x N`.

```
>>> np.identity(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Функция `diag(v[,k=0])` создает квадратный массив (матрицу), на диагонали которого стоят элементы вектора `v` (одномерного массива), а остальные элементы равны нулю. Заметим, что когда мы говорим о векторе или матрице, то на самом деле подразумеваем одномерный или двумерный массив (как типы Python понятия массива и матрицы различаются), или списки соответствующих размеров.

```
>>> np.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Необязательный аргумент `k` представляет номер (побочной) диагонали, на которую будут поставлены элементы вектора `v`. Аргумент `k` может быть отрицательным, тогда побочная диагональ располагается снизу от главной.

```
>>> np.diag([1,2,3],1)
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

Построить массив, используя функцию над его индексами для вычисления элементов, можно следующим образом:

```
>>> def g(i):
    return i**2

>>> a=np.fromfunction(g,(5,),dtype=np.int32)  # запятая после пятерки
>>> print(a)
[ 0  1  4  9 16]
```

У функции `fromfunction(...)` обычно три аргумента. Первый – это идентификатор функции–генератора, которая в качестве аргументов принимает индексы элементов. Второй аргумент – кортеж размеров массива, в третьем необязательном аргументе указывается тип элементов. Для создания квадратного массива функция–генератор должна принимать два аргумента.

```
>>> def f(i,j):
    return i**2+j**2

>>> np.fromfunction(f, (3, 3), dtype=int)
array([[0, 1, 4],
       [1, 2, 5],
       [4, 5, 8]])
```

В Python можно создавать неименованные *lambda* функции. При генерировании массивов их удобно использовать в качестве первого аргумента.

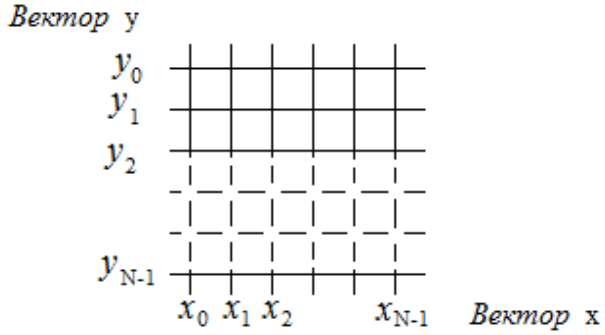
```
>>> np.fromfunction(lambda i, j: i + j, (3, 3))
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.],
       [ 2.,  3.,  4.]])
```

Нам часто придется использовать двумерные массивы с идентичными строками или столбцами. Для создания таких массивов используется функция `meshgrid()`.

```
>>> x=np.array([1,2,3])
>>> y=np.array([-1,1])
>>> X, Y = np.meshgrid(x, y)
>>> print(X)
[[1 2 3]
 [1 2 3]]
>>> print(Y)
[[-1 -1 -1]
 [ 1  1  1]]
```

Здесь генерируются две матрицы/массива. Строки массива X состоят из элементов вектора x, а количество строк определяется длиной вектора y. Наоборот, столбцы массива Y состоят из элементов вектора y, а количество столбцов определяется длиной вектора x.

Функция `meshgrid()` полезна для вычисления значения функций в узлах двумерной сетки, координаты узлов которой задаются парой векторов ***x*** и ***y***.



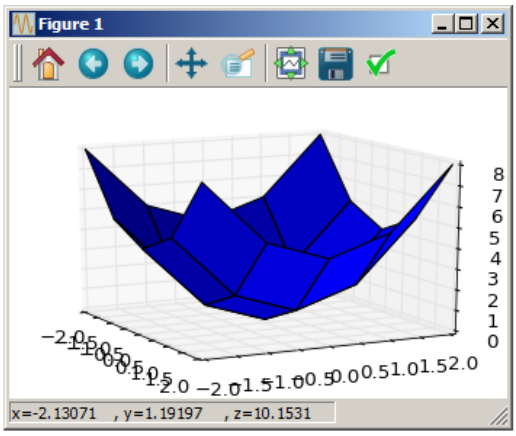
Для примера построим двумерный массив, элементы которого вычисляются в узлах сетки по формуле $z_{ij} = x_i^2 + y_j^2$ при $x_i = -2,-1,0,1,2$ и $y_j = -2,-1,0,1,2$.

```
>>> x=np.linspace(-2,2,5)
>>> y=np.linspace(-2,2,5)
>>> print(y)
[-2. -1.  0.  1.  2.]
>>> X,Y=np.meshgrid(x,y)
>>> print(X)
[[-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]]
```

```
>>> print(Y)
[[-2. -2. -2. -2. -2.]
 [-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]
 [ 2.  2.  2.  2.  2.]]
>>> Z=X**2+Y**2
>>> print(Z)
[[ 8.  5.  4.  5.  8.]
 [ 5.  2.  1.  2.  5.]
 [ 4.  1.  0.  1.  4.]
 [ 5.  2.  1.  2.  5.]
 [ 8.  5.  4.  5.  8.]]
```

Имея массивы координат вершин X,Y,Z, можно построить поверхность многогранника.

```
>>> from matplotlib.pyplot import *
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig=figure()              # открывает/создает графическое окно
>>> ax=Axes3D(fig)            # рисует 3–мерные координатные оси
>>> ax.plot_surface(X,Y,Z,rstride=1,cstride=1) # рисует поверхность
```



Графические функции мы будем обсуждать в следующей главе. Не вдаваясь в подробности, скажем, что здесь подключаются два графических модуля и затем вызываются функции, которые строят график многогранной поверхности по массивам `X`, `Y`, `Z` координат ее вершин.

В модуле `numpy` имеется оператор `mgrid[...]`, близкий по смыслу функции `meshgrid(...)` (его аргументы заключаются в квадратные скобки).

```
>>> x1, y1 = np.mgrid[0:8:2, -10:0:4]
>>> print(x1)
[[0 0 0]
 [2 2 2]
 [4 4 4]
 [6 6 6]]
```

```
>>> print(y1)
[[-10  -6  -2]
 [-10  -6  -2]
 [-10  -6  -2]
 [-10  -6  -2]]
>>> x2, y2 = np.meshgrid(np.arange(0, 8, 2), np.arange(-10, 0, 4))
>>> print(x2)
[[0 2 4 6]
 [0 2 4 6]
 [0 2 4 6]]
>>> print(y2)
[[-10 -10 -10 -10]
 [ -6  -6  -6  -6]
 [ -2  -2  -2  -2]]
```

Как видите, функция `meshgrid(...)` создает транспонированные массивы по сравнению с теми, которые создает оператор `mgrid[...]`.

Если результат действия последнего оператора `mgrid[...]` присвоить одной переменной

```
>>> g = np.mgrid[0:8:2, -10:0:4]
```

то она будет представлять массив из 2D массивов. Т.е. `g[0]` будет содержать массив `x1`, а `g[1]` – массив `y1`.

Если в команде `np.mgrid[start:stop:step]` параметр `step` содержит мнимую единицу, т.е. имеет вид `Nj`, то `N` представляет количество точек на отрезке `[start,stop]` (а не шаг!), и начальная и конечная точки включаются в массив.

```
>>> x3, y3 = np.mgrid[0:6:4j, -10:-2:3j]
```

Массивы `x3`, `y3` содержат те же значения, что и массивы `x1`, `y1`. В данном случае они отличаются типом элементов (вещественые и целые).

```
>>> type(x1[0][0])
numpy.int32
>>> type(x3[0][0])
numpy.float64
```

Случайные последовательности и массивы. Часто приходится использовать объекты, созданные случайным образом. В Python имеется модуль `random` (не путать с модулем `numpy.random`), который содержит функции, предназначенные для генерирования случайных чисел, символов, и т.д., а также последовательностей из них. Приведем краткое описание наиболее часто используемых функций этого модуля.

```
>>> import random as rnd
```

Функция `rnd.seed()` инициализирует генератор случайных чисел, используя системное время.

Функция `rnd.randrange(start,stop,step)` возвращает случайно выбранное целое число `X` из диапазона `start ≤ X < stop`, где `start,stop` и `step` должны быть целыми числами.

```
>>> rnd.seed()
>>> rnd.randrange(5,25,5)
15
```
Функция `N=rnd.randint(A,B)` возвращает случайное целое число `N` из диапазона `A ≤ N ≤ B`.
```
>>> rnd.randint(5, 25)
11
```
Функция `rnd.choice(последовательность)` возвращает случайный элемент из непустой `последовательности`, которая может быть списком, кортежем, массивом и т.д.
```
>>> rnd.choice([1,'x',"str",(21,3),[1,2,3]])
'str'
>>> rnd.choice(np.array([-1,3,2,6]))
3
```
Функция `rnd.shuffle(последовательность)` случайным образом перемешивает `последовательность`. Функция не работает для неизменяемых объектов.
```
>>> z=[1,'x',"str",(21,3),[1,2,3]]
>>> rnd.shuffle(z);z
[[1, 2, 3], 1, (21, 3), 'x', 'str']
```
Функция `rnd.random()` возвращает одно случайное число из диапазона `[0,1]`.
```
>>> rnd.random()
0.30184417938671937
```
Функция `rnd.sample(последовательность,k)` возвращает список длины `k` случайно выбранных элементов из `последовательности`.
```
>>> z=rnd.sample((1,'x',"str",(21,3),[1,2,3]), 3); z
['str',(21,3),[1, 2, 3]]
```
Функция `X=rnd.uniform(X0,X1)` возвращает случайное число с плавающей точкой из интервала `X0 < X < X1` (или `X0 ≤ X ≤ X1`, зависит от настроек округления).
```
>>> rnd.uniform(1.3, 2.4)
2.3132945207171103
```
Для генерирования массивов случайных чисел предназначены функции модуля `numpy.random`. Они имеют сходный синтаксис с одноименными функциями модуля `random`,. Приведем несколько примеров.
```
>>> import numpy as np
```
Функция `numpy.random.random(k)` генерирует массив `k` случайных вещественных чисел из полуинтервала `[0,1)`.
```
>>> np.random.random(4)        # массив 4–х вещественных чисел от 0 до 1
array([0.1501402, 0.68299761, 0.5332606, 0.59519775])
```
Имеется несколько похожих функций, отличающихся друг от друга в деталях.
```
>>> np.random.rand(3)          # массив 3–х вещественных чисел от 0 до 1
array([0.16960323, 0.59456467, 0.34565601])
np.random.rand(2,3)            # 2x3 массив вещественных чисел от 0 до 1
```

```
array([[ 0.04864568,  0.24654989,  0.56020694],
       [ 0.38628115,  0.86663464,  0.09863981]])
>>> np.random.sample()          # одно вещественное число от 0 до 1
0.812962885304781
>>> np.random.sample(3)         # массив 3–х вещественных чисел от 0 до 1
array([ 0.32872632,  0.21954113,  0.88342433])
>>> np.random.sample((2, 3))   # 2x3 массив вещественных чисел от 0 до 1
array([[0.13162862, 0.65688322, 0.6176331 ],
       [0.6012843 , 0.16264842, 0.19002568]])
>>> np.random.randint(3, 8, 5)          # массив 5–ти  целых чисел 3≤N<8
array([3, 4, 4, 3, 7])
>>> np.random.random_integers(3, 8, 5) # массив 5–ти целых чисел 3≤N≤8
array([8, 7, 3, 8, 3])
>>> np.random.randint(3, 8, (2, 4))   # 2x4 массив целых чисел 3≤N<8
array([[4, 5, 6, 7],
       [5, 6, 5, 3]])
```

Функция numpy.random.uniform(x0,x1,k) генерирует массив вещественных чисел, равномерно распределенных на интервале (x0,x1).

```
>>> np.random.uniform(3, 8, (2, 4)) # 2x4 массив вещественных чисел
array([[ 5.51213273, 4.95171824, 7.73454383, 3.37876069],
       [7.24894408, 7.70482718,  6.56001367, 4.54157435]])
```

Функция numpy.random.shuffle(массив) выполняет случайное перемешивание элементов массива.

```
>>> z = np.arange(8)
>>> np.random.shuffle(z);z
array([5, 3, 7, 6, 4, 2, 0, 1])
```

Функция numpy.random. permutation(N) возвращает массив перемешанных случайным образом целых чисел 0≤n<N.

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

В модуле numpy.random. имеются и другие, реже используемые функции.

Операции между массивом и скаляром. Операции +,–,*,/ между массивом и числом означает прибавление (вычитание) числа к каждому элементу массива, умножение (деление) на число каждого элемента массива.

```
>>> a=np.array([1.0,2.0,3.0])
>>> a+10
array([11., 12., 13.])
>>> print(a*5)
[ 5. 10. 15.]
```

Можно возвести в «числовую» степень каждый элемент массива

```
>>> a**2
array([ 1.,  4.,  9.])
```

Допустимы операции составного присваивания типа массив+=число.

```
>>> a+=10
```

```
>>> print(a)
[ 11.  12.  13.]
>>> b=np.array([[1,2],[3,4]])
>>> b+=10
>>> print(b)
[[11 12]
 [13 14]]
>>> b*=10
>>> b
array([[110, 120],
       [130, 140]])
```

Допустимы и многие другие поэлементные операции между массивами и числами, например, побитовые.

```
>>> b=np.array([[1,2],[3,4]])
>>> print(b | 1)
[[1 3]
 [3 5]]
```

Операции между массивами. С массивами можно выполнять поэлементные арифметические операции. Размерности массивов должны быть одинаковыми, или один из массивов должен быть одноэлементным. Во втором случае операция выполняется как между массивом и скаляром. Если операнды имеют разный тип элементов, то результат приводится к «старшему» типу. Например, можно сложить поэлементно два массива.

```
>>> a=np.array([1.0,2.0,3.0])
>>> b=np.array([10,20,30])
>>> print(a+b)
[11. 22. 33.]
```

Можно умножить поэлементно два массива. Это умножение не является скалярным (или матричным) произведением.

```
>>> print(a*b)
[10. 40. 90.]
```

Можно разделить поэлементно два массива.

```
>>> print(a/b)
[ 0.1  0.1  0.1]
```

Допустимо сложение массивов различной размерности.

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

Каждый элемент первого массива складывается с соответствующими элементами внутренних массивов второго слагаемого. Это работает так же, как и сложение скаляра с массивом. Схематически это выглядит, например, так:

$$\mathrm{obj} + \begin{bmatrix} \mathrm{obj}_1 \\ \mathrm{obj}_2 \end{bmatrix} = \begin{bmatrix} \mathrm{obj} + \mathrm{obj}_1 \\ \mathrm{obj} + \mathrm{obj}_2 \end{bmatrix},$$

где объектами obj могут быть скаляры, массивы и другие объекты, для которых определена операция '+', например, строки. Здесь квадратные скобки обозначают массив, а не список. Сложение массивов a и b, приведенное выше, действует следующим образом:

$$[1,2,3]+\begin{bmatrix}[1,2,3]\\[4,5,6]\end{bmatrix}=\begin{bmatrix}[1,2,3]+[1,2,3]\\[1,2,3]+[4,5,6]\end{bmatrix}=\begin{bmatrix}[2,4,6]\\[5,7,9]\end{bmatrix}.$$

Вот как выглядит схема сложения вектора–строки (одномерного массива) и вектора–столбца (двумерного массива с одним столбцом).

$$[1,2,3]+\begin{bmatrix}[4]\\[5]\\[6]\end{bmatrix}=\begin{bmatrix}[1,2,3]+[4]\\[1,2,3]+[5]\\[1,2,3]+[6]\end{bmatrix}=\begin{bmatrix}[5,6,7]\\[6,7,8]\\[7,8,9]\end{bmatrix}$$

Реализацией последней операции сложения являются следующие инструкции.

```
>>> x=np.array([1,2,3]);x
array([1, 2, 3])
>>> y=np.array([[4],[5],[6]]);y
array([[4],
       [5],
       [6]])
>>> x+y
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

Заметим, что существует простой способ преобразования вектора–строки y (одномерного массива) в вектор–столбец (двумерный массив с одним столбцом). Он реализуется командой y[:,None].

```
>>> y = np.linspace(-1.,1.,3); y
array([-1.,  0.,  1.])
>>> z=y[:,None]; z
array([[-1.],
       [ 0.],
       [ 1.]])
```

Тогда допустимо следующее сложение:

```
>>> x=np.array([1,2,3])
>>> x+y[:,None]
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.],
       [ 2.,  3.,  4.]])
```

Вот примеры других операций с массивами.

```
>>> a=np.array([10,20,30])
>>> b=np.array([2,4,6])
>>> print(a**b)        # поэлементное возведение в степень
[ 100   160000  729000000]
>>> b*=a               # составное поэлементное умножение
```

```
>>> b
array([ 20,  80, 180])
>>> a+=b            # составное поэлементное сложение
>>> a
array([ 30, 100, 210])
>>> a=np.array([10.,20.,30.])
>>> b=np.array([2,4,6])
>>> a/=b            # составное поэлементное деление
>>> a
array([ 5.,  5.,  5.])
>>> a=np.array([2,3,5])
>>> b=np.array([11,14,19])
>>> print(b%a)      # поэлементное вычисление остатка деления
[1 2 4]
```

Массивы можно поэлементно сравнивать. Результатом являются булевы массивы.

```
>>> a=np.array([1,2,3])
>>> c=np.array([0,3,2])
>>> print(a<c)
[False  True False]
```

Векторные и матричные операции над массивами. Одномерные и двумерные массивы интерпретируются как вектора и матрицы, если они используются в качестве аргументов функций, которые вычисляют скалярное или векторное произведение.

Функция `dot()` вычисляет скалярное произведение одномерных массивов одинаковой длины.

```
>>> np.dot(3, 4)      # скалярное умножение чисел
12
>>> a=np.array([1,2,3,4])
>>> b=np.array([-3,-2,-1,0])
>>> np.dot(a,b)       # скалярное умножение «векторов»
-10
```

Для двумерных массивов функция `dot()` выполняет матричное умножение.

```
>>> a=np.array([[1,2,3],[4,5,6]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
>>> b=np.array([[1,2],[3,4],[5,6]])
>>> print(b)
[[1 2]
 [3 4]
 [5 6]]
>>> np.dot(a,b)       # матричное умножение массивов
array([[22, 28],
       [49, 64]])
```

Массивы имеют метод dot( ), работающий аналогично функции dot( ).

```
>>> a=np.array([[1,2],[3,-4]])
>>> b=np.array([1,3])
>>> a.dot(b)
array([ 7, -9])
```

Это удобно, когда нужно выполнить подряд несколько операций скалярного произведения, например,

```
>>> a.dot(b).dot(b)
-20
```

Функции dot( ) можно передавать аргументы – списки, а не только массивы.

```
>>> a=[[-1,3],[4,1]]
>>> b=[[1,2],[3,4]]
>>> print(np.dot(a,b))
[[ 8 10]
 [ 7 12]]
```

Однако, списки не имеют метода dot( ).

Матричное произведение двумерных массивов выполняет также функция matmul(), которой можно передавать аргументы – списки. Она появилась в пакете numpy в версии 1.10, и в более ранних версиях вам следует использовать функцию dot.

```
>>> np.matmul(a, b)
array([[ 8, 10],
       [ 7, 12]])
```

Функцию matmul()можно использовать и для вычисления произведения матрицы на вектор.

```
>>> b=[1,2]
>>> np.matmul(a, b)
array([5, 6])
>>> np.matmul(b, a)
array([7, 5])
```

Функция matmul() с аргументами векторами вычисляет скалярное произведение.

```
>>> np.matmul([1,2,3],[4,5,6])
32
```

При этом компоненты вектора могут быть комплексными числами.

```
>>> np.matmul([2-1j,3+2j],[5+2j,2-3j])
(24-6j)
```

Функция matmul() отличается от dot() тем, что не вычисляет произведение чисел. Но она умеет вычислять произведение трехмерных массивов, интерпретируя каждый слой таких массивов как матрицу, и вычисляя послойное матричное произведение.

Для вычисления скалярного произведения векторов можно использовать функцию vdot().

```
>>> np.vdot([1,2,3],[4,5,6])
32
```

В отличие от matmul, функция vdot выполняет комплексное сопряжение первого вектора, если его компоненты являются комплексными числами.

```
>>> np.vdot([2-1j,3+2j],[5+2j,2-3j])
(8-4j)
```

Если аргументы функция vdot() являются массивами большей размерности, чем единица, то они (массивы) выравниваются до одномерных.

```
>>> a=[[1,2],[3,4]]
>>> b=[[1,-1],[2,3]]
>>> np.vdot(a,b)
17
```

Функция cross() вычисляет векторное произведение векторов (одномерных массивов или списков, содержащих три или два элемента)

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> np.cross(x, y)
array([-3,  6, -3])
>>> x = [1, 2]
>>> y = [4, 5]
>>> np.cross(x, y)
array(-3)
```

Внешнее произведение векторов (двух одномерных массивов) $a_{ij} = u_i v_j$ вычисляет функция outer().

```
>>> u=np.array([2,3]);
>>> v=np.array([5,10,15])
>>> a=np.outer(u,v);print(a)
[[10 20 30]
 [15 30 45]]
```

Тот же результат можно получить с помощью матричного умножения, но для этого вместо одномерных массивов u и v следует использовать двумерные массивы с теми же данными.

```
>>> U=np.array([[2,3]]).T; print(U)     # array.T – транспонирование массива
[[2]
 [3]]
>>> V=np.array([[5,10,15]]);V
array([[ 5, 10, 15]])
>>> np.dot(U, V)
array([[10, 20, 30],
       [15, 30, 45]])
```

Здесь, чтобы матричное умножение сработало, мы создали матрицу U, состоящую из одного столбца, и матрицу V, состоящую из одной строки.

Реорганизация массивов. Функция и метод reshape() реорганизовывают массив. Число элементов новой матрицы должно совпадать с числом элементов исходной. При этом данные массива не меняются.

```
>>> a=np.arange(0,1,0.25)
```

```
>>> a
array([ 0.  ,  0.25,  0.5 ,  0.75])
>>> b=a.reshape(2,2)
>>> b
array([[ 0.  ,  0.25],
       [ 0.5 ,  0.75]])
>>> d = np.arange(6).reshape((2,3))
>>> print(d)
[[0 1 2]
 [3 4 5]]
```

Можно изменить структуру массива, если атрибуту `shape` присвоить кортеж новых размеров. В этом случае изменится форма массива, но его данные останутся без изменений.

```
>>> b=np.arange(4);b
array([0, 1, 2, 3])
>>> b.shape
(4,)
>>> b.shape=2,2
>>> b
array([[0, 1],
       [2, 3]])
```

Точно также двумерный массив можно сделать одномерным.

```
>>> A=np.array([[1,2,3],[4,5,6]]); A
array([[1, 2, 3],
       [4, 5, 6]])
>>> A.shape
(2, 3)
>>> A.shape=(6,);  A
array([1, 2, 3, 4, 5, 6])
```

Как видите, присваивание значений атрибуту `shape` делает то же самое, что и метод `reshape()`.

Имеется функция `resize(a[,new_size])`, которая возвращает новый массив, элементами которого являются элементы массива `a`. Если количество элементов нового массива больше, чем исходного, то новый массив заполняется копиями элементов исходного. Элементы повторяются в том порядке, в котором они хранятся в памяти.

```
>>> a=np.array([3,5,7,-2,4])
>>> np.resize(a,(9,))
array([ 3,  5,  7, -2,  4,  3,  5,  7, -2])
>>> a=np.array([[0,1],[2,3]])
>>> np.resize(a,(2,3))
array([[0, 1, 2],
       [3, 0, 1]])
```

Метод `resize(new_size)` отличается от функции тем, что новый массив заполняется нулями, а не копиями данных исходного массива.

```
>>> b=np.array([8,9,7,5,3])
```

```
>>> b.resize(9)
>>> b
array([8, 9, 7, 5, 3, 0, 0, 0, 0])
```
Если на массив имеется другая ссылка, то изменение размера невозможно!
```
>>> b=np.array([8,9,7,5,3])
>>> c=b
>>> b.resize(9)
```
Ошибка!

Изменение размера массива невозможно также в случае, если до этого он подвергался реорганизации с помощью функции или метода `reshape()`.

Преобразовать одноэлементный массив в скаляр можно с помощью функции `asscalar()`.
```
>>> x=np.array([123]);  x
array([123])
>>> y=np.asscalar(x);y
123
```
Метод `diagonal()` возвращает вектор из диагональных элементов двумерного массива.
```
>>> a = np.arange(9).reshape((3,3))
>>> print(a)
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> a.diagonal()
array([0, 4, 8])
>>> a = np.arange(6).reshape((2,3))
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> a.diagonal()
array([0, 4])
```
Функция `transpose()` транспонирует массив.
```
>>> x=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> print(x)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> print(np.transpose(x))
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```
У массивов имеется также метод `transpose()`.
```
>>> x = np.array([[1,2,3]])  # двойные квадратные скобки
>>> x.transpose()
```

```
array([[1],
       [2],
       [3]])
```

Транспонирование выполняется над массивами размерности больше или равной 2, иначе массив оставляется без изменений. В предыдущем примере мы использовали двойные квадратные скобки для того, чтобы массив x был двумерным. Вместо функции и метода `transpose()` можно использовать атрибут `T`, который также возвращает транспонированный массив.

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

Создание массивов из других массивов. Массивы можно конструировать путем объединения уже имеющихся массивов. Функция `append()` «пристраивает» справа к одному массиву другой

```
>>> a=np.array([1,2,3])
>>> a=np.append(a,[10,20,30])
>>> a
array([ 1,  2,  3, 10, 20, 30])
```

Функции `delete`, `insert` и `append` не меняют массив, а возвращают новый, в котором удалены, вставлены в середину или добавлены в конец какие-то элементы.

```
>>> a=np.arange(2,9,1);print(a)
[2 3 4 5 6 7 8]
>>> b=np.delete(a,[2,4]);print(b)  # [2,4] – индексы удаляемых элементов
[2 3 5 7 8]
>>> c=np.insert(a,2,23)              # вставить на 2-е место число 23
>>> c
array([2,3,23,4,5,6,7,8])
>>> a=np.arange(2,9,1);print(a)
[2 3 4 5 6 7 8]
>>> c=np.insert(a,2,[5,10,15]);print(c) #  с 2–го индекса вставить 5,10,15
[ 2  3  5 10 15  4  5  6  7  8]
```

```
>>> np.insert(a,[0,2,5],[11,22,33])  # [0,2,5] – индексы мест,
                                     # [11,22,33]) – значения вставляемых элементов
array([11, 2, 3, 22, 4, 5, 6, 33, 7, 8])
```
По умолчанию многомерные массивы выравниваются до одномерных, а затем выполняется вставка элементов.
```
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.insert(a, 1, 5)
array([1, 5, 2, 3, 4, 5, 6])
```
Функция `hstack()` соединяет массивы по горизонтали.
```
>>> a=np.arange(2,6,1);print(a)
[2 3 4 5]
>>> b=np.arange(10,15,1);print(b)
[10 11 12 13 14]
>>> np.hstack((a,b))
array([ 2,  3,  4,  5, 10, 11, 12, 13, 14])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[5],[6],[7]])
>>> np.hstack((a,b))
array([[1, 5],
       [2, 6],
       [3, 7]])
```
Функция `vstack()` соединяет массивы по вертикали.
```
>>> a = np.array([1, 2, 3])
>>> b = np.array([5, 6, 7])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [5, 6, 7]])
>>> a=np.arange(1,7,1).reshape((2,3)); print(a)
[[1 2 3]
 [4 5 6]]
>>> b=np.arange(10,70,10).reshape((2,3));print(b)
[[10 20 30]
 [40 50 60]]
>>> np.vstack((a,b))
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [10, 20, 30],
       [40, 50, 60]])
```
В пакете `numpy` имеется «оператор» `numpy.r_` , предназначенный для конкатенации массивов. Кроме того, его можно использовать для их создания. Это не функция, его аргументы следует заключать в квадратные скобки.

```
>>> np.r_ [1:10:1]          # создание массива целых чисел 1≤X<10
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```
Ессли в команде `np.r_ [start:stop:step]` параметр `step` содержит мнимую единицу, т.е. имеет вид `Nj`, то `N` представляет количество точек на отрезке `[start,stop]` (а не шаг!), и начальная и конечная точки включаются в массив.
```
>>> z=np.r_ [1:4:7j]; z          # 7 - это количество точек
array([1., 1.5, 2., 2.5, 3.,3.5, 4.])
```
Если операнды являются массивами (или числами, разделенными запятыми), то они объединяются (по первой оси) в новый массив. В следующей команде мы создаем массив из двух массивов, между которыми вставляем два числа (нули).
```
>>> np.r_[np.array([1,2,3]), 0, 0, np.array([4,5,6])]
array([1, 2, 3, 0, 0, 4, 5, 6])
```
В следующем примере умножение списка на число интерпретируется как повторение списка.
```
>>> np.r_[-1:1:5j, [0]*3, 5, 6]
array([-1. ,-0.5, 0., 0.5, 1., 0., 0., 0., 5., 6.])
```
Если первым аргументом является строка, состоящая из одного символа – цифры, то он интерпретируется как номер оси, по которой будет выполняться конкатенация.
```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.r_ ['0', a, a]          # соединение по 0–ой оси
array([[0, 1, 2],
       [3, 4, 5],
       [0, 1, 2],
       [3, 4, 5]])
>>> np.r_ ['1', a, a]          # соединение по 1–ой оси
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```
Если первым аргументом является строка `'r'` или `'c'`, то результатом конкатенации по строкам (**r**ow) или столбцам (**c**olumn) является матрица.
```
>>> np.r_['r',[1,2,3], [4,5,6]]
matrix([[1, 2, 3, 4, 5, 6]])
>>> np.r_['c',[1,2], [5,6]]
matrix([[1],
        [2],
        [5],
        [6]])
```
Массивы можно расщеплять. Функция `hsplit()` разделяет массив по горизонтали на несколько подмассивов. Номера столбцов, по которым происходит разрыв, указываются в виде списка вторым аргументом этой функции.
```
>>> a=np.arange(9);print(a)
[0 1 2 3 4 5 6 7 8]
>>> x,y,z=np.hsplit(a,[2,5])
>>> print(x); print(y); print(z)
```

```
[0 1]
[2 3 4]
[5 6 7 8]
```
Если второй аргумент функции `hsplit` является числом N, то массив делится на N одинаковых по размеру подмассивов. При этом размеры массива и подмассивов должны быть согласованы.
```
>>> a = np.arange(8).reshape(2, 4); print(a)
[[0 1 2 3]
 [4 5 6 7]]
>>> x,y=np.hsplit(a, 2) # разбиение на два подмассива одинакового размера
>>> print(x)
[[0 1]
 [4 5]]
>>> print(y)
[[2 3]
 [6 7]]
```
Функция `vsplit()` разделяет массив по вертикали на несколько подмассивов. Если второй аргумент функции является числом N, то массив делится на N одинаковых по размеру массивов.
```
>>> a = np.array([[1,2],[3,4],[5,6],[7,8]])
>>> print(a)
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
>>> x,y=np.vsplit(a,2)  # разбиение на два подмассива одинакового размера
>>> print(x)
[[1 2]
 [3 4]]
>>> print(y)
[[5 6]
 [7 8]]
```
Если массивы должны иметь различные размеры, то номера строк, по которым будет происходить разбиение, указываются вторым аргументом в квадратных скобках.
```
>>> a = np.array([[1,2],[3,4],[5,6]]); print(a)
[[1 2]
 [3 4]
 [5 6]]
>>> x,y=np.vsplit(a,[1])
>>> print(x)
[[1 2]]
>>> print(y)
[[3 4]
 [5 6]]
>>> x,y,z=np.vsplit(a,[1,2])
```

```
>>> print(z)
[[5 6]]
```
Имеются также функции, перемещающие элементы в пределах массива, точнее возвращающие новые массивы, содержащие прежние данные, но в другом порядке. Например, функция `roll(x,n)` прокручивает вправо элементы массива `x` на `n` позиций.
```
>>> x = np.arange(9)
>>> print(x)
[0 1 2 3 4 5 6 7 8]
>>> x2=np.roll(x,2)
>>> print(x2)
[7 8 0 1 2 3 4 5 6]
```
Многомерный массив перед прокручиванием преобразуется в одномерный (в двумерном массиве выравнивание выполняется построчно), а затем его форма восстанавливается.
```
>>> x=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> print(np.roll(x,2))
[[8 9 1]
 [2 3 4]
 [5 6 7]]
```
Функция `sort()` возвращает отсортированную копию массива
```
>>> a=np.array([5,1,-2,6,0,9])
>>> b=np.sort(a); print(b)
[-2  0  1  5  6  9]
```
Метод `sort()` сортирует текущий массив.
```
>>> a=np.array([5,1,-2,6,0,9])
>>> a.sort(); print(a)
[-2  0  1  5  6  9]
```
В двумерных массивах данные сортируются в каждой строке
```
>>> b=np.array([[2,5,0],[4,1,3]])
>>> print(np.sort(b))
[[0 2 5]
 [1 3 4]]
```

Логические массивы. Логические (булевы) массивы состоят из логических элементов `True` и `False`.
```
>>> a = np.array([True, False, True, False])
>>> a
array([True,False, True,False],dtype=bool)
>>> b = np.array([1, 1, 0, 0], dtype=bool)
>>> b
array([True, True, False, False], dtype=bool)
```
Обычно логические массивы создаются при выполнении операций сравнения.
```
>>> x=np.array([-3.5, 2.2, -5.1, 4.3, 6.8, -8.0, 4.0])
>>> y=np.array([4.5, 2.2, -2.1, 4.3, 7.8, -1.0, 4.0])
>>> Z=x==y
```

```
>>> Z
array([False, True, False, True, False, False, True],dtype=bool)
>>> X=x<0
>>> X
array([True, False, True, False, False, True, False],dtype=bool)
```
Двойное неравенство для массивов не работает.
```
>>> -3<x<5
```
Ошибка!

Для сложных условий следует использовать функции пакета numpy, которые реализуют операции логической алгебры для массивов: logical_and(), logical_or(), logical_xor(), logical_not(). Например, последнее неравенство можно записать в виде
```
>>> np.logical_and(x>=-3,x<6)
array([False,True,False,True,False,False,True],dtype=bool)
```
Вот более сложное логическое условие
```
>>> np.logical_not(np.logical_or(x>0,y>2))
array([False,False,True,False,False,True,False],dtype=bool)
```
Функция any(логический_массив) пакета numpy возвращает True, если хотя бы один из элементов логического_массива равен True.
```
>>> np.any(X)
True
```
Функция all(логический_массив) возвращает True, если все элементы логического_массива равны True.
```
>>> np.all(X)
False
>>> a = np.array([1, -2, 3, -1])
>>> b = np.array([2, 2, 3, 2])
>>> c = np.array([6, 4, 7, 5])
>>> ((a <= b) & (b <= c)).all()
True
```
Функция array_equal(arr1, arr2) возвращает True или False в зависимости от того совпадают ли значения элементов массива arr1 с соответствующими элементами массива arr2.
```
>>> np.array_equal(a,b)
False
>>> d=np.array([4,1,6,2])-3
>>> np.array_equal(a,d)
True
```
Работа с элементами массивов. Доступ к элементам массива осуществляется с использованием индексации. При этом можно использовать список индексов.
```
>>> a = np.arange(6)
>>> print(a[[1,3,5]])
[ 1.  3.  5.]
```
Имеется возможность логической индексации. Для этого используются булевы массивы.

```
>>> a=np.array([2,-1,3,4,-5,7,-2,6,-7]); print(a)
[ 2 -1  3  4 -5  7 -2  6 -7]
>>> b=a>0;  print(b)        # массив b является логическим
[True False  True  True  False  True False  True False]
```

Логическая индексация имеет следующий формат:

`основной_массив[логический_массив],`

где логический массив должен иметь тот же размер, что и основной. Результатом этой операции является массив, составленный из элементов основного массива, для которых в логическом массиве соответствующие элементы равны `True`.

```
>>> c=a[b]; print(c)
[2 3 4 7 6]
```

Пусть из двумерного массива `B` требуется выбрать все отрицательные элементы и записать их в одномерный массив `F`.

```
>>> B=np.array([[4, -3, -1],[ 2, 7, 0],[ -5, 1, -2]]); print(B)
[[ 4 -3 -1]
 [ 2  7  0]
 [-5  1 -2]]
```

Сначала создадим логический массив.

```
>>> ind=B<0; print(ind)
[[False  True  True]
 [False False False]
 [ True False  True]]
```

Использование логического массива `ind` в качестве индекса исходного массива `B` позволяет решить поставленную задачу

```
>>> F=B[ind]; print(F)
[-3 -1 -5 -2]
```

Можно было обойтись и без вспомогательного массива `ind`, написав сразу

```
>>> F = B[B<0]; print(F)
[-3 -1 -5 -2]
```

Аналогично

```
>>> A=np.arange(-3,6).reshape(3,3);print(A)
[[-3 -2 -1]
 [ 0  1  2]
 [ 3  4  5]]
>>> C=A[A>B]; print(C)
[-2  2  3  4  5]
```

Логическое индексирование позволяет выбрать из массива элементы, удовлетворяющие определенным условиям, которые заданы логическим выражением.

Если требуется присвоить новое значение элементам массива, удовлетворяющим определенному условию, то массив с логическим индексом должен войти в левую часть оператора присваивания.

```
>>> A=np.arange(1,10).reshape(3,3);print(A)
```

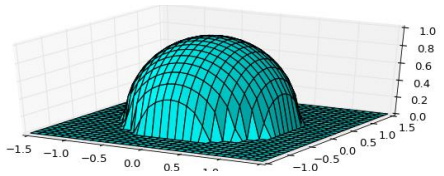
```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> B=np.array([[8,1,6],[3,5,7],[4,9,2]]); print(B)
[[8 1 6]
 [3 5 7]
 [4 9 2]]
>>> C=A<=B;  print(C)
[[ True False  True]
 [False  True  True]
 [False  True False]]
>>> A[C]=0;  print(A)
[[0 2 0]
 [4 0 0]
 [7 0 9]]
```

Пример. Построим график полусферы. В начале этого параграфа мы рассмотрели пример построения графика поверхности, заданной явным уравнением $z = f(x, y)$. Если для этого использовать формулу $z = \sqrt{1 - x^2 - y^2}$ $(-1 < x < 1, -1 < y < 1)$, то при некоторых значениях x,y под корнем будут стоять отрицательные числа. Поэтому задавать уравнение поверхности следует так:

$$z = \begin{cases} \sqrt{1 - x^2 - y^2}, & x^2 + y^2 < 1 \\ 0, & x^2 + y^2 \geq 1 \end{cases}.$$

Но тогда придется перебирать всю сетку значений x,y и обнулять подкоренное выражение, если оно отрицательно. Вот как это можно сделать, используя логическое индексирование.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x=np.linspace(-1.5,1.5,31)
y=np.linspace(-1.5,1.5,31)
X,Y=np.meshgrid(x,y)
Z=1-X**2-Y**2
Z[Z<0]=0                    # обнуление отрицательных значений
Z=np.sqrt(Z)
fig=plt.figure()
ax=Axes3D(fig)
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,color='cyan')
plt.show()
```



■

Если числовой массив преобразовать в логический (булев), то результат можно использовать при логическом индексировании. Для этого числовой массив можно преобразовать в логический с помощью функции `bool8(массив)`. Она любое число, отличное от `0`, преобразует в `True`, а нули – в `False`.

```
>>> D=np.arange(5); print(D)
[0 1 2 3 4]
>>> C=np.bool8([1,0,-1,0,-3]);
print(C)
[ True False  True False  True]
```

или по - другому

```
>>> C=np.array([1,0,-1,0,-3],dtype=bool)
```

Тогда

```
>>> D[C]=55; print(D)
[55  1 55  3 55]
```

Или наоборот

```
>>> D[C==False]=55; print(D)
[ 0 55  2 55  4]
```

Для получения индексов элементов, отличных от нуля, используется функция `nonzero(массив)`. Она возвращает массив индексов.

```
>>> a=np.array([2,0,-1,0,6,0,3])
>>> b=np.nonzero(a); b
(array([0, 2, 4, 6], dtype=int64),)
```

Эту функцию можно использовать для определения индексов элементов массива, удовлетворяющих любому условию

```
>>> a=np.array([2,0,-1,5,6,0,3,4,7,0,3])
>>> c=np.nonzero(a>2); c
(array([ 3,  4,  6,  7,  8, 10], dtype=int64),)
```

Здесь мы получили индексы всех элементов, значения которых больше 2. Это работает потому, что в логическом массиве `'a>2'` значение `False` интерпретируется как 0 и выражение `np.nonzero(a>2)` возвращает индексы True–элементов логического массива. У массивов есть метод `nonzero()`, который эквивалентен функции `nonzero()`.

Функция `flatnonzero()` эквивалентна функции `nonzero()` для одномерных массивов, а с многомерными она работает как с одномерными, предварительно выравнивая их.

```
>>> B=np.array([[3,-4,0],[-2,0,2],[0,0,-3]])
>>> np.flatnonzero(B)
array([0, 1, 3, 5, 8], dtype=int64)
```

Для массива определена операция получения среза `':'` (двоеточие), которая позволяет выбрать диапазон его элементов. В отличие от списков, она создает ссылку, указывающую на данные внутри массива. Поэтому, изменения, сделанные через нее, видны в исходном массиве.

```
>>> a=np.linspace(0,8,9); print(a)
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.]
```

```
>>> b=a[2:6]; b[0]=123; print(b)
[123.   3.   4.  5.]
>>> print(a)      # изменения в массиве b привели к изменениям в массиве a
[  0.   1.  123.   3.   4.   5.   6.   7.   8.]
```
Следующая команда также только создает новую ссылку на массив
```
>>> c=a[:]; c[2]=-55; print(a)
[  0.   1.  -55.   3.   4.   5.   6.   7.   8.]
```
Чтобы создать копию массива следует использовать функцию `array()`.
```
>>> A=np.array(a); A[2]=77;  print(A)
[ 0.  1.  77.  3.  4.  5.  6.  7.  8.]
>>> print(a)          # массив a остался без изменений
[ 0.  1. -55.  3.  4.  5.  6.  7.  8.]
```
Кроме того, для создания копии массива имеется метод `copy()`.
```
>>> B=a.copy();B[2]=999; print(B)
[ 0.  1.  999.  3.  4.  5.  6.  7.  8.]
>>> print(a)
[ 0.  1. -55.  3.  4.  5.  6.  7.  8.]
```
Аналогично создается копия фрагмента массива.
```
>>> b=a[2:6].copy(); print(b)
[-55.   3.   4.   5.]
>>> b[0]=99; print(a)
[0.  1. -55.  3.  4.  5.  6.  7.  8.]
```
Используя срез можно переписать массив в обратном порядке.
```
>>> b=a[len(a):0:-1]; b[1]=77; print(b)
[  8.  77.   6.   5.   4.   3. 123.   1.]
>>> print(a)      # изменения в массиве b приводят к изменениям в массиве a
[  0.   1.  123.   3.   4.   5.   6.  77.   8.]
```
Можно использовать диапазон с неединичным шагом.
```
>>> a=np.linspace(0,8,9); b=a[0:9:2]; print(b)
[ 0.  2.  4.  6.  8.]
```
Элементам среза можно присвоить скаляр или значения из массива «правильного» размера.
```
>>> a=np.arange(1,9)
>>> a[3:7]=10; print(a)
[ 1  2  3 10 10 10 10  8]
>>> a[3:7]=np.arange(-4,0,1);print(a) # срезу присваивается массив
[ 1  2  3 -4 -3 -2 -1  8]
>>> a=np.linspace(0,8,9,dtype=np.int16); print(a)
[0 1 2 3 4 5 6 7 8]
>>> a[1:9:3]=np.array([111,222,333]); print(a) # срез с шагом 3
[ 0  111   2   3  222   5   6  333   8]
```
Присваивание срезу массива неправильного размера приводит к ошибке.
```
>>> a[1:9:3]=np.array([111,222,333,444])
Ошибка!
```
Доступ к столбцу или строке массива с помощью среза возвращает массив
```
>>> a=np.array([[1,2,3],[4,5,6],[7,8,9]])
```

```
>>> a[:,1]        # выводим 2-й столбец
array([2, 5, 8])
>>> a[2,:]        # выводим 3-ю строку
array([7, 8, 9])
```
Используя срез, можно изменять значения элементов двумерного массива.
```
>>> a=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a[0:2,0:2]=0;print(a)
[[0 0 3]
 [0 0 6]
 [7 8 9]]
```
Однако срез `a[0:2][0:2]` работает не так, как предыдущий.
```
>>> a=np.array([[1,2,3],[4,5,6],[7,8,9]]
>>> a[0:2][0:2]=0;print(a)          # осторожно!
[[0 0 0]
 [0 0 0]
 [7 8 9]]
```
Разберем эту особенность для двумерных массивов. Использование «повторной» индексации `a[X][Y]` работает как суперпозиция двух операций индексации – вначале выполняется левая `a[X]`. Она возвращает одномерный массив, если X индекс, или набор одномерных массивов (т.е. двумерный массив), если X является срезом.
```
>>> a=np.array([[1,2,3],[4,5,6],[7,8,9]]
>>> a[1:3]
array([[4, 5, 6],
       [7, 8, 9]])
```
Затем выполняется вторая индексация `(a[X])[Y]`. Если X являлся индексом, то из одномерного массива `a[X]` выбирается элемент или диапазон элементов Y. Если X являлся срезом, то `a[X]` является двумерным массивом и индекс Y выбирает соответствующую строку из этого массива.
```
>>> a[1:3][1]
array([7, 8, 9])
```
Если Y диапазон индексов, то для двумерного массива `a[X]` диапазон Y снова выбирает диапазон строк, т.е. опять возвращается двумерный массив
```
>>> a[1:3][0:1]
array([[4, 5, 6]])
```
Если X и Y индексы, то в соответствии с описанным алгоритмом возвращается элемент a_{XY} исходного массива, т.е. в этом случае `a[X,Y]=a[X][Y]`.

Обратите также внимание на следующую особенность присваивания переменных массивам. Присваивание другого имени `b=a` не создает новый массив, а создает синоним его имени.
```
>>> a=np.arange(0,5,1);print(a)
[0 1 2 3 4]
```
Следующие команды присваивают переменной `b` имя массива. Но этим не создается новый массив, а создается другое имя. Если изменить элементы

массива b, то тем самым изменяется и массив a. Изменения, сделанные через массив b, видны и в исходном массиве a.

```
>>> b=a; b[2:4]=0;print(a)
[0 1 0 0 4]
```

Мы уже говорили, что для копирования данных одного массива в другой, нужно использовать метод copy.

```
>>> a=arange(0,5,1);print(a)
[0 1 2 3 4]
>>> b=a.copy()
>>> b[2:4]=0;print(b); print(a)
[0 1 0 0 4]
[0 1 2 3 4]
```

Перебирать элементы массива можно в цикле.

```
>>> A=np.linspace(0,5,6); A
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> for elem in A:
      print(elem**2, end=" ")
0.0 1.0 4.0 9.0 16.0 25.0
```

А следующий цикл выполняется по строкам двумерного массива.

```
>>> B=np.array([[3,-4,0],[-2,0,2],[0,0,-3]])
>>> for row in B:
      print(row)
[ 3 -4  0]
[-2  0  2]
[ 0  0 -3]
```

Функции работы с массивами. Модуль numpy содержит элементарные функции (дублирующие функции модуля math), аргументами которых могут быть массивы. В таком случае эти функции применяются к массивам поэлементно. Они называются универсальными функциями (ufunc).

```
>>> type(np.sin)
<class 'numpy.ufunc'>
>>> c=np.linspace(0,np.pi/2,4)
>>> print(np.sin(c))
[0.  0.5  0.8660254  1.]
```

Методы массивов sum(),prod(),max(),min(),mean(),std() вычисляют сумму и произведение элементов массива, максимальный и минимальный элемент, среднее и среднеквадратичное отклонение.

```
>>> b=np.array([[1,2,3],[4,5,6]]);b
[[1 2 3]
 [4 5 6]]
>>> b.sum()
21
>>> b.prod()
720
```

```
>>> b.max()
6
>>> b.mean()
3.5
>>> b.std()
1.707825127659933
```
Эти методы умеют вычислять соответствующие значения по каждой из координат массива.
```
>>> a=np.array([[1,2,3],[4,5,6],[7,8,9]]); print(a)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> a.sum()                 # сумма всех элементов
45
>>> print(a.sum(0))         # суммирование по нулевому индексу
[12 15 18]
```
Сравните с суммой по столбцам
```
>>> a[:,0].sum(), a[:,1].sum(),a[:,2].sum()
(12, 15, 18)
```
Аналогично
```
>>> print(a.sum(1))         # суммирование по первому индексу
[ 6 15 24]
```
Сравните с суммой по строкам
```
>>> a[0,:].sum(), a[1,:].sum(),a[2,:].sum()
(6, 15, 24)
```
Аналогично работают методы `prod()`, `max()`, `min()` и т.д.
```
>>> a.min(1)                # минимум по строкам
array([1, 4, 7])
```
Методы `argmin()` и `argmax()` возвращают индексы минимального и максимального элементов по всему массиву или по строкам (столбцам). При определении индекса по всему массиву он вначале выравнивается построчно до одномерного массива. Затем возвращается индекс элемента в таком массиве.
```
>>> a=np.array([[2,-1,5],[7,3,-2],[3,9,5]]);a
array([[ 2, -1,  5],
       [ 7,  3, -2],
       [ 3,  9,  5]])
>>> a.argmin()        # индекс минимального элемента
5
>>> a.argmax()        # индекс максимального элемента
7
>>> a.argmin(0)       # индексы минимальных по столбцам
array([0, 0, 1], dtype=int64)
>>> a.argmin(1)       # индексы минимальных по строкам
array([1, 2, 0], dtype=int64)
```

```
>>> a.argmax(0)      # индексы максимальных по столбцам
array([1, 2, 0], dtype=int64)
>>> a.argmax(1)      # индексы максимальных по строкам
array([2, 0, 1], dtype=int64)
```

Накопительной суммой массива $\mathbf{a}=\{a_0,a_1,...,a_j,...,a_n\}$ называется массив $\mathbf{c}=\{c_0,c_1,...,c_j,...,c_n\}$, где $c_j=\sum_{i=0}^{j}a_i$. Она вычисляется с помощью функции `cumsum()`.

```
>>> a=np.array([-1,3,2,-4,5])
>>> np.cumsum(a)
array([-1,  2,  4,  0,  5], dtype=int32)
```

Функция `numpy.diff(a, n=1,axis=-1)` вычисляет конечные разности `n`–го порядка массива `a` вдоль заданной оси `axis`. В простейшем случае функция `diff(a)` вычисляет конечные разности первого порядка массива `a` по последней оси. Например, для одномерного массива $\mathbf{a}=\{a_i\}_{i=0}^{n}$ функция возвращает вектор $\mathbf{d}=\{\Delta a_i\}_{i=1}^{n}=\{a_i-a_{i-1}\}_{i=1}^{n}$, т.е. вектор, составленный из разностей соседних элементов.

```
>>> np.diff([1,3,8,25])
array([ 2,  5, 17])
>>> np.diff([[1,3,8,25],[3,4,5,6]])
array([[ 2,  5, 17],
       [ 1,  1,  1]])
>>> np.diff([[1,3,8,25],[3,4,5,6],[1,2,3,4]],axis=0)
array([[  2,   1,  -3, -19],
       [ -2,  -2,  -2,  -2]])
```

Функции `round()`, `around()` (и метод `round()`) округляют элементы массивов до ближайших значений, имеющих заданное количество десятичных знаков.

```
>>> x=np.array([-3.546, 2.215, -5.149, 4.375, 6.85, -8.4237])
>>> b=x.round(2); b
array([-3.55,  2.22, -5.15,  4.38,  6.85, -8.42])
```

Имеется возможность вычисления кусочных выражений вида

$$f(x)=\begin{cases} f_1(x), x<x_1 \\ f_2(x), x_1\le x<x_2 \\ \quad \cdots\cdots \\ f_n(x), x_{n-1}\le x<x_n \\ f_{n+1}(x), x\ge x_n \end{cases}.$$

Знаки условных операций, содержащие равенства (меньше или равно, больше или равно) могут стоять в любой стороне условий.

Для вычисления кусочных выражений в модуле `numpy` определена функция `piecewise()`, которая имеет следующий формат:

```
piecewise(x, список_условий, список_функций [,*args,**kw])
```
Здесь в качестве первого аргумента `x` должен стоять массив и результатом будет массив того же размера.

Вот как можно вычислять функцию абсолютного значения, которая равняется $-x$, если $x<0$, и x, если $x\geq 0$.

```
>>> x=np.array([-3.5, 2.2, -5.1, 4.3, 6.8, -8.4])
>>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
array([ 3.5,  2.2,  5.1,  4.3,  6.8,  8.4])
```
Если участков «аналитичности» больше двух, то следует использовать операции логической алгебры для массивов: `logical_and()`, `logical_or()`, `logical_xor()`, `logical_not()`.
```
>>> np.piecewise(x,[x<-3,np.logical_and(x>=-3,x<6),x>=6],[-1,1,2])
array([-1.,  1., -1.,  1.,  2., -1.])
>>> X=np.piecewise(x,[x<-3,
            np.logical_and(x>=-3,x<4),
            np.logical_and(x>=4,x<=6),
            x>=6],
           [lambda t: -t,
            lambda t: t**2,
            lambda t: t+100,
            lambda t: t**0.5])
>>> X
array([3.5, 4.84, 5.1, 104.3, 2.60768096, 8.4])
```
Имеется возможность преобразовывать Python функции для работы с аргументами массивами. Это называется векторизацией. Векторизация позволяет преобразовать скалярную функцию так, чтобы в качестве аргументов она могла принимать последовательности элементов (списки, кортежи и т.д.) или массивы, а не только скаляры. После векторизации результатом работы функции будет последовательность элементов или массив, полученные поочередным применением скалярной функции к каждому элементу аргумента. Векторизация используется для удобства, а не для ускорения вычислений. Для векторизации используется функция `numpy.vectorize(pyfunc[,...])` . Ее первым аргументом является скалярная Python функция. Вот несколько примеров.
```
>>> from math import sin
>>> import numpy as np
```
Векторизация библиотечной скалярной функции
```
>>> mysin = np.vectorize(sin)  # векторизация функции math.sin
>>> x=[0,np.pi/6,np.pi/4,np.pi/2,np.pi/2]
>>> mysin(x)                          # аргумент - список
array([ 0. ,0.5 , 0.70710678, 1. , 1. ])
```
Векторизация Python функции пользователя.
```
>>> def f(x):
      return sin(x)**2+x
```

```
>>> fvec = np.vectorize(f)        # векторизация функции пользователя
>>> fvec(x)                       # аргумент - список
array([ 0. , 0.77359878, 1.28539816, 2.57079633, 2.57079633])
```
Векторизация лямбда функции одной переменной.
```
>>> g=lambda x: x**2+x+1
>>> m=np.array([[2,5],[1,8]])
>>> gvec=np.vectorize(g)          # векторизация лямбда функции
>>> gvec(m)                       # аргумент - двумерный массив
array([[ 7, 31],
       [ 3, 73]])
```
Векторизация лямбда функции двух переменных.
```
>>> x=np.linspace(0,4,5); x
array([ 0., 1., 2., 3., 4.])
>>> y=np.linspace(-2,2,5); y
array([-2.,-1., 0., 1., 2.])
>>> h=lambda x,y: x**2+y**2     # лямбда функция двух аргументов
>>> hvec=np.vectorize(h)
>>> hvec(x,y)                   # аргументы – два одномерных массива
array([ 4., 2., 4., 10., 20.])
```

Имеется большое количество других функций и методов манипулирования массивами. Кроме того, мы не описали необязательные аргументы многих функций. С ними вы можете познакомиться по справочной системе пакета numpy. Для этого следует выполнить команду

`help(np.имя_функции)`,

если np – это имя модуля numpy.

Подведем небольшой итог. Массивы похожи своим поведением на списки, но имеют следующие особенности:

- элементы массива всегда представляются объектами одного типа, например, все элементы являются вещественными числами;
- в момент создания массива количество его элементов должно быть известно; в порядке исключения оно может быть изменено, например, функцией `resize()`;
- объекты массивов импортируются из пакета NumPy и не являются стандартной частью Python;
- над массивами можно выполнять «векторные» и «матричные» операции с помощью функций пакета NumPy.

3.2 Элементы линейной алгебры

Функции подмодулей `numpy.linalg` и `scipy.linalg` реализуют основные операции линейной алгебры. Они в значительной степени дублирую друг друга. Однако лучше использовать модуль `scipy.linalg` в силу большей эффективности его процедур. Кроме того, имеется модуль `numpy.matlib`, содержащий функции, которые возвращают/создают матрицы, а не массивы (в

NumPy тип «матрица» не совпадает с типом «массив»). В этом модуле для объектов типа «матрица» основные матричные операции реализованы в виде операций, а не функций.

Здесь мы опишем только наиболее важные функции этих модулей. При этом для краткости не будем упоминать о необязательных аргументах многих функций. В тех случаях, когда они есть, но мы о них не упоминаем, в описании будут использоваться квадратные скобки, означающие необязательность, и троеточие, обозначающее пропуск в описании одного или нескольких аргументов. Например, `norm(a[,ord,...])` означает, что `a` является обязательным аргументом, кроме того может использоваться необязательный аргумент `ord` и могут быть другие необязательные аргументы. Возвращаемые функциями величины обычно понятны по смыслу, однако более подробную информацию о них вам следует получить из справочной системы.

Функция `det(a[,...])` вычисляет определитель квадратной матрицы.

```
>>>import numpy as np
>>>import scipy.linalg as la
>>>a = np.array([[1, 2],[3, 4]])
>>>la.det(a)      # вычисление определителя квадратного массива
-2.0
```

Функция `norm(a[,ord,...])` вычисляет норму массива (вектора или матрицы).

```
>>> la.norm(a)          # норма Евклида
5.4772255750516612
>>> np.sqrt(1**2+2**2+3**2+4**2)
5.4772255750516612
```

Второй аргумент определяет вид нормы (Евклида, Фробениуса, Минковского и т.д.). Если аргумент `ord` опущен, то вычисляется норма Евклида.

Функция `inv(a[,...])` вычисляет обратную матрицу/массив.

```
>>> a=np.array([[1,2],[3,4]])
>>> a1=la.inv(a); a1
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Проверим, что `a*a1=I`. Имеем (напомним, что функция `matmul` появилась в пакете `numpy` начиная с версии 1.10)

```
>>> np.matmul(a,a1)        # или np.dot(a,a1)
array([[  1.00000000e+00,   0.00000000e+00],
       [  8.88178420e-16,   1.00000000e+00]])
```

Здесь недиагональные элементы практически равны нулю. Неточность возникает из–за погрешности вычислений. Если в последнем матричном умножении выполнить округление с точностью до 10 цифр после десятичной точки, то получим

```
>>> np.round_(np.dot(a,a1),10)
array([[ 1.,  0.],
       [ 0.,  1.]])
```

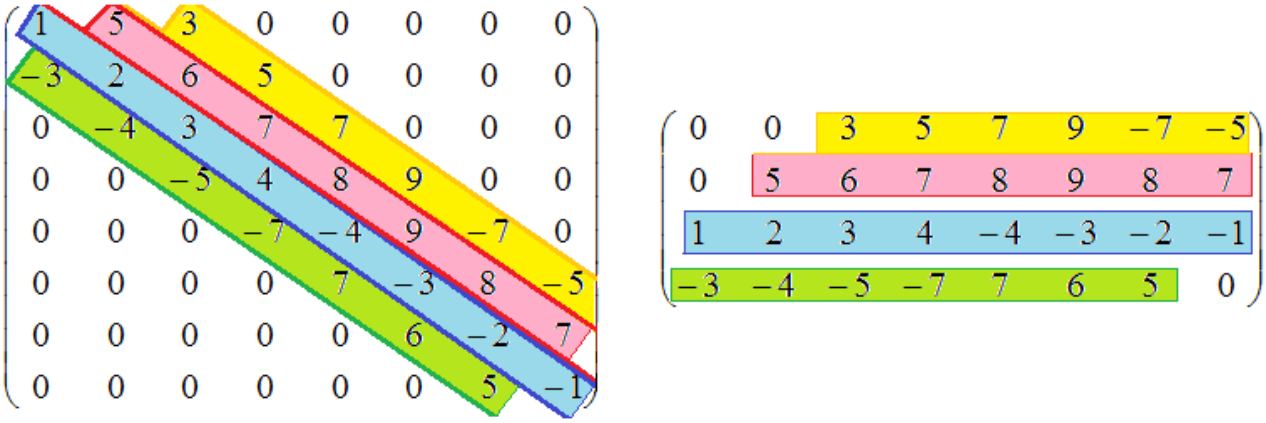
Функция `solve(a,v[,...])` решает систему линейных уравнений $\mathbf{a}\cdot\mathbf{u}=\mathbf{v}$, где **a** – квадратная матрица (массив) коэффициентов, а **v** – вектор (массив) правых частей системы.

```
>>> a=np.array([[2,-3],[3,1]])
>>> v=np.array([1,7])
>>> u=la.solve(a,v); print(u)
[ 2.  1.]
```

То же решение можно получить с использованием обратной матрицы.

```
>>> a1=la.inv(a)
>>> print(np.matmul(a1,v))
[ 2.  1.]
```

В модуле имееется функция `solve_banded(...)`, которая решает уравнение $\mathbf{a}\cdot\mathbf{u}=\mathbf{v}$ с матрицей коэффициентов **a**, имеющей ленточную структуру. У ленточных матриц все ненулевые элементы сосредоточены на главной диагонали и нескольких побочных. Для таких матриц нет необходимости хранить все коэффициенты системы, поскольку большинство из них равно нулю и, кроме того, имеются эффективные алгоритмы решения таких систем уравнений. На следующем рисунке слева показан пример ленточной матрицы, а справа – ее «нелеточная» форма (прямоугольная матрица, данные которой используются в вычислениях).



Нелеточная форма строится по следующему правилу. Каждая диагональ исходной матрицы представляется строкой прямоугольной матрицы. Строки, представляющие верхние побочные диагонали, имеют ведущие нули. Количество нулей равно «номеру» побочной диагонали, отсчитываемому от главной диагонали. Строки, представляющие нижние диагонали дополняются нулями справа.

Для решения уравнение $\mathbf{a}\cdot\mathbf{u}=\mathbf{v}$ с ленточной матрицей коэффициентов **a** используется функция `solve_banded((l,u),A,v[,...])`. Здесь `(l,u)` – кортеж, в котором `l` является количеством ненулевых нижних побочных диагоналей, а `u` – количеством ненулевых верхних побочных диагоналей. `A` – матрица коэффициентов в «нелеточной» форме (описано выше), `v` – вектор правой части системы.

Пример 1. Решим краевую задачу $y''(x)=1, y(0)=0, y(1)=0$ методом конечных разностей (ее точное решение известно $y=x\cdot(x-1)/2$).

В соответствии с этим методом область непрерывного изменения аргумента заменяется дискретным множеством (сеткой), а дифференциальное уравнение заменяется разностным. В нашем примере сеткой будет разбиение отрезка [0,1] точками $x_i=i\cdot h, i=0,1,2,...,n-1$, где $h=1/(n-1)$ – шаг сетки. Вторая производная приближается выражением $y''(x_i)\approx\frac{y_{i+1}-2y_i+y_{i-1}}{h^2}$, где $y_i=y(x_i)$. Приближенная задача формулируется следующим образом: найти дискретную функцию (массив) $y=(y_0,y_1,...,y_{n-1})$, которая удовлетворяет системе уравнений.

$$\begin{cases} y_0=0 \\ \frac{y_{i-1}-2y_i+y_{i+1}}{h^2}=1,\ i=1,2,...,n-2. \\ y_{n-1}=0 \end{cases}$$

Здесь первое и последнее уравнение выражают краевые (граничные) условия. В матричной записи эта система имеет вид:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \cdots \\ y_{n-3} \\ y_{n-2} \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 0 \\ h^2 \\ h^2 \\ \cdots \\ h^2 \\ h^2 \\ 0 \end{pmatrix}$$

Матрица коэффициентов ленточная, и систему уравнений можно решать с помощью функции `solve_banded(...)`. Решение задачи можно условно разделить на три части. Вначале создаем матрицу коэффициентов в «неленточной» форме.

```
>>>import numpy as np
>>>import scipy.linalg as la
>>>n=9          # количество узлов сетки
# формирование матрицы
>>>a0=np.ones(n);
>>>a1=np.ones(n)*(-2)
>>>a=np.vstack((a0,a1,a0));
>>>a[0][0]=0; a[0][1]=0
>>>a[1][0]=1; a[1][-1]=1
>>>a[2][-1]=0; a[2][-2]=0
>>>print(a)
```

```
[[ 0.  0.  1.  1.  1.  1.  1.  1.  1.]
 [ 1. -2. -2. -2. -2. -2. -2. -2.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  0.  0.]]
```

Затем формируем вектор правой части системы.

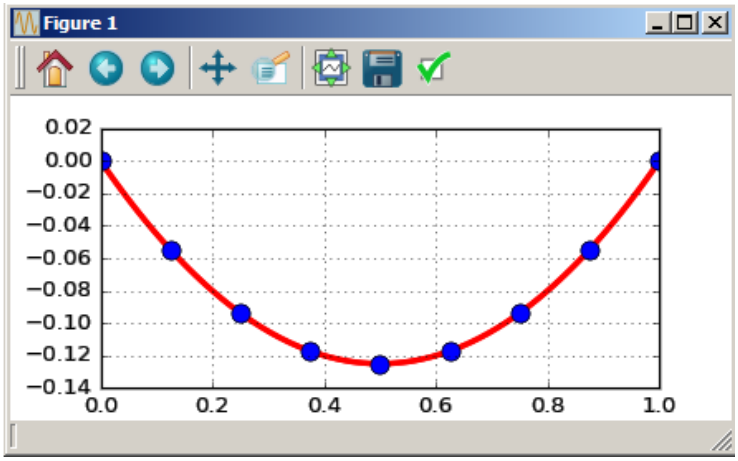
```
>>>h=1/(n-1)
>>>v=np.ones(n)*(h**2)
>>>v[0]=0
>>>v[n-1]=0
>>>print(v)
[ 0.   0.015625  0.015625  0.015625  0.015625  0.015625  0.015625
0.015625  0. ]
```

Теперь решаем систему уравнений.

```
>>>y=la.solve_banded((1,1),a,v);
>>>print(y)
[ 0.  -0.0546875 -0.09375  -0.1171875 -0.125 -0.1171875
-0.09375 -0.0546875  0. ]
```

Для сравнения строим график дискретного решения (x_i, y_i) (множество найденных нами точек), и кривой $y = x \cdot (x-1)/2$, представляющей точное решение.

```
>>>import matplotlib.pyplot as plt
>>>xx=np.linspace(0,1,101)
>>>yy=xx*(xx-1)/2
>>>plt.plot(xx, yy, '-r',  linewidth=3)  # кривая точного решения
>>>x=np.linspace(0,1,n);
>>>plt.plot(x, y, 'o',  markersize=10) # точки дискретного решения
>>>plt.grid(True)
>>>plt.show()
```



Пояснение графических функций вы найдете в следующей главе. Вкратце это выглядит так. Вначале импортируется графический модуль `matplotlib.pyplot`. Затем с помощью функции `plot()` рисуются точки и кривая. Функции `grid(True)` рисует координатную сетку. Функция `show()` показывает график в отдельном окне.

■

Команда `r=lstsq(a,v[,...])` решает систему линейных уравнений $\mathbf{a}\cdot\mathbf{x}=\mathbf{v}$ методом наименьших квадратов. Решением считается вектор **x,** для которого евклидова норма вектора $\mathbf{v}-\mathbf{a}\cdot\mathbf{x}$ минимальна. Функция возвращает кортеж, первым элементом которого является массив решений, вторым элементом – невязка, третьим идет ранг матрицы коэффициентов, а затем массив сингулярных чисел. Т.о. массив решений находится в `r[0]`.

Пример 2. Проведем аппроксимирующую прямую через `n` точек.

Линейная аппроксимация состоит в отыскании коэффициентов *a* и *b* уравнения $y=a\cdot x+b$ таких, чтобы все экспериментальные точки лежали максимально близко к аппроксимирующей прямой. В качестве меры близости множества точек $(x_i,y_i)_{i=1}^{n}$ и прямой обычно принимают выражение $\sum_{i=1}^{n}\left(y_i-(a\cdot x_i+b)\right)^2$, которое и следует минимизировать. Эту задачу можно переформулировать так: решить методом наименьших квадратов переопределенную систему линейных уравнений $a\cdot x_i+b=y_i\ (i=1,2,\ldots,n)$ относительно неизвестных *a* и *b*. Систему можно записать в матричном виде.

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \cdots & \cdots \\ x_n & 1 \end{pmatrix}\cdot\begin{pmatrix} a \\ b \end{pmatrix}=\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

Пусть координаты точек равны $(0,-1),(1,0.5),(2,1.5),(3,1)$. Следующий код создает массив `A` коэффициентов системы, и вектор `y` ее правой части.

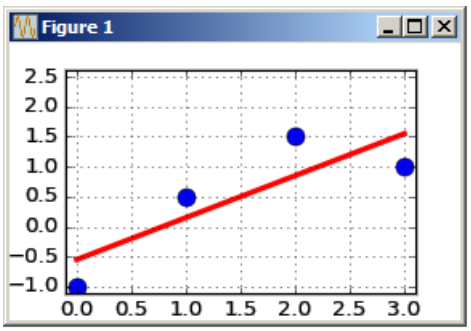
```
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.5, 1.5, 1])
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 2.,  1.],
       [ 3.,  1.]])
```

Теперь решаем систему уравнений методом наименьших квадратов.

```
>>> a, b = la.lstsq(A, y)[0]
>>> print(a,b)   # коэффициенты a и b уравнения прямой
0.7 -0.55
```

Поясним решение графически.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o',  markersize=10)
>>> plt.plot(x, a*x + b, 'r', linewidth=3)
>>> plt.xlim(-0.1,3.1);plt.ylim(-1.1,2.6);plt.grid(True)
>>> plt.show()
```



Здесь вначале импортируется графический модуль `matplotlib.pyplot`. Затем с помощью функции `plot()` рисуются исходные точки и аппроксимирующая прямая. Функции `xlim()`, `ylim()`задают диапазон графика по X и Y координатам, а функция `grid(True)` рисует координатную сетку. Функция `show()` показывает график в отдельном окне.

■

Произведение Кронекера матриц (массивов) вычисляется с помощью функции `kron(a, b)`.

```
>>> u=np.array([[2,3]])
>>> v=np.array([[5,10,15]]).T
>>> la.kron(u,v)
array([[10, 15],
       [20, 30],
       [30, 45]])
```

Функция `tril(m[,k=0])` возвращает матрицу с элементами, расположенными выше `k`–ой диагонали, равными нулю. Например, `k=1` соответствует первой верхней побочной диагонали, а `k=–1` обнуляет все элементы, которые расположены выше нижней побочной диагонали.

```
>>> la.tril([[1,2,3,4],[4,5,6,7],[7,8,9,10],[10,11,12,13]], -1)
array([[ 0,  0,  0,  0],
       [ 4,  0,  0,  0],
       [ 7,  8,  0,  0],
       [10, 11, 12,  0]])
```

Аналогично, функция `triu(m[,k=0])` возвращает матрицу со всеми нулевыми элементами, которые расположены ниже `k`–ой диагонали.

Функция `w,v=eig(a[,...])` вычисляет собственные значения и собственные векторы квадратной матрицы `a`. Собственные значения возвращаются в векторе `w`, и повторяются в нем столько раз, какова их кратность. Собственные векторы возвращаются в матрице `v`, каждый столбец которой представляет собственные вектор, отвечающий соответствующему собственному значению в векторе `w`.

```
>>> w, v = la.eig(np.diag((1, 2, 3)))
>>> print(w); print(v)
[ 1.  2.  3.]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

```
>>> w, v = la.eig(np.array([[5,4],[8,9]]))
>>> w,v
(array([ 1.+0.j, 13.+0.j]), array([[-0.70710678, -0.4472136 ],
        [ 0.70710678, -0.89442719]]))
```

Таким образом, в этом примере собственному числу `w[0]=1` отвечает собственный вектор, элементы которого стоят в первом столбце матрицы `v`.

```
>>> v[:,0]
array([-0.70710678,  0.70710678])
```

Легко проверить, что собственные векторы нормированы.

```
>>> print(v[:,1])        # 2 – й собственный вектор, отвечает с.ч. w[1]=13
[-0.4472136  -0.89442719]
>>> la.norm(v[:,1])
1.0
```

Для решения системы дифференциальных уравнений $\dot{\mathbf{x}} = \mathbf{A}\,\mathbf{x}$, где $\mathbf{x}$ – вектор, $\dot{\mathbf{x}} = \dfrac{d\,\mathbf{x}}{d\,t}$, $\mathbf{A}$ – матрица:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \cdots \\ x_n \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \cdots & \cdots & \cdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

надо найти собственные числа $\lambda_i \ \ (i = 1,...,n)$ матрицы **A**. Если все собственные числа простые, то каждому λ_i соответствует решение $C_i \mathbf{v}_i e^{\lambda_i t}$, где C_i – произвольная постоянная, $\mathbf{v}_i$ – собственный вектор, соответствующий этому λ_i

Пример 3. Решим задачу Коши для системы дифференциальных уравнений

$$\begin{cases} \dot{x} = 2x + y \\ \dot{y} = 3x + 4y \end{cases}, \ x(0) = 2, y(0) = 5.$$

Составим матрицу коэффициентов и найдем ее собственные числа.

```
>>> A=np.array([[2,1],[3,4]])
>>> w, v = la.eig(A); print(w); print(v)
[ 1.+0.j  5.+0.j]
[[-0.70710678 -0.31622777]
 [ 0.70710678 -0.9486833 ]]
```

Отбросим, если нужно, нулевые мнимые части собственных чисел.

```
>>> w=w.real; print(w)
[ 1.  5.]
```

Общее решение можно записать в виде

$$C_0\,\mathbf{v}[:,0]\exp(w[0]t) + C_1\,\mathbf{v}[:,1]\exp(w[1]t),$$

где **v**[:,0] и **v**[:,1] являются собственными векторами. Если сюда подставить начальные значения `t=0` и $x(0) = x_0$, $y(0) = y_0$, то мы приходим к задаче решения линейной алгебраической системы уравнений

$$\mathbf{v} \cdot \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix},$$

где **v** – матрица, столбцы которой состоят из собственных векторов, т.е. это матрица `v`, которую вернула функция `eig( )`. Решим эту систему уравнений.

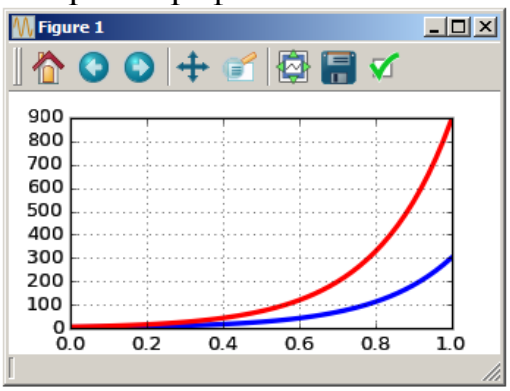
```
>>> r=np.array([3,5])        # вектор правых частей
>>> c=la.solve(v,r); print(c)
[-1.41421356 -6.32455532]
```

Построим вектора $x_i = x(t_i), y_i = y(t_i)$, содержащие значения функций `x(t)`, `y(t)` решения в дискретные моменты времени (в точках некоторого вектора `t`).

```
>>> t=np.linspace(0,1,51) # значения независимого аргумента
>>> et=np.exp(np.outer(w,t))
>>> cet=c.reshape((2,1))*et    # cet=(c*et.T).T
>>> x,y=np.dot(v,cet)        # значения функций x(t), y(t) в точках вектора t
```

Строим графики решений.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(t, x, '-b',  t,y,'-r', linewidth=3)  # построение графиков
>>> plt.grid(True)      # отображение сетки
>>> plt.show()          # открыть графическое окно
```



■

В модуле `scipy.linalg` есть много других функций, предназначенных, для решения задач линейной алгебры, когда матрицы имеют специальных вид. Кроме того, имеются функции предназначенные для представления матриц через другие матрицы, например, в произведение верхне– и нижне–треугольных. Имеется много функций, предназначенных для конструирования матриц, например, блочно – диагональных, матриц Ханкеля, Гильберта и других.

В модуле имеются также матричные функции. Например, функция

$$\exp m(M) = \sum_{n=0}^{\infty} \frac{M^n}{n!}$$

не совпадает с поэлементной функцией $\exp(M)$.

```
>>> la.expm(M)
array([[ 1.54308063,  1.17520119],
       [ 1.17520119,  1.54308063]])
```

Сравните

```
>>> np.exp(M)
array([[ 1.        ,  2.71828183],
       [ 2.71828183,  1.        ]])
```

Имеются матричные функции, такие как `cosm(M)` (матричный косинус), `sinm(A)` (матричный синус), `sqrtm(A[, ...])` матричный корень квадратный и другие.

Кроме модуля `scipy.linalg`, в пакете NumPy имеется аналогичный модуль `numpy.linalg`. Многие его функции идентичны функциям модуля `scipy.linalg`. Однако в `numpy.linalg` есть несколько функций, которых нет в `scipy.linalg`. Например, функция `numpy.linalg.matrix_power(M, n)` возводит матрицу `M` в степень `n`.

```
>>> M=np.array([[1,2],[2,-1]])
>>> M2=np.linalg.matrix_power(M, 2)
>>> print(M2)
[[5 0]
 [0 5]]
>>> M=np.array([[1,2,3],[4,5,-1],[2,5,-3]])
>>> np.linalg.matrix_power(M, 2)
array([[15, 27, -8],
       [22, 28, 10],
       [16, 14, 10]])
```

Все перечисленные выше функции выполняли матричные операции с массивами. Если же вы хотите использовать для матричных операций привычные обозначения, тогда вы должны преобразовать массив в матричный тип. Для создания матриц в NumPy предназначен специальный класс `matrix`, объекты которого всегда являются двумерными. Вместо `matrix` можно использовать сокращение `mat`. В отличие от массивов (объектов класса `ndarray`), для объектов типа `matrix` операция `'*'` обозначает матричное произведение, а `'**'` – операцию матричного возведения в степень. Атрибут `matrix.T` возвращает транспонированную матрицу, а атрибут `matrix.I` – обратную матрицу. Для создания объектов `matrix` можно использовать специальный синтаксис, когда конструктору матрицы передается текстовая строка. В ней элементы каждой строки матрицы разделяются пробелами, а строки – символом `';'` (точка с запятой).

```
>>> A = np.matrix('1 1;1 1');print(A)
[[1 1]
 [1 1]]
>>> B=np.mat([[1,-2],[-1,1]]);print(B)
[[ 1 -2]
 [-1  1]]
>>> A+B
matrix([[ 2, -1],
        [ 0,  2]])
>>> A*B
matrix([[ 0, -1],
        [ 0, -1]])
```

```
>>> B*A
matrix([[-1, -1],
        [ 0,  0]])
>>> B**2
matrix([[ 3, -4],
        [-2,  3]])
>>> B.T
matrix([[ 1, -1],
        [-2,  1]])
>>> B.I                      # вычисление обратной матрицы
matrix([[-1., -2.],
        [-1., -1.]])
>>> C=B.getI(); print(C)    # вычисление обратной матрицы
[[-1. -2.]
 [-1. -1.]]
>>> A = np.arange(12)
>>> A.shape = (3,4)
>>> print(A)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> M = np.mat(A.copy())# конструктору матрицы можно передавать массив
>>> M.T
matrix([[ 0,  4,  8],
        [ 1,  5,  9],
        [ 2,  6, 10],
        [ 3,  7, 11]])
```

Многие функции, предназначенные для работы с матрицами, расположены в модуле `numpy.matlib`.

```
>>> import numpy.matlib as ml
>>> ml.zeros((3,4))
matrix([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
```

Здесь мы привели только основные процедуры, предназначенные для решения задач линейной алгебры. Имеется еще много других функций и методов, не описанных нами. Вы можете познакомиться с ними самостоятельно по справочной системе.

4. Графические возможности

4.1 Двумерные графики matplotlib

Основные графические возможности «научного» Python реализованы функциями, сосредоточенными в пакте matplotlib, являющегося аналогом графических инструментов MATLAB.

Вначале определите версию вашего пакета matplotlib.

```
import matplotlib as mpl
print ('Текущая версия ', mpl.__version__)
Текущая версия  1.5.1
```

Matplotlib состоит из множества модулей. Основной из них – высокоуровневый модуль matplotlib.pyplot. Наиболее распространенный метод его вызова состоит в выполнении следующей инструкции:

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

Изображение в matplotlib создается путём последовательного вызова команд/функций этого модуля. Графические объекты (точки, линии, фигуры и т.д.) последовательно накладываются один на другой, если они занимают общие области на рисунке.

Объектом самого высокого уровня при работе с matplotlib является рисунок (Figure). На нем располагаются одна или несколько областей рисования (Axes) и элементы оформления рисунка (заголовки, легенда и т.д.). Каждый объект Axes содержит две (или три) координатных оси Axis. Но основное назначение Axes состоит в том, что на него наносится графика: кривые, диаграммы, легенды и так далее.

Создать рисунок figure позволяет инструкция:

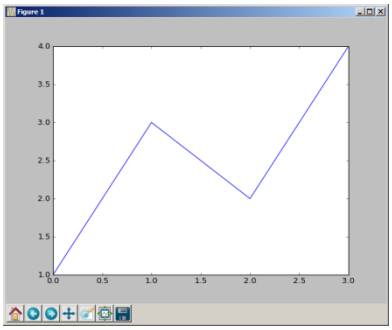
```
plt.figure()
```

Чтобы результат рисования, то есть текущее состояние рисунка, отразилось на экране, можно воспользоваться командой plt.show(). После создания рисунка, т.е. экземпляра класса matplotlib.figure.Figure, результат выполнения всех графических команд будут отображаться именно в нём. Объект Figure может автоматически создаваться при первом выполнении какой – либо графической функции.

Приступим к выполнению примеров. Наберите в Python Shell точно такие команды, и вы получите аналогичный рисунок.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1, 3, 2, 4])
[<matplotlib.lines.Line2D object at 0x00000000072927B8>]
>>> plt.show()
```

После ввода команды plt.show() и нажатия клавиши Enter, открывается окно Matplotlib с графиком.



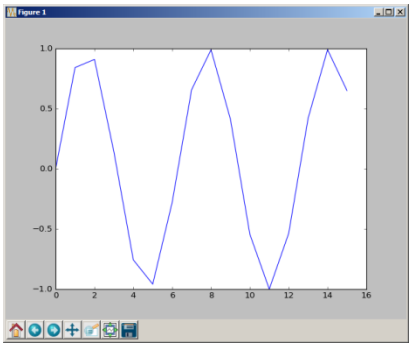
В зависимости от используемого режима отображения, в его командном окне символ приглашения может появиться или не появиться. Если символ приглашения не появился, то вы сможете продолжать вводить команды только после закрытия графического окна.

Функция plot. В приведенной, самой простой форме функции `plot()`, использован один вектор – список `y` – значений узлов ломаной, а в качестве `x` координат использованы целочисленные значения `0,1,2,...`.

В первой команде нашего примера мы импортировали модуль, используемый для построения графиков, и дали ему сокращенное имя `plt`. После этого мы использовали функцию `plot()` этого модуля, которая собственно и строит график. Затем вызвали функцию `show()`, которая его нам показывает. Зачастую, как мы говорили в п.1.2, вызов функции `show()` не требуется.

В другой форме функция `plot(...)` использует два вектора – первый список значений независимой переменной, второй – список значений функции в этих точках.

```
>>> import matplotlib.pyplot as plt
>>> from math import sin
>>> x=range(16)
>>> y = [sin(t) for t in x]
>>> plt.plot(list(x),y)
[<matplotlib.lines.Line2D object at 0x00000000054356D8>]
>>> plt.show()
```

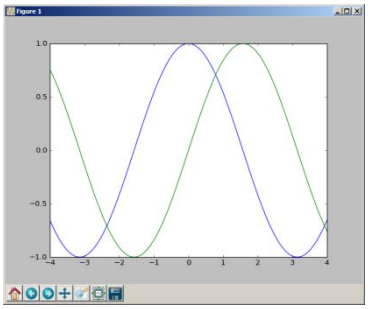


Напомним, что в Python существует специальная синтаксическая конструкция, позволяющая создавать списки, называемая генератором списков. В предыдущем коде она записана строкой `y=[sin(t) for t in x]`. Вся конструкция заключена в квадратные скобки, говорящие, что будет создан список. В начале внутри скобок стоит выражение, которое задает формулу для вычисления элементов списка. Потом идет цикл, в котором указывается имя

переменной цикла и диапазон, который эта переменная пробегает. В приведенном коде после создания списка `y` вызывается функция `plot()`. Чтобы для первого аргумента получить список, мы передали объект `x` в функцию `list()`, хотя в этом примере функцию `list` можно было бы не использовать.

Когда точек много, ломаная неотличима от гладкой кривой

```
>>> import numpy as np
>>> X = np.linspace(-4, 4, 200)
>>> C, S = np.cos(X), np.sin(X)
>>> plt.plot(X, C)
[<matplotlib.lines.Line2D object at 0x0000000003C48F28>]
>>> plt.plot(X, S)
[<matplotlib.lines.Line2D object at 0x0000000003C4E860>]
>>> plt.show()
```



Здесь в графическом окне сначала командой `plt.plot(X,C)` рисуется график косинусоиды, затем, командой `plt.plot(X,S)` – график синусоиды. Графики не видны, пока не выполнится команда `plt.show()` (в консоли Spyder графическое окно появляется сразу после выполнения функции `plot()`).

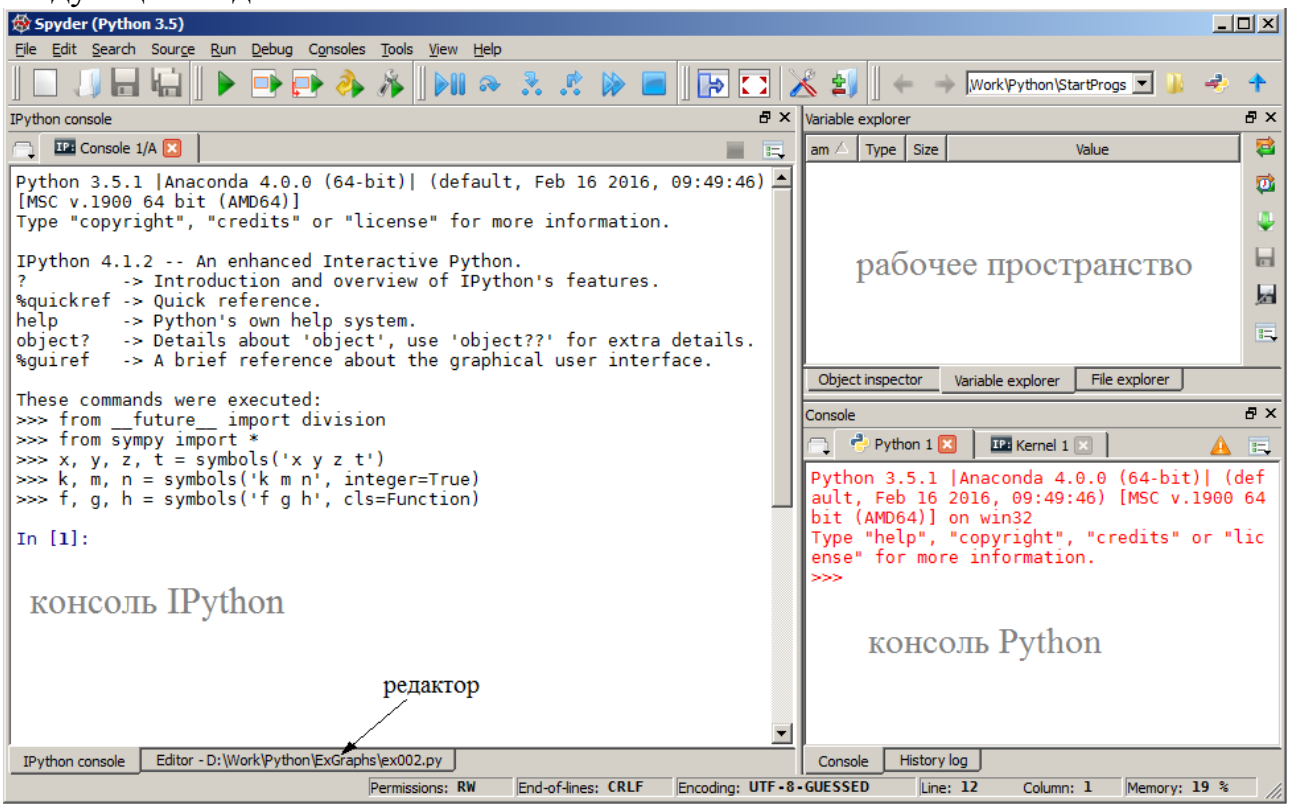
В графическом окне имеется панель управления,

которая состоит из нескольких кнопок. Поясним их назначение, перемещаясь слева направо. Первая кнопка, на которой изображен дом, возвращает просмотр графика к виду, который был создан при открытии окна. Вторая и третья кнопки со стрелками позволяют перемещаться между видами, которые могут иметь, например, разный масштаб, смещение и т.д. Четвертая кнопка (с крестом) имеет два возможных режима. Выбрав эту кнопку, а затем, зажав левую клавишу мыши, можно перемещать график в пределах окна. Нажав на эту кнопку, а затем, зажав правую клавишу мыши, можно менять масштаб изображения. Пятая кнопка позволяет увеличивать или уменьшать выбранную область графического окна. Нажатие шестой кнопки приводит к вызову панели настроек графического окна. Седьмая кнопка позволяет сохранить рисунок в графический файл.

Когда графическое окно активно, то можно использовать горячие клавиши, заменяющие и дополняющие команды кнопок панели. Вот некоторые из них: `h` (home) первая кнопка (дом); `c` или ← вторая кнопка; `v` или → третья кнопка; удерживая `x` (или `y`) можно менять масштаб или перемещать изображение только по горизонтали (или вертикали); `g` – отображение сетки.

Управление внешним видом графика удобно, но не всегда нужно. Например, исполнительная система IPython позволяет отображать графики прямо в своем окне. Во многих случаях это удобнее, чем отображение в отдельнои окне.

Прежде, чем продолжать, договоримся о среде выполнения графических примеров. Пусть это будет Spider. Откройте его. Если вы не меняли настроек, то слева у вас открыто окно редактора кода, а в правой нижней части расположены одна над другой панели двух консолей: Python и IPython. Команды можно выполнять в любой из них. Обычно консоль IPython настроена так, что графики будут отображаться прямо в ней. Захватите мышью заголовок окна IPython и перетащите его, расположив консоль IPython поверх редактора. Затем, если нужно, растяните границы окна консоли. Окно Spider примет следующий вид.



Теперь продолжим обсуждение графических команд библиотеки `matplotlib`, имея в виду, что они вводятся и выполняются в интерпретаторе IPython Spyder. При этом при печати кодов примеров мы не будем использовать метки IPython, поскольку вы можете эти примеры выполнять и в других средах, имеющих другие метки, или не имеющие их вовсе. Кроме того, при отображении графиков мы не будем показывать рамки окна, поскольку окна может и не быть (для встраиваемой графики IPython). Отказ от встраиваемой (и статической) графики полезен в случае построения поверхностей. Трехмерные изображения в отдельном графическом окне можно «вращать» с помощью мыши, в то время как встроенные трехмерные изображения такой способностью пока не обладают.

Напомним, что команда `%matplotlib inline` включает режим отображения графиков внутри окна IPython, а команда `%matplotlib qt`

включает режим отображения графиков в отдельном графическом окне. Выполните следующие команды.

```
%matplotlib inline      # включение режима встроенной графики в IPython
import numpy as np
import matplotlib.pyplot as plt   # загружаем графический модуль
```

Договоримся, что во всех последующих примерах, если не оговорено противное, действуют приведенные выше команды. Это значит, что к функциям модулей `matplotlib.pyplot` и `numpy` можно обращаться через их короткие имена `plt` и `np`.

У функции `plot` имеется большое количество опций, предназначенных для оформления графиков. Можно задавать цвет, стиль линий и маркеры с помощью «форматной» строки следующим образом

```
plot(x, y, 'цвет стиль маркер')
```

(пробелы не обязательны). Цвета определяются символами (по первым буквам английских названий цветов): `c`, `m`, `y`, `r`, `g`, `b`, `w`, `k` (черный). Стиль линии определяются символами: `'-'` (минус) сплошная линия, `'--'` (два минуса) разрывная линия, `':'` (двоеточие) пунктирная линия, `'-.'` (минус точка) штрих пунктирная. Наиболее часто встречаются маркеры: `'s'` – квадрат, `'o'` – круг, `'p'` – пятиугольник, `'d'` – ромб, `'x'` – крест, `'*'` – звезда, `'+'` – плюс, `'h'` – шестиугольник.

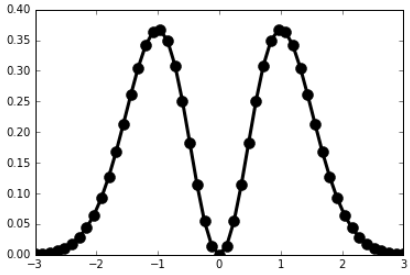
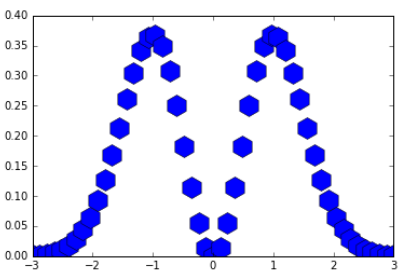
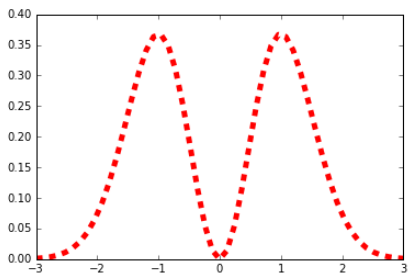
```
x=np.linspace(-3,3,51)
y=x**2*np.exp(-x**2)
plt.plot(x, y, '-ko',linewidth=3, markersize=10) # следующий рисунок слева
```

По следующей команде будет построен график, показанный на следующем рисунке в середине.

```
plt.plot(x,y,'-wh',linewidth=3,markersize=20,markerfacecolor='b')
```

Следующая команда построит рисунок, показанный на следующем рисунке справа.

```
plt.plot(x, y, '--r',linewidth=3)          # следующий рисунок справа
```

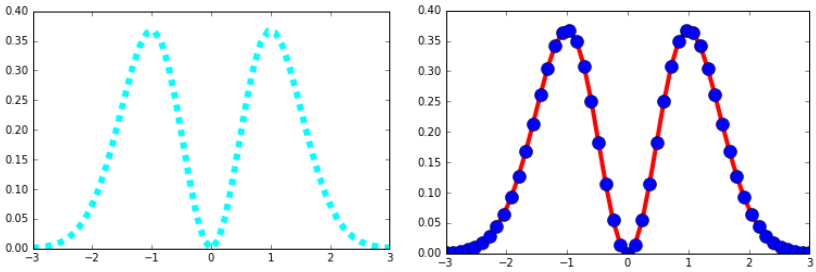




Опции можно задавать, используя их название, которое позволяет легко понять их назначение.

```
plt.plot(x, y, color= 'cyan', linestyle='dashed',linewidth=4) # рис. слева
```

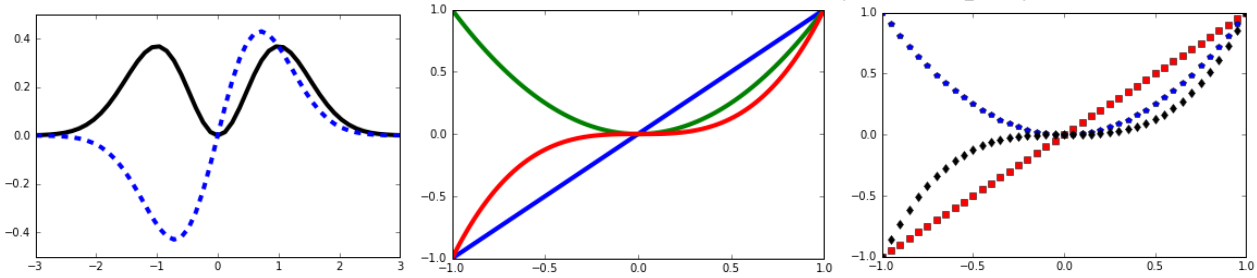
Следующий рисунок справа построен командой

```
plt.plot(x, y, color='red',linewidth=4,  marker='o',
      markerfacecolor='blue',markersize=12) # следующий рисунок справа
```



Можно построить несколько кривых на одном графике. При этом для простой регулировки цветов и типов линий после пары `x, y` координат используется форматная строка.

```
y2=x*np.exp(-x**2)
plt.plot(x, y, '-k',x, y2, 'b--', linewidth=4)    # следующий рисунок слева
```



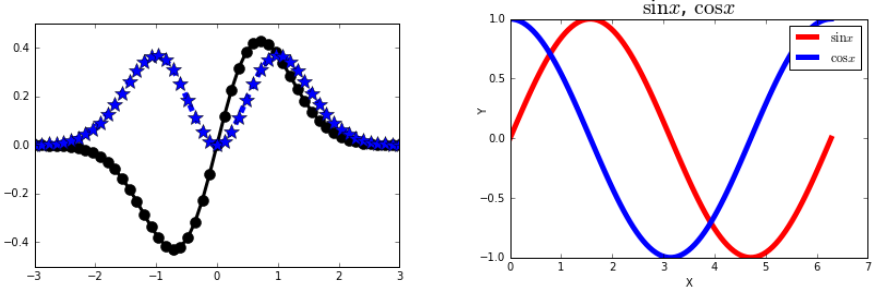
```
x=np.linspace(-1,1,100)
plt.plot(x,x,x,x**2,x,x**3, linewidth=4)        # предыдущий рисунок в центре
```

Если задана опция `linestyle=''` (апостроф+апостроф), то точки не соединяются линиями. Сами точки рисуются маркерами, типы которых должны быть указаны (иначе ничего не будет нарисовано).

```
x=np.linspace(-1,1,41)
plt.plot(x,x,'r-s',x,x**2,'b--p',x,x**3,'kd', linewidth=4,
                    linestyle='')              # предыдущий рисунок справа
```

Если в одной командной ячейке IPython используется несколько инструкций `plot`, то графики рисуются в общей области. Напомним, что в IPython Spyder, если вы желаете набрать несколько однострочных команд в одной ячейке, то после первой строки нужно нажать комбинацию клавиш Ctrl – Enter. Последующие строки можно завершать клавишей Enter, а для завершения всей многострочной инструкции следует два раза нажать Enter (или один раз Shift – Enter)

```
x=np.linspace(-3,3,51)
y1=x*np.exp(-x**2)
y2=x**2*np.exp(-x**2)
plt.plot(x,y1,'-ko',linewidth=3,markersize=10) #кривая 1 на след. рис. слева
plt.plot(x,y2,'--b*',linewidth=5,markersize=14) #кривая 2 на след. рис. слева
```



Для других оформительских целей следует использовать функции модуля `matplotlib.pyplot`, которые должны следовать за функцией `plot` в той же ячейке, и которые влияют на активное графическое окно.

```
x=np.linspace(0,2*np.pi,100)
plt.plot(x, np.sin(x), '-r', linewidth=5, label=r'$\sin x$')
plt.plot(x, np.cos(x), '-b', linewidth=5, label=r'$\cos x$')
plt.legend(fontsize=12)              # legend(loc='upper left')
plt.title(r'$\sin x$, $\cos x$',fontsize=20)   # предыдущий рисунок справа
```

Здесь использована функция `legend()`, которая отображает легенды в графической области активного окна и, кроме того, задает размер шрифта легенд. Функция `title()` создает заголовок графика. Обратите внимание на строки, использованные для задания текста легенды и заголовка. Они могут содержать теги (команды) языка разметки текстов TeX, а символ `'r'`, стоящий впереди, нужен для того, чтобы символы `'\'` (слэш) внутри строки не интерпретировались как спецсимволы.

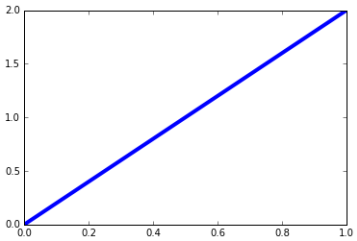
Если вы добавите к предыдущему коду инструкцию

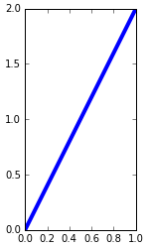
```
savefig("MyFirstGraph.png",dpi=200)
```

то рисунок будет сохранен в соответствующий графический файл. Второй аргумент задает разрешение изображения.

Теперь построим график функции $y = 2 \cdot x$.

```
x=np.linspace(0,1,101)
plt.plot(x,2*x)          # следующий рисунок слева
```





Физический наклон линии отличается от того, как мы себе это представляем из вида функции. Так получается потому, что по умолчанию задается разный масштаб в горизонтальном и вертикальном направлении. Изменить экранное соотношение высоты и ширины графической области можно с помощью команды `set_aspect(число)`. `Число` определяет отношение длин осей графика (длина/высота). Вызов этой функции с единичным аргументом `set_aspect(1)` обеспечивает для нашего примера одинаковый масштаб по осям x и y.

```
x=np.linspace(0,1,101)
plt.plot(x,2*x)
plt.axes().set_aspect(1)          # предыдущий рисунок справа
```

В методе `set_aspect` вместо числа допустимо использовать строки `'auto'` и `'equal'`. Первое соответствует значению по умолчанию, второе – единице.

Вот еще одна ситуация, когда полезно использовать функцию `set_aspect()`. На следующем графике слева синусоида выглядит густовато.

```
x=np.linspace(0,40*np.pi,1000)
plot(x,np.sin(x))          # следующий рисунок слева
```

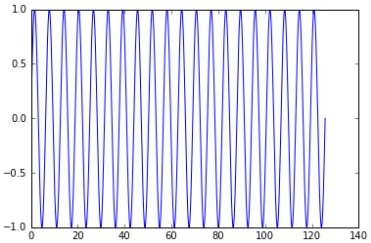
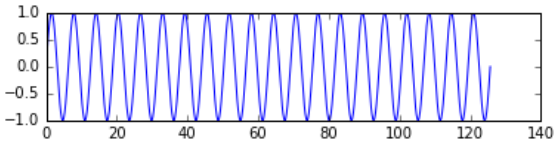



График будет выглядеть лучше, если его растянуть (т.е. увеличить отношение ширина/высота)

```
x=np.linspace(0,40*np.pi,1000)
plot(x,np.sin(x))
axes().set_aspect(15)          # предыдущий рисунок справа
```

Прежде чем продолжить описание других графических возможностей, обсудим инструменты, которые предназначены для оформления графиков. Рассмотрим функции, которые выводят на графиках текст: в заголовках, на осях, печатают аннотации и т.д.

Функция `text(x,y,текст[,...])` выводит в области `Axes` текст. Положение текста определяется в координатах данных или, при задании опции `transform=ax.transAxes`, в относительных координатах $0 \le x \le 1,\ 0 \le y \le 1$ с началом (точка (0, 0)) в левом нижнем углу графической области. Здесь переменная `ax` является ссылкой на соответствующий объект `Axes` рисунка. Ее (ссылку) можно получить командой `ax = fig.gca()` (get current axes), где `fig` является ссылкой на объект рисунка. Опция `fontsize=n` функции `text()` задает размер шрифта, а опции `horizontalalignment` и `verticalalignment` определяют положение текста относительно точки привязки. Опция `rotation` задает угол поворота текста в градусах.

Чтобы написать математическое выражение нужно окружить строку в формате TeX знаками доллара. Текст можно заключать в рамку с цветным фоном, передав аргумент `bbox`, который является словарем. Если вы не знаете TeX, не беда. Можно попросить Python написать TeX код для вашего выражения, используя функцию `latex()` модуля SymPy.

```
>>> from sympy import latex,S  # импортируем функции latex() и S()
>>> latex(S('2*4+10',evaluate=False))
'2 \\cdot 4 + 10'
>>> latex(S('exp(x*2)/2',evaluate=False))
'\\frac{e^{2 x}}{2}'
```

Мы вводим строку, содержащую формулу, в обозначениях Python, а получаем строку, содержащую TeX код этой же формулы. Полученную строку вы можете затем использовать в функциях, отображающих текст в графическом окне.

По умолчанию в matplotlib используются шрифты, которые не поддерживают кириллицу. Но это легко исправить. В Windows шрифты 'Arial', 'Times New Roman', 'Tahoma', 'Courier New' корректно отображают кириллицу. Чтобы их подключить, можно выполнить следующие команды:

```
import matplotlib as mpl
mpl.rcParams['font.family'] = ' fantasy'
```

mpl.rcParams['font. fantasy'] = 'Arial', 'Times New Roman', 'Tahoma'

В результате создается набор шрифтов `fantasy`, который становится основным. Matplotlib будет стараться использовать первый шрифт `Arial`. Если его не окажется, то будет выбран второй по списку и так далее.

Другим условием нормального отображения на рисунках кириллицы является использование UNICODE–строк. В Python 3 все строки являются юникодовыми, но можно также перед строкой поставить префикс `u`, обозначающий, что строка задана в кодировке UNICODE.

```
str = u'Это юникод строка' # unicode-строка
```

Не лишним будет в начале вашей программы вставить строку

```
# -- coding: utf-8 –
```

Функция `matplotlib.pyplot.annotate(...)` добавляет примечание, которое состоит из текста и стрелки, проведенной от текста в некоторую точку на рисунке. Она может иметь следующий формат вызова:

```
annotate(текст, xy=(xa,ya), xytext=(xt, yt), shrink=0.05[,...])
```

Пара чисел `(xa,ya)` представляет координаты указываемой точки, а пара `(xt,yt)` – положение текстовой строки, опция `shrink` задает расстояние от конца стрелки до точки `(xa,ya)`.

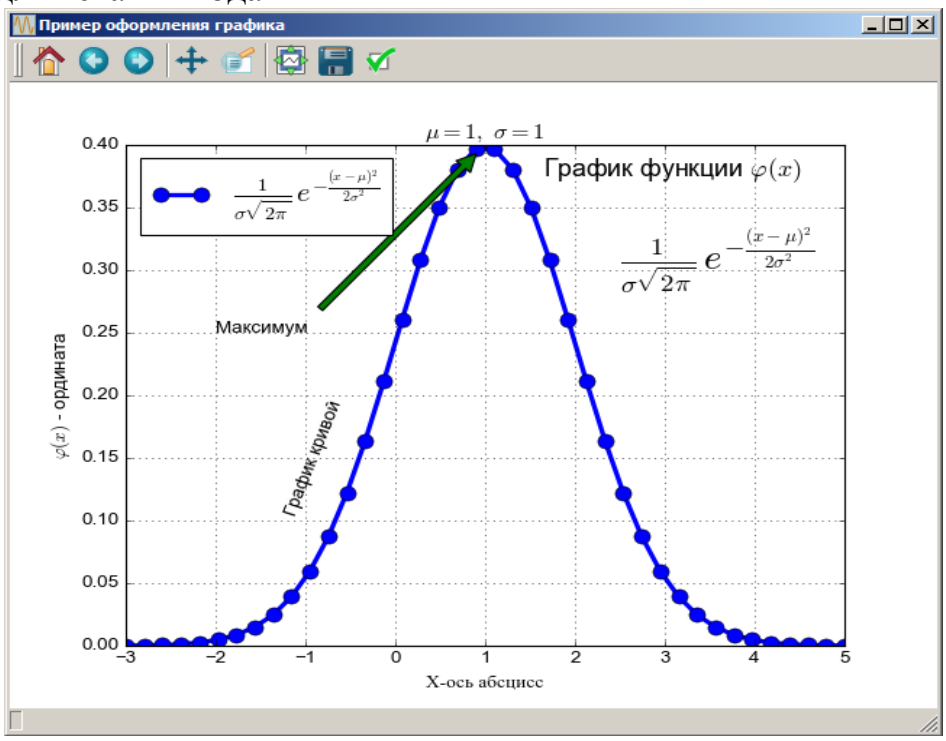
Функции `xlabel()` и `ylabel()` задают подписи к осям. Они имеют одинаковый синтаксис, например,

```
xlabel('текст'[,fontsize='small', verticalalignment='top',
          horizontalalignment='center',...])
```

Пример. Использование различных функций и их опций для оформления рисунка.

```
# -*- coding: utf-8 -*-
%matplotlib qt
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rcParams['font.family'] = 'fantasy'
mpl.rcParams['font.fantasy'] = 'Arial','Times New Roman','Tahoma'

str=r'$\frac{1}{\sigma\sqrt{2\pi}} \,e^{ -\frac{(x- \mu)^2}{2\sigma^2}}$'
x=np.linspace(-3,5,40)
sigma=1
mu=1
y=(1/(sigma*np.sqrt(2*np.pi)))*np.exp(-(x-mu)**2/(2*sigma**2))
fig = plt.figure(facecolor='white',num='Пример оформления графика')
plt.plot(x,y,'-bo',linewidth=3,markersize=10,label=str)
plt.legend(fontsize=18, loc='upper left')
ax = fig.gca()     # получить ссылку на объект axes рисунка fig
plt.title( r'$\mu=1,\ \sigma=1$')
plt.text(0.58, .95,r'График функции $\varphi(x)$',
      horizontalalignment='left',verticalalignment='center',
```

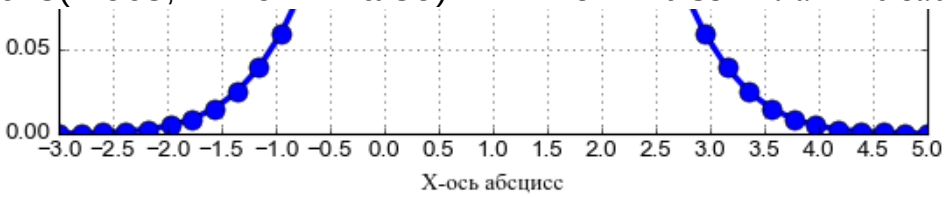
```
        transform=ax.transAxes,
        fontsize=16)  # положение в относительных координатах окна
ax.annotate('Максимум', xy=(1, 0.4), xytext=(-2, 0.25),
            arrowprops=dict(facecolor='green', shrink=0.05))
plt.text(-0.9, 0.15, 'График кривой',
        rotation=70,
        horizontalalignment='center',verticalalignment='center')
plt.text(2.5, 0.3, str, fontsize=24,
        bbox=dict(edgecolor='w', color='cyan'),
        color='black')    # положение в координатах данных
plt.xlabel(u'X-ось абсцисс',{'fontname':'Times New Roman'})
plt.ylabel(r'$\varphi(x)$ - ордината')
plt.grid(true)
# Метка 1 для вставки кода
```



В примере была использована инструкция `ax = fig.gca()` (**g**et **c**urrent **a**xes), которая возвращает ссылку на активный объект `axes` рисунка `fig`. Если объект рисунка создается автоматически при вызове какой–либо графической функции, то доступ к нему (активному графическому окну) можно получить с помощью инструкции `fig=plt.gcf()` (**g**et **c**urrent **f**igure).

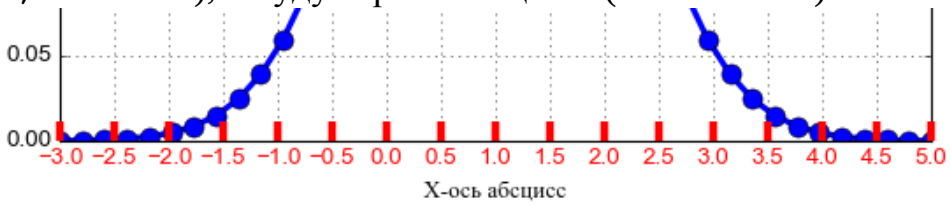
Координатная сетка на рисунке появилась в результате выполнения команды `plt.grid(True)`. Эта сетка связана с делениями координатных осей (ticks). В matplotlib существуют главные деления/засечки (major ticks) и вспомогательные (minor ticks) для каждой координатной оси. По умолчанию рисуются только главные деления и связанные с ними линии сетки. Узнать положение основных засечек можно следующим образом:

```
xax = ax.xaxis
xlocs = xax.get_ticklocs()
xlocs
```

```
array([-3., -2., -1.,  0.,  1.,  2.,  3.,  4.,  5.])
```
Как видите, положение засечек является массивом. Его можно изменить следующими командами:
```
xlocs =np.linspace(-3,5,17)
ax.set_xticks(xlocs, minor = False)   # minor = False – главные засечки
```


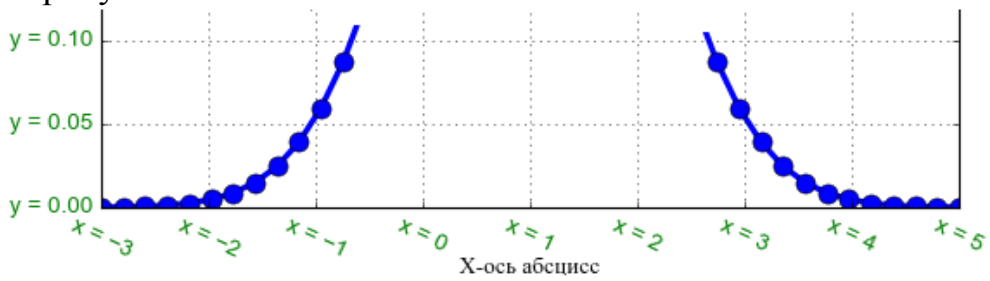
Для экономии места мы привели только нижнюю часть рисунка, на котором изменилось количество засечек на оси X и густота вертикальных линий координатной сетки.

Для самой сетки можно задать опции цвета, стиля, толщины и т.д. Например, следующая инструкция задает опции сетки для первой (и, как правило, единственной) графической области.
```
ax[0].grid(color='b',alpha=0.5,linestyle='dashed',linewidth=0.5)
```
Оформление засечек на оси можно выполнить следующей командой.
```
ax.tick_params(axis='x', which='major',
               labelleft='off', labelright='off', labeltop='on', labelbottom='on',
               direction='in', length=10, width=4, colors='r')
```
Здесь мы говорим, что будем оформлять главные засечки координатной оси X (`axis='x',which='major'`), расположенные на верхней и нижней стороне области рисунка (`labeltop='on',labelbottom='on'`). Засечки будут направлены внутрь рисунка (`direction='in'`), иметь длину 10 и толщину 4 (`length=10, width=4`), и будут красного цвета (`colors='r'`).



Можно поменять подписи у меток, стоящих возле засечек. В коде примера после строки «`# Метка 1 для вставки`» добавьте следующий код:
```
xax = ax.xaxis          # ссылка на объект горизонтальной координатной оси
xlabels = xax.get_ticklabels()          # итератор меток (не список) на оси X
for lbl in xlabels:
    lbl.set_text('x = '+ lbl.get_text())    # меняем текст каждой метки
# меняем список меток  на оси X
ax.set_xticklabels(xlabels, color='green', rotation=335)

yax = ax.yaxis          # ссылка на объект вертикальной координатной оси
ylabels = yax.get_ticklabels()          # итератор меток
for lbl in ylabels:
    lbl.set_text('y = '+ lbl.get_text())    # меняем текст каждой метки
# меняем список меток  на оси Y
ax.set_yticklabels(ylabels, color='green')
```

В результате его выполнения подписи к засечкам примут вид, показанный на следующем рисунке.

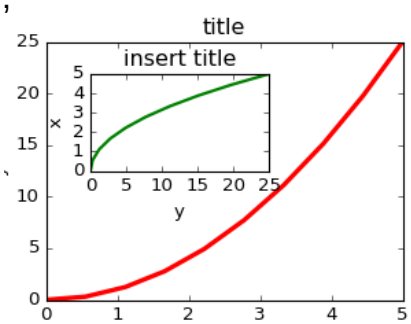


Для экономии места мы привели только нижнюю часть рисунка.

Команды `xax=ax.get_xaxis()` и `yax=ax.yaxis` предоставляют доступ к объекту `axis` горизонтальной координатной оси. Через него можно получить доступ ко всем методам и атрибутам, управляющим внешним видом оси без использования функции `ax.tick_params()`.

Объектов `axes` на графике можно создать несколько, и каждый из них будет содержать свой график. В следующем коде мы рисуем график функции $y = x^2$, как основную кривую, и график обратной функции $y = \sqrt{x}$, как вспомогательный.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 5, 10)
y = x**2
fig = plt.figure(facecolor='white')
ax1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # основной объект axes
ax2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # внутренний объект axes
# Главный график
ax1.plot(x, y, 'r',linewidth=3)
ax1.set_title('title')
# внутренний график
ax2.plot(y, x, 'g',linewidth=2)
ax2.set_xlabel('y')
ax2.set_ylabel('x')
ax2.set_title('inner title');
```



Здесь при создании объектов `ax1` и `ax2` методами `fig.add_axes()` использовались относительные координаты и размеры прямоугольных областей, которые представляют эти объекты.

■

Цвет графического элемента задается при его создании каким–либо аргументом (по умолчанию или явно). Его имя (название опции) может быть различным: color (или colors), facecolor (или facecolors), backgroundcolor и т.д. У многих графических объектов имеются методы, которые устанавливают цвет его элементов, например, set_color(цвет). Однако все они используют одинаковые способы задания цвета. Для основных цветов можно использовать три формата задания: длинное или короткое имя, заключенное в кавычки, а также числовой триплет. Так черный цвет можно задать строками 'black', 'k' или кортежем (0,0,0). Создать графическое окно с белым фоном можно любой из следующих инструкций:

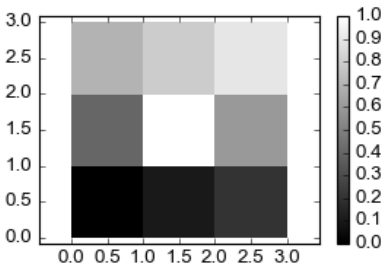
```
fig = plt.figure(facecolor='white')
fig = plt.figure(facecolor='w')
fig = plt.figure(facecolor=(1,1,1))
```

Для обозначения сложных цветов используются кортежи из трех дробных чисел из интервала [0, 1]. Например, (0.5,0.5,0.5) – серый цвет, (0.5,0,0) – темно–красный, а (0.49,1.0,.0.83) – аквамариновый. Первое число задает относительную интенсивность красной составляющей цвета, второе – зеленой, третье – синей. Получить массив относительных интенсивностей из абсолютных можно следующим образом:

```
rgb = np.array([204,255,51])/255
```

Яркость серых цветов можно задать с помощью строки, содержащей число в интервале от 0 до 1 (1 – белый, 0 – черный), например, color='0.75'. Эти числовые значения можно использовать вместе с цветовой картой 'gray', в которой нулю отвечает черный цвет, единице – белый, а промежуточным числам – различные оттенки серого (от темно-серого до светло-серого). В следующем примере функция plt.pcolor() строит псевдоцветное изображение двумерного массива, используя заданную нами цветовую карту. В таких картах (палитрах) одному числовому значению отвечает некоторый цвет. Список всех палитр можно получить командой plt.cm.datad. Задание конкретной палитры, используемой графическим объектом, можно выполнить командой объект.set_cmap('название_палитры') или с помощью подходящей опции графической функции. В нашем примере используются только оттенки серого цвета, содержащиеся в палитре 'gray'.

```
A=np.array([[0,0.1,0.2],[0.4,1,0.6],[0.7,0.8,0.9]],dtype=float)
im=plt.pcolor(A)        # изображение массива
im.set_cmap('gray')  # выбор цветовой карты изображения
fig=plt.gcf()
fig.set_facecolor('w')
ax=fig.gca()
ax.axis('equal')        # одинаковый масштаб по осм X и Y
fig.colorbar(im)        # создание палитры цветов справа
```

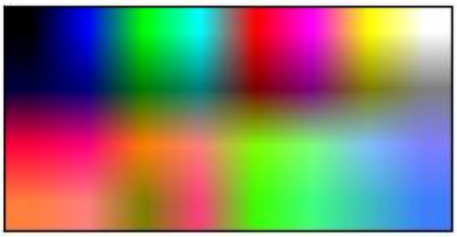


Обратите внимание на то, что квадрат отвечающий элементу массива A[0][0], расположен в левом нижнем углу изображения. Инструкция `fig.colorbar(im)` нарисовала справа палитру цветов, показывающую связь между числовым значением и цветом.

Заметим, что для изображения массива аналогичным способом можно использовать функции `plt.pcolormesh(A)` и `plt.matshow(A)`.

В следующем примере каждый цвет задается списком из трех чисел, представляющих относительные интенсивности основных цветов.

```
B=np.array([[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]],
[[0,0,0.25],[0,0,0.5],[0,0.5,0],[0,0.5,0.5],[0.5,0,0],[0.5,0,0.5],
                                                [0.5,0.5,0],[0.5,0.5,0.5]],
[[1,0,0.25],[1,0,0.5],[1,0.5,0],[1,0.5,0.5],[0.5,1,0],[0.5,1,0.5],
                                                [0.5,0.75,1],[0.5,0.5,1]],
[[1,0.5,0.25],[1,0.5,0.5],[0.5,0.5,0],[1,0.25,0.5],[0.25,1,0],
                                    [0.25,1,0.5],[0.25,0.75,0.75],[0.25,0.5,1]]])
im=plt.imshow(B, interpolation='nearest')  # следующий рисунок слева
fig=plt.gcf()
fig.set_facecolor('w')
ax=fig.gca()
ax.set_xticks([])       # удаляем засечки на оси X
ax.set_yticks([])       # удаляем засечки на оси Y
```



Основные цвета записаны в элементе `B[0]` матрицы и на рисунке слева представлены в верхней строке.

Массив A может иметь размеры `MxN`, `MxNx3` или `MxNx4`. Здесь функция `plt.imshow(B, interpolation='nearest')` используется для представления «двумерного» массива `B`, элементами которого являются трехэлементные вектора цветов. Опция `interpolation='nearest'` необходима для того, чтобы функция не создавала плавного перехода цветов между соседними квадратами. Если эту опцию удалить, то вы получите изображение, показанное на предыдущем рисунке справа.

Обычно функция `plt.imshow(A[,...])` используется для отображения массивов цветов, прочитанных из графических файлов функцией `plt.imread(имя_файла)`. Например,

```
import matplotlib.cbook as cbook
image_file = cbook.get_sample_data(
      r'D:\Work\Python\StartProgs\Graphics\PogorelovDesk01.png')
image = plt.imread(image_file)
plt.imshow(image)
```

Того же результата можно добиться следующим кодом:

```
from scipy.ndimage import imread
im = imread(
       r'D:\Work\Python\StartProgs\Graphics\PogorelovDesk01.png')
plt.imshow(im)
```

Цвет можно задавать RGB строкой, содержащей шестнадцатеричное число, перед которым стоит символ `'#'` (решетка). Например, `color='#E0A9F3'`. Первые две цифры задают абсолютную яркость красной составляющей (от `0` до `255=0xff`), вторые – зеленой, третьи – синей.

Пример. Нарисуем таблицу, состоящую из квадратов, цвет заливки которых задается шестнадцатеричной строкой.

```
%matplotlib qt
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

fig, ax = plt.subplots()
fig.set_facecolor('white')

n=3
rg=range(n)
for r in rg:
   cr=r*127
   for g in rg:
      cg=g*127
      for b in rg:
         cb=b*127
         clr='#'+'{0:02x}'.format(cr)+\
                 '{0:02x}'.format(cg)+'{0:02x}'.format(cb)
         x=n*r+g
         y=b
         rect=Rectangle((x, y),1, 1,facecolor=clr)
         ax.add_patch(rect)
         ax.text(x+0.5,y+0.5,clr,color='w',\
                 verticalalignment='center',horizontalalignment='center')
```

```
ax.set_xlim(0, 9)
ax.set_ylim(0, 3)
ax.set_aspect('equal')
```

В этом примере в тройном цикле составляющие основных цветов `(cr,cg,cb)` принимают по 3 значения: 0, 127, 254. Из них компонуется строка цвета `clr`. Она получается конкатенацией 4–х строк. Первая строка состоит из одного символа `'#'`. Остальные три являются форматированными строками вида`'{0:02x}'.format(cr)`. Здесь десятичное число `cr` преобразуется к 16-тиричному виду (на это указывает символ `'x'`). Под него выделяется две позиции (двойка перед `'x'`), и символом заполнения свободных позиций является `0`. Таким образом, строка `'02x'` означает вывод двухсимвольного шестнадцатеричного числа с ведущим нулем, если он потребуется. Начальный ноль, стоящий перед двоеточием `{0:...}`, является номером аргумента метода `format(...)`, который будет выводиться (у нас других аргументов нет). Созданная строка цвета `clr` используется при задании цвета фона квадрата (`facecolor=clr`), а также печатается белым цветом в его центре.

Координата `y` левой нижней вершины каждого квадрата соответствует номеру индекса `b` в списке значений синей составляющей цвета [0, 127, 254]. Координата `x` конструируется по формуле `x=n*r+g` (у нас `n=3`), где `r` и `g` – индексы в списке значений красной и зеленой составляющей цвета. В результате красная составляющая цвета левых `3x3` квадратов равна нулю, средних `3x3` квадратов равна 127, а правых `3x3` квадратов равна `254` (т.е. `0xfe`).

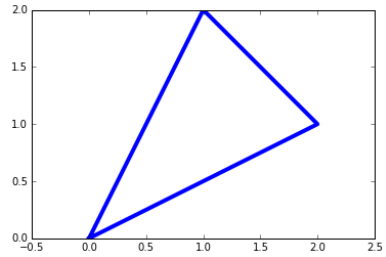
■

Набор цветов можно разнообразить за счёт различной степени прозрачности, которая обычно задается числовым параметром `alpha` (от полностью прозрачного – 0, до непрозрачного – 1). Однако эта возможность чаще используется в трехмерном случае, где мы ее и рассмотрим.

Вернемся к обсуждению функций, используемых при построении двумерных графиков.

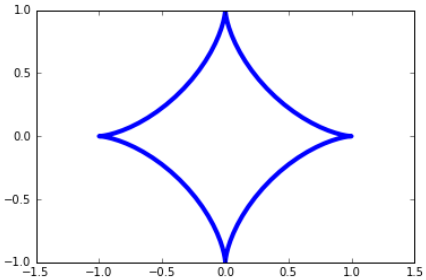
Параметрические графики. Последовательность `x` координат точек у функции `plot()` не обязана быть монотонно возрастающей.

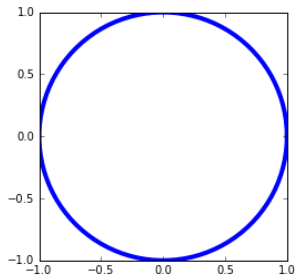
```
plot([0,1,2,0],[0,2,1,0],linewidth=4);
axis('equal');     # задание одинаковых масштабов по осям
```



Это позволяет строить кривые по их параметрическому уравнению.

```
t=np.linspace(0,2*np.pi,100)
x=np.sin(t)**3
y=np.cos(t)**3          # астроида
plot(x,y,linewidth=4);
axis('equal');  # следующий рисунок слева
```



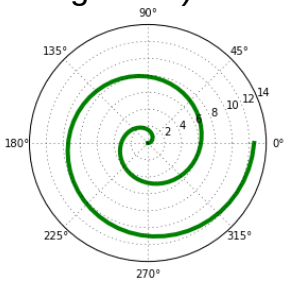


В следующем примере использование функции `axes().set_aspect(1)` позволяет построить окружность, которая выглядит как окружности, а не как эллипс.

```
t=np.linspace(0,2*np.pi,100)
plot(np.cos(t),np.sin(t),linewidth=4)
axes().set_aspect(1)        # предыдущий рисунок справа
```

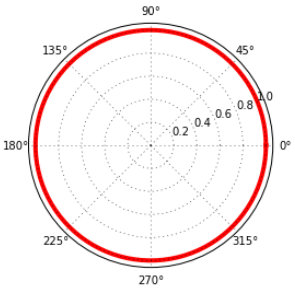
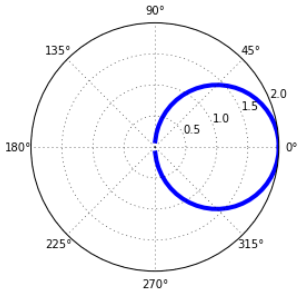
График функции в полярных координатах строится с помощью функции `plt.polar(angle, radius)`. У нее первый аргумент представляет массив углов, второй – радиусов. Параметры и функции оформления такие же, как у функции `plot()`.

```
t=np.linspace(0,4*np.pi,100)
plt.polar(t,t, linewidth=4, color='green')
```



Функция `plt.polar()` автоматически выравнивает вертикальный и горизонтальный масштабы. В следующих примерах окружность похожа сама на себя, а не на овал.

```
t=np.linspace(0,2*np.pi,100)
r = [1 for a in t]
polar(t,r,linewidth=4,color='red')        # следующий рисунок слева
```

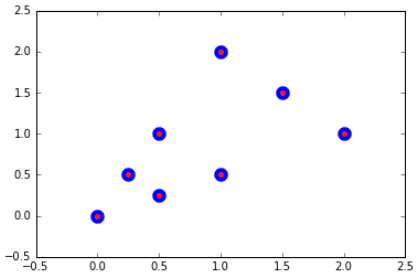
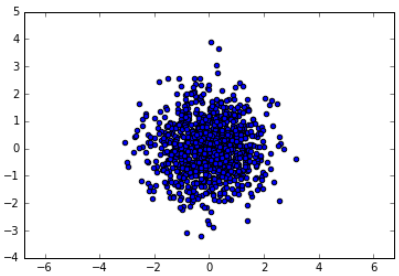



Сдвинутая окружность тоже выглядит как окружность.

```
t=np.linspace(0,2*np.pi,100)
polar(t,2*np.cos(t),linewidth=4,color='blue') # предыдущий рисунок справа
```

Функция `plt.scatter()` рисует точки (график рассеяния). Ей надо передать массивы `x` и `y` координат этих точек.

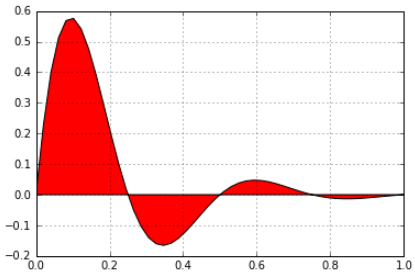
```
plt.scatter([0,1,2,1.5,0.5,0.25,0.5,1],[0,2,1,1.5,1,0.5,0.25,0.5],
        linewidths=8,c='red',edgecolors='blue'); # следующий рисунок слева
```

```
n = 1024
X = np.random.normal(0,1,n)
Y = np.random.normal(0,1,n)
plt.scatter(X,Y)          # предыдущий рисунок справа
plt.axis('equal');
```

Области между кривыми. Для построения графика с залитой областью между кривой и горизонтальной осью можно использовать функцию `plt.fill()`.

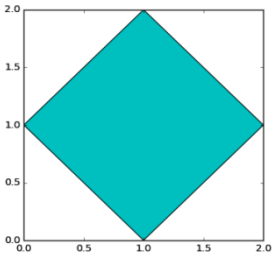
```
x = np.linspace(0, 1)
y = np.sin(4 *np.pi*x)*np.exp(-5*x)
plt.fill(x, y, 'r')
plt.grid(True)
```



Функция `fill` предназначена для заливки замкнутых многоугольников. Поэтому она соединяет последнюю точку с первой (полагает, что массивы координат `x,y` задают замкнутый многоугольник), но это не всегда совпадает с осью `Ox`.

```
x=np.array([0,1,2,1])
y=np.array([1,2,1,0])
plt.fill(x, y, 'c')
```

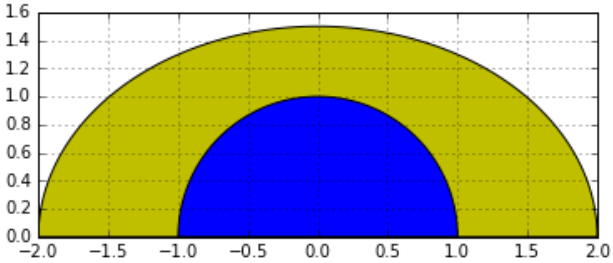
```
plt.show()
```



Использование двух функций при построении залитых графиков допустимо.

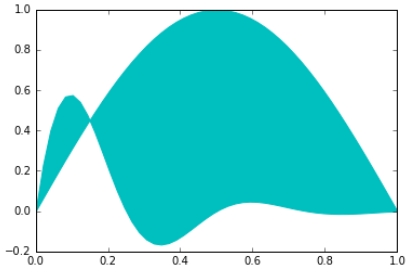
```
t = np.linspace(0, np.pi,100)
x1=np.cos(t)
y1=np.sin(t)
x2=2*np.cos(t)
y2=1.5*np.sin(t)
plt.fill(x2, y2,'y',x1,y1, 'b')
plt.grid(True)
plt.axes().set_aspect('equal')
```

Однако, если здесь использовать массивы в другом порядке `fill(x1,y1,'r',x2,y2,'b')`, то вы увидите только бо'льшую область, поскольку меньшая будет полностью закрыта бо̀льшей.



Функция `plt.fill()` старается залить область между кривой и осью X. Чтобы залить область между двумя кривыми нужно использовать функцию `fill_between(x,y1,y2,опции)`, где `y1` и `y2` являются массивами точек ограничивающих кривых.

```
x = np.linspace(0, 1)
y1 = np.sin(4 *np.pi*x)*np.exp(-5*x)
y2=np.sin(x*np.pi)
plt.fill_between(x,y1,y2,color='c')
```



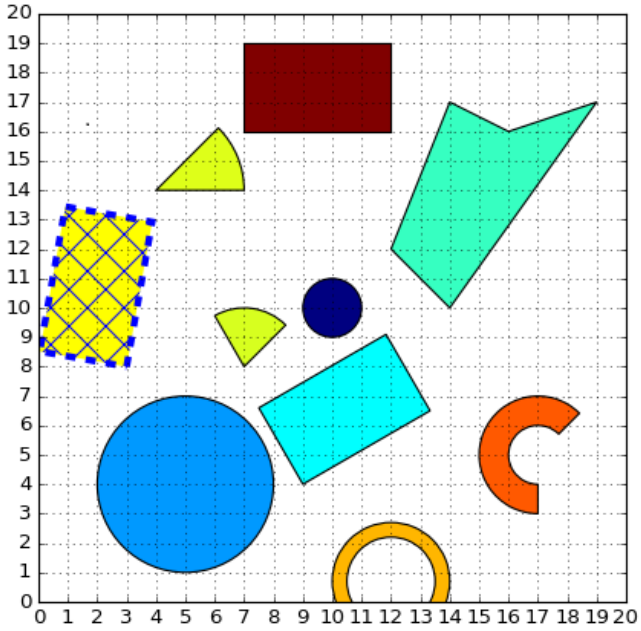
Графические примитивы. На двумерные рисунки можно добавлять множество графических примитивов: ломаных, прямоугольников, многоугольников, кругов, секторов и т.д. Часто их называют «патчами» (patch – пятно на рисунке). Мы будем использовать термины «графический примитив»,

а также графический фрагмент или фигура. В следующем примере демонстрируются способы рисования некоторых фигур.

Пример. Рисование графических примитивов.

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib
from matplotlib.patches import Circle, Wedge, Polygon, Rectangle
from matplotlib.collections import PatchCollection
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
fig.set_facecolor('white')
patches = [ ]

circle = Circle((5, 4), 3)    # круг: центр, радиус
patches.append(circle)      # добавление круга в набор фигур

wedge = Wedge((7, 8),      # сектор: центр
              2,           # радиус
              45,          # начальный угол
              120)         # конечный угол
patches.append(wedge)     # добавление сектора в набор фигур

# Создание и добавление нескольких фигур
patches += [
   Wedge((10, 10), 1, 0, 360),                # круг
   Wedge((12, .7), 2, 0, 360, width=0.5),   # кольцо
   Wedge((4, 14), 3, 0, 45),                   # сектор
   Wedge((17, 5), 2, 45, 270, width=1)]     # кольцевой сектор
# многоугольник
polygon = Polygon([[12,12],[14,17],[16,16],[19,17],[14,10]])
patches.append(polygon)  # добавление многоугольника в набор фигур
# прямоугольники
rect1=Rectangle((7, 16),   # координаты вершины
              5,           # ширина прямоугольника
              3)           # высота прямоугольника
patches.append(rect1)      # добавление прямоугольника в набор фигур

rect2=Rectangle((9, 4),5, 3,
      angle=30,            # угол поворота в градусах
      facecolor='cyan'     # цвет фона
      #fill=False)         # нет фона
ax.add_patch(rect2)        # добавление прямоугольника сразу на рисунок

rect3=Rectangle((3, 8),5, 3,angle=80,
      facecolor='yellow',  # цвет заливки
      hatch='x',           # шриховка, допустимо: '/','//',+','-','o','0','.','*'
```

```
        edgecolor="#0000ff", # цвет контура
        linewidth=4,         # толщина контура
        linestyle='dashed')  # допустимо: 'solid', 'dashed','dashdot','dotted'
ax.add_patch(rect3) # добавление прямоугольника в графическую область
# создание коллекции фигур
p=PatchCollection(patches,cmap=matplotlib.cm.jet)
# задание цвета в коллекции фигур
colors = 80*np.random.rand(len(patches))
p.set_array(np.array(colors))
ax.add_collection(p) # добавление коллекции фигур на рисунок
# оформление рисунка: пределы, пропорции, засечки и координатная сетка
ax.set_xlim(0, 20)
ax.set_ylim(0, 20)
ax.set_aspect('equal')
xlocs =np.linspace(0,20,21)
ax.set_xticks(xlocs, minor = False)   # положение главных засечек по оси X
ax.set_yticks(xlocs, minor = False)   # положение главных засечек по оси Y
ax.grid(True)
```



Для создания графического примитива нужно выполнить три шага:

- создать «математический» примитив, задав его геометрические характеристики;
- преобразовать «математический» примитив в графический, задав его цвет, стиль, толщину линий и т.д.;
- перенести графический примитив на рисунок, используя функцию `add_patch()`.

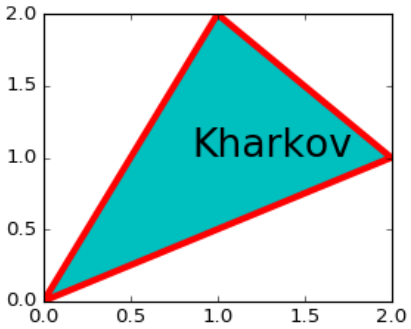
В нашем примере эта последовательность действий была выполнена двумя разными способами. Вначале из модуля `matplotlib.patches` мы импортировали функции `Circle`, `Wedge`, `Polygon`, `Rectangle`, которые создают примитивы: круг, сектор, многоугольник и прямоугольник. Затем мы создали список patches. Элементы списка могут иметь любой тип, и в нашем

случае они являются объектами примитивов. Мы добавили все объекты в список patches, который с помощью функции `PatchCollection(...)` преобразовали в коллекцию графических примитивов. Затем, используя инструкцию `ax.add_collection(...)`, мы добавили коллекцию фигур на рисунок. Второй способ состоял в создании одного примитива, например, прямоугольника `rect2=Rectangle(...)`, который сразу с помощью инструкции `ax.add_patch(rect2)` переносился на рисунок.

Шаги, описанные выше, иногда объединяются. Например, графический примитив – прямоугольник `rect2` (первые два шага) был создан сразу функцией `Rectangle(...)`. Кроме того, все три шага иногда можно записать одной строкой. Например, чтобы нарисовать стрелку от точки (0,0) до точки (1, 1) можно выполнить инструкцию `gca().add_patch(Arrow(0,0,1,1))`.

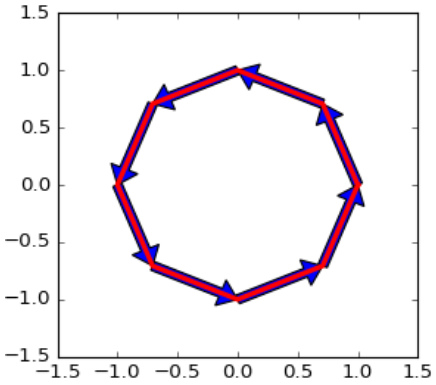
В следующем примере мы демонстрируем способ работы с графическим примитивом `Path` («путь»), который является ломаной. Если ломаная замкнутая, то ее внутренняя область может быть залита цветом, и тогда `Path` будет представлять многоугольник.

```
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches
fig =plt.figure(facecolor='white')
ax = fig.add_subplot(111)
coords=[(0,0),(2,1),(1,2),(1,2)]
linecmds=[Path.MOVETO,Path.LINETO,Path.LINETO,
                                        Path.CLOSEPOLY]
path=Path(coords,linecmds)  # создаем путь
# создаем графический примитив закрашенный треугольник
patch=patches.PathPatch(path,lw=4,facecolor='c',edgecolor='r')
# нет заливки
#patch=patches.PathPatch(path,lw=4,facecolor='none',edgecolor='r')
#patch=patches.PathPatch(path,lw=4,fill=False,edgecolor='r')
ax.add_patch(patch)  # добавляем фигуру в графическую область рисунка
ax.text(0.85,1,'Kharkov',fontsize=24)        # пишем текст
ax.set_xlim(0,2)
ax.set_ylim(0,2)
```



В следующем примере мы иллюстрируем работу с графическим примитивом `Arrow`.

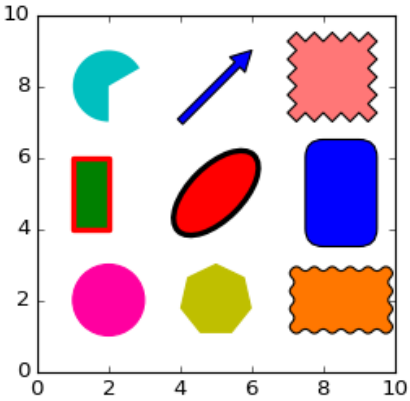
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
# стрелки по окружности
t=np.linspace(0,2*np.pi,9)
x=np.cos(t)
y=np.sin(t)
fig =plt.figure(facecolor='white')
ax = fig.add_subplot(111)
arrows=[(x0,y0,dx,dy) for (x0,y0,dx,dy) in zip(x,y,np.diff(x),np.diff(y))]
for x0,y0,dx,dy in arrows:
    ax.add_patch(patches.Arrow(x0,y0,dx,dy,width=0.4))
ax.plot(x,y,'r',linewidth=2)
```



Приведем еще пример использования графических примитивов.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.collections import PatchCollection

fig, ax = plt.subplots()
fig.set_facecolor('white')
ptchs = [ ]
ptchs.append(patches.Circle((2,2), 1,color='#ff00A0'))
ptchs.append(patches.Rectangle((1,4), 1, 2, facecolor='g',
                                edgecolor='r',linewidth=3))
ptchs.append(patches.Wedge((2,8), 1, 30, 270, ec="none",
                                facecolor='c'))  # сектор
# RegularPolygon( (xc,yc), numVertices, radius=5,...) Правильный многоугольник
ptchs.append(patches.RegularPolygon((5,2), 7, 1,color='y'))
# Ellipse((xc,yc), width, height, angle=0.0,...)
ptchs.append(patches.Ellipse((5,5), 3, 1.5,45,facecolor='r',
                                edgecolor='k',linewidth=3))
# Arrow( xstart, ystart, dx, dy, width=1.0,...)
ptchs.append(patches.Arrow(4, 7, 2, 2, width=1))
# FancyBboxPatch((xleft,ybottom), width, height, boxstyle='round',...)
ptchs.append(patches.FancyBboxPatch((8,4),1,2,
                                boxstyle=('round,pad=0.5')))
ptchs.append(patches.FancyBboxPatch((7.5,1.5),2,1,
```

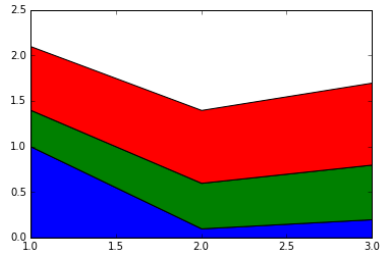
```
                facecolor='#ff7700',
                boxstyle=patches.BoxStyle("roundtooth", pad=0.5)))
ptchs.append(patches.FancyBboxPatch((7.5,7.5),1.5,1.5,
                facecolor='#ff7777',
                boxstyle=patches.BoxStyle("sawtooth", pad=0.5)))
collection = PatchCollection(ptchs, match_original=True)
ax.add_collection(collection)
ax.axis('image');
ax.set_xlim(0, 10);
ax.set_ylim(0, 10);
```



Формат функций, создающих большинство графических примитивов, очевиден. Скажем несколько слов о примитиве `FancyBboxPatch`. Он представляет собой прямоугольник, вокруг которого рисуется фигурная область (полоса). Размеры прямоугольника передаются аргументами функции. Тип и размеры полосы определяются опцией `boxstyle`, которая может принимать строковое значение, либо объект класса `patches.BoxStyle`.

Комбинированный (многоярусный, пакетный, этажерочный, штабельный) график (stackplot) набора функций $f_1(x), f_2(x), f_3(x), ..., f_n(x)$ состоит из кривых построенных таким образом, что точки `i+1` – ой кривой находятся выше `i` – ой на величину $f_i(x)$. Т.о. уравнение `k` – ой кривой имеет вид $y_k(x)=\sum_{i=1}^{k} f_i(x)$. Функция `plt.stackplot(X,A[,...])` строит такой график. Она принимает одномерный массив X горизонтальных координат, и двумерный массив A, строки которого используются при построении графика. Каждая строка массива A содержит `y` координаты узлов ломаных f_i, которые используются для построения пакетного графика. Количество столбцов массива A должно совпадать с количеством элементов массива X.

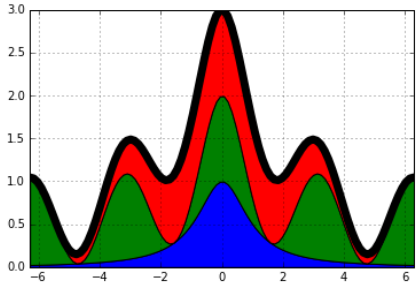
```
aa=np.array([1,2,3])
A=np.array([[1,0.1,0.2],[0.4,0.5,0.6],[0.7,0.8,0.9]],dtype=float)
plt.stackplot(aa,A)
```



Вторая строка массива A интерпретируется как добавки к значениям первого массива, третья – как добавки к результату суммирования двух первых строк и так далее.

Вместо двумерного массива A можно использовать любое количество одномерных массивов такой же длины как и массив X.

```
x=np.linspace(-2*np.pi,2*np.pi,100)
y1=1/(1+x**2)
y2=np.cos(x)**2
y3=np.exp(-x**2/10)
z=y1+y2+y3
plt.stackplot(x,y1,y2,y3)
plt.plot(x,z,'k',linewidth=4)
plt.xlim(-2*np.pi, 2*np.pi)
plt.grid(True)
```



Здесь для сравнения мы еще построили график кривой `z=y1+y2+y3`, показанный на рисунке черным цветом.

Контурные графики. Функции `plt.contour()` и `plt.contourf()` рисуют контурные графики функции двух переменных. Первая – рисует линии постоянного значения, а вторая – залитые области между линиями постоянного значения. Начнем с примера построения линий уровня функции $f(x,y)=x^4-2x^2+y^4-2y^2+1$. Вначале создадим массивы.

```
import numpy as np
import matplotlib.pyplot as plt
def f(x,y):
    return x**4-2*x**2+y**4-2*y**2+1

n=101
x = np.linspace(-2, 2, n)
y = np.linspace(-2, 2, n)
X, Y = np.meshgrid(x, y)
Z=f(X,Y)
```

В формате `contour(Z)` функция строит контурный график массива Z. Значения линий уровня выбираются автоматически.

```
fig = plt.figure(facecolor='white')
ax1 = fig.add_subplot(1,2,1)
ax1.contour(Z)        # линии уровня
ax2 = fig.add_subplot(1,2,2)
ax2.contourf(Z)       # залитые области между линиями уровня
```

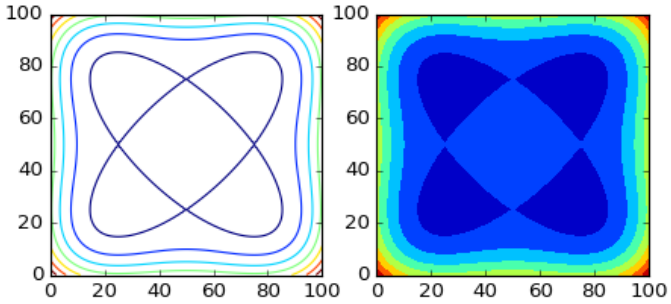
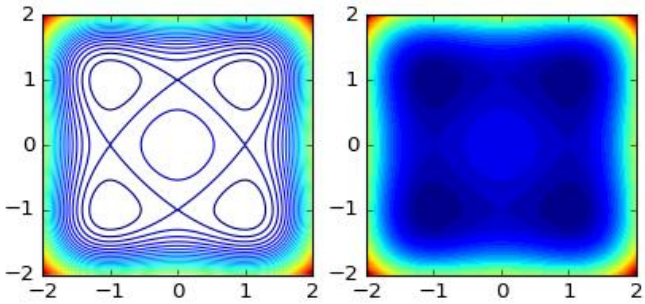


Рисунок (объект Figure) в matplotlib представляет графическое окно (автономное или внедренное в документ). Внутри окна (рисунка) может быть создано несколько графических областей (subplots), в каждой из которых может быть построен свой график. В нашем примере команда `fig.add_subplot(1,2,1)` создает графическое окно и определяет, на какое количество графических областей оно будет разбито. В качестве первых двух аргументов функция принимает количество строк (`numrows`) и количество колонок (`numcols`), которые образуют графические области. Третьим аргументом указывается номер активной области, в которую будет направляться графический вывод. Если `numrows×numcols<10`, то запятые между этими аргументами ставить необязательно. Например, последнюю инструкцию можно записать в виде `fig.add_subplot(121)`.

В первой графической области мы строим график линий уровня, а во второй – залитые различным цветом области между линиями уровня.

Обратите внимание на отметки координатных осей. По осям X и Y отложены индексы элементов в массиве Z. Чтобы по осям были отложены координаты X и Y, следует использовать формат `contour(X,Y,Z)`. Количество линий уровня на графике можно задать четвертым аргументом `contour(X,Y,Z,N)`. Массивы X и Y обычно являются двумерными такого же размера, как массив Z.

```
fig = plt.figure(facecolor='white')
ax1 = fig.add_subplot(1, 2, 1)
ax1.contour(X,Y,Z,40);

locs =np.linspace(-2,2,5)
ax1.set_xticks(locs, minor = False);
ax1.set_yticks(locs, minor = False);

ax2 = fig.add_subplot(1, 2, 2)
ax2.contourf(X,Y,Z,40);
ax2.set_xticks(locs, minor = False);
ax2.set_yticks(locs, minor = False);
```



Элементы оформления графических областей создаются аналогично тому, как это делается в случае одной графической области.

Можно задавать не количество линий уровня, а массив значений, для которых рисуются линии постоянного значения. Добавьте перед первой строкой предыдущего кода инструкцию

```
vals=np.array([-1.5,-0.9,-0.6,-0.36,-0.18,-0.05,0,0.05,0.25,0.5,0.75,
                                                    0.95,1.5, 2.5,3.5,5,7])
```

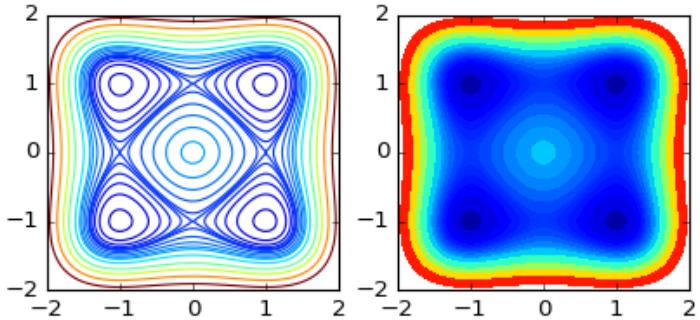
и замените строки `ax1.contour(X,Y,Z,40)` и `ax2.contourf(X,Y,Z,40)` следующими командами

```
ax1.contour(X,Y,Z,vals);
```

и

```
ax2.contourf(X,Y,Z,vals);
```

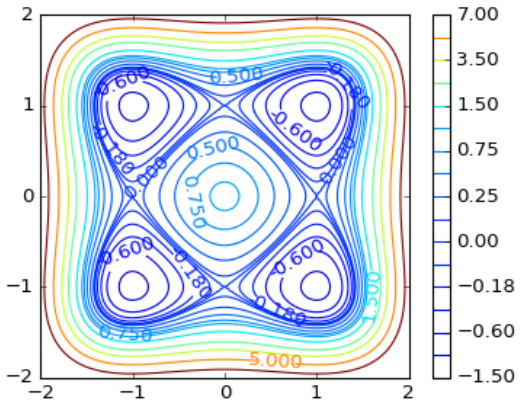
Выполните блок нового кода и получите следующие графики.



Вместо массива значений линий уровня `vals` можно использовать опцию `levels=список_значений`.

С помощью функции `clabel(cs[,массив_значений,...])` на контурных линиях можно отобразить значения, которым они соответствуют. Здесь `cs` – ссылка на объект, который возвращает функция `contour()`. При задании аргумента `массив_значений`, метки будут рисоваться только на линиях, перечисленных в этом массиве.

```
vals=np.array([-1.5,-0.9,-0.6,-0.36,-0.18,-0.05,0,0.05,0.25,0.5,0.75,
                                    0.95, 1.5,2.5,3.5,5,7])
fig = plt.figure(facecolor='white');
ax=fig.gca()
cs=ax.contour(X,Y,Z,vals);
v=np.array([-0.6,-0.18,0,0.5,0.75, 1.5,5])
ax.clabel(cs, v)
locs =np.linspace(-2,2,5)
ax.set_xticks(locs, minor = False);
ax.set_yticks(locs, minor = False);
plt.colorbar(cs)
```

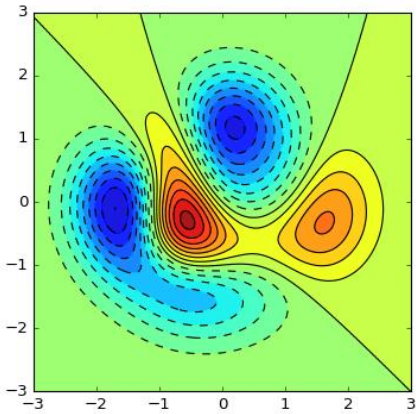


Функция `plt.colorbar(cs)` рисует палитру цветов, которая использована графическим объектом `cs`.

У функции `contour()` также имеются опции `linewidths` (толщина линий уровня), `linestyles` (стиль линий уровня с допустимыми значениями `None | 'solid' | 'dashed' | 'dashdot' | 'dotted'`). Если опции `colors` присвоить значение фиксированного цвета, то кривые постоянных положительных значений изображаются сплошными линиями, а отрицательных – пунктирными.

Часто удобно накладывать контурный график на залитый контурный график.

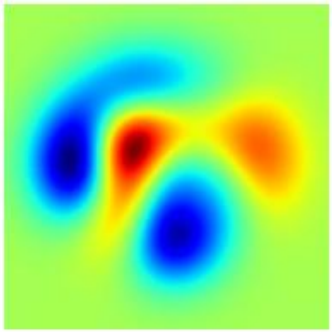
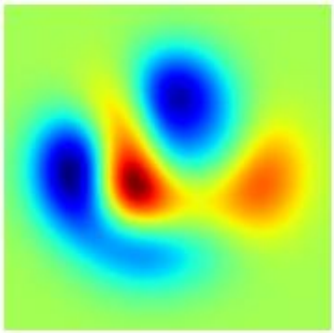
```
def f(x, y):
    return (1-x/4 + x**3 + y** 3) * np.exp(-x**2 - y**2)*(x**2-y-x)

t = np.linspace(-3, 3, 121);
X, Y = np.meshgrid(t, t);
Z= f(X, Y);
plt.contourf(X, Y, Z, 16, alpha=0.9, cmap='jet');
plt.contour(X, Y, Z, 16, colors='black', linewidth=.5);
plt.gcf().set_facecolor('w');
plt.gcf().gca().axis('image');
```



Матрицу Z можно представить графически с помощью функции `imshow()`. В результате получается изображение близкое к залитому контурному графику, но по–другому ориентированное.

```
plt.figure()
plt.imshow(Z);    # следующий рисунок слева
plt.gcf().set_facecolor('w');
```

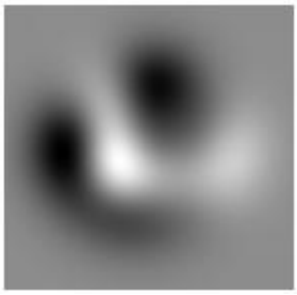
```
plt.gcf().gca().axis('image');
plt.axis('off')
```

Мы отключили отображение меток на осях командой `plt.axis('off')`, поскольку функция `imshow(Z)` относит график к индексам элементов матрицы `Z`. Следует помнить, что при отображении матрицы Z функцией `imshow(Z)` начало (точка [0,0]) расположено в левом верхнем углу и оси направлены вправо и вниз. На контурном же графике, оси направлены вправо вверх. Чтобы сравнить полученное изображение матрицы с контурным графиком, можно преобразовать матрицу Z подходящим образом. В данном случае ее нужно перевернуть сверху вниз. Это можно сделать функцией `numpy.flipud(Z)`.

```
plt.figure()
img=plt.imshow(np.flipud(Z));    # предыдущий рисунок справа
plt.gcf().set_facecolor('w');
plt.gcf().gca().axis('image');
plt.axis('off')
```

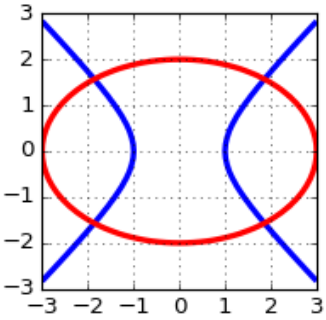
Заметим, что контурный график и изображение матрицы Z использовали одинаковую палитру цветов. Если палитры будут разными, то графики будет трудно сравнивать. Например, добавьте в конец предыдущего кода инструкцию `plt.gray()`, которая устанавливает серую палитру, и выполните код. Вы получите следующее изображение.



На контурном графике, если пожелаете, можно построить только одну нулевую линию уровня функции $f(x,y)$. В таком случае она (линия) будет представлять кривую, заданную неявным уравнением $f(x,y)=0$. В следующем примере мы строим гиперболу и эллипс по их неявным уравнениям: $x^2-y^2-1=0$ и $\frac{x^2}{3^2}+\frac{y^2}{2^2}-1=0$.

```
t=np.linspace(-3,3,41)
X,Y=np.meshgrid(t,t)
plt.figure()
```

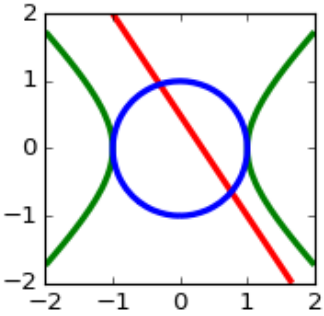
```
plt.contour(X, Y, X**2-Y**2-1, [0], linewidths=3,colors='b')
plt.contour(X, Y, X**2/9+Y**2/4-1, [0], linewidths=3,colors='r')
plt.gcf().set_facecolor('w');
plt.gcf().gca().axis('image');
plt.gcf().gca().grid(True)
```



Константу, которую мы использовали в левой части неявного уравнения, можно перенести в список значений линий уровня, т.е. в этом списке вместо нуля написать другое числовое значение. В результате будет построена та же кривая.

В команде `plt.contour(X,Y,Z[,...])` оба массива X и Y могут быть одномерными. При этом количество столбцов в двумерном массиве Z должно равняться количеству элементов в X, а количество строк в Z равняться количеству элементов в Y.

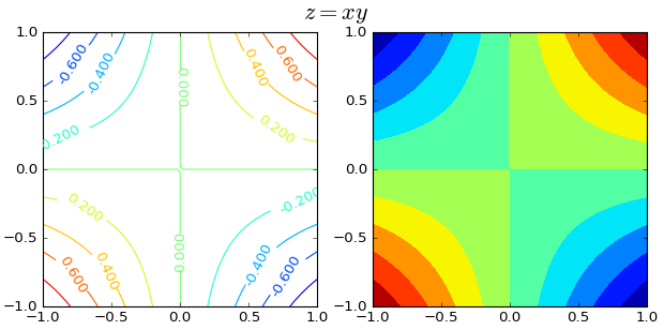
```
x = np.linspace(-2., 2.,41)
y = np.linspace(-2., 2.,41)
fig = plt.figure(facecolor='white');
ax=fig.gca()
plt.contour(x, y, 3*x + 2*y[:, None], [1],linewidths=3, colors='r');
plt.contour(x, y, x**2 - y[:, None]**2, [1],linewidths=3, colors='g')
plt.contour(x, y, x**2 + y[:, None]**2, [1],linewidths=3,colors='b')
ax.axis('image')
locs =np.linspace(-2,2,5)
ax.set_xticks(locs, minor = False);
ax.set_yticks(locs, minor = False);
```



Вот еще один пример.

```
x=np.linspace(-1,1,101);
y=x;
z=np.outer(x,y);
vals=np.linspace(-1,1,11);
fig = plt.figure(facecolor='white')
ax1 = fig.add_subplot(121)
```

```
curves=ax1.contour(x,y,z,vals,cmap='jet');
ax1.clabel(curves);
ax2 = fig.add_subplot(122)
ax2.contourf(x,y,z,vals,cmap='jet');
plt.suptitle(r'$z=xy$',fontsize=20)
```



Здесь функция `plt.suptitle()` печатает единый заголовок рисунка.

Для построения двумерных векторных полей используется функция `plt.quiver()`, а для построения линии тока векторного поля – функция `plt.streamplot()`.

Функция `plt.quiver()` строит график двумерных векторных полей. Допустимы следующие варианты вызова этой функции.

```
quiver(U, V[,...])
quiver(U, V, C[,...])
quiver(X, Y, U, V[,...])
quiver(X, Y, U, V, C[,...])
```

Массивы `U` и `V` представляют `x` и `y` компоненты векторного поля. Массивы `X` и `Y` задают положение стрелок (по умолчанию хвостов). Массив `C` определяет цвет стрелок. Обычно все эти массивы двумерные одинакового размера. Если массивы X и Y отсутствуют, то они соответствуют узлам регулярной координатной сетки.

Рассмотрим пример. Вначале подготовим массивы X, Y, U, V, C.

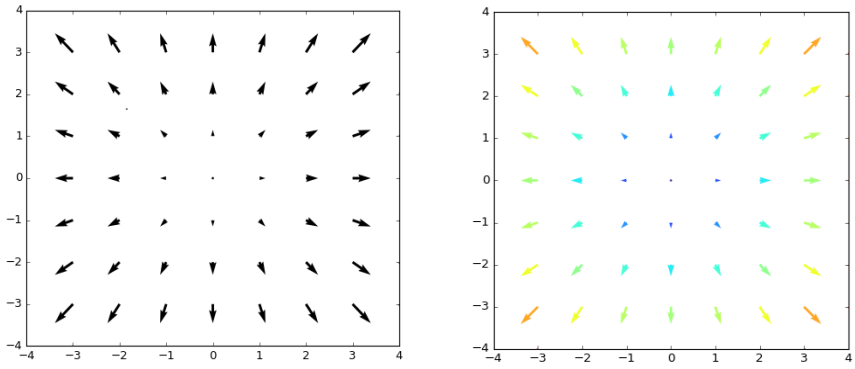
```
n = 4
X, Y = np.mgrid[-n:n+1, -n:n+1]
U=X; V=Y
C =  np.sqrt(X ** 2 +Y ** 2)
```

Теперь введите и выполните следующий блок кода.

```
fig = plt.figure(facecolor='white')
ax=fig.gca()
ax.quiver(X, Y, U, V) # следующий рисунок слева.
```

Замените последнюю строку следующей инструкцией, и снова выполните пример.

```
ax.quiver(X, Y, U, V, clr, alpha=.85)  # следующий рисунок справа
```



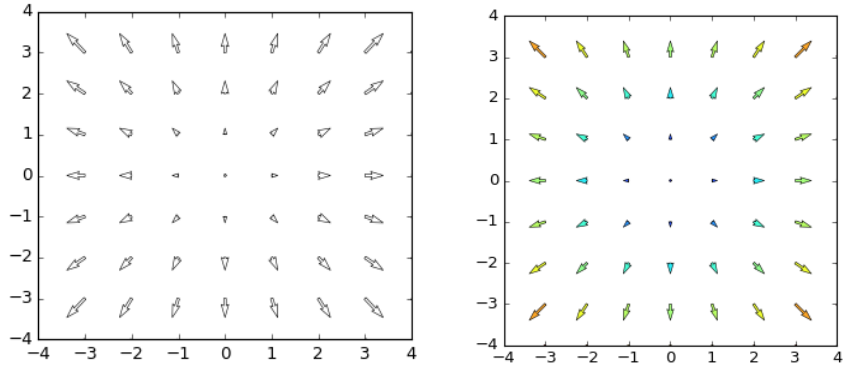
Опять замените последнюю строку инструкцией, и снова выполните пример.

```
ax.quiver(X, Y, U, V, edgecolor='k', facecolor='None', linewidth=.5)
```

Результат показан на следующем рисунке слева. Затем выполните код, который выполняет две последние инструкции сразу.

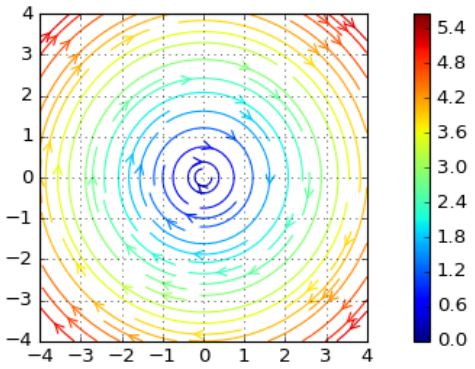
```
fig = plt.figure(facecolor='white')
ax=fig.gca()
ax.quiver(X, Y, U, V, clr, alpha=.85)
ax.quiver(X, Y, U, V, edgecolor='k', facecolor='None', linewidth=.5)
```

Результат показан на следующем рисунке справа.



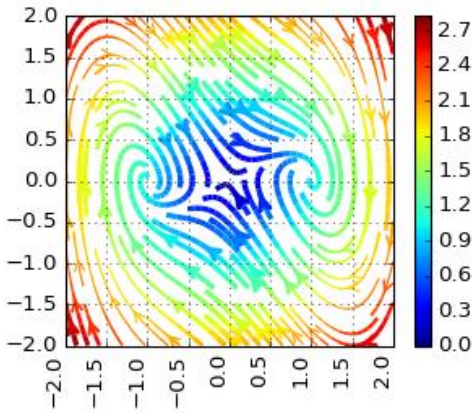
Похожий синтаксис имеет функция `plt.streamplot(X,Y,U,V[,...])`, которая рисует линии тока векторного поля.

```
fig = plt.figure(facecolor='white')
x=np.linspace(-4,4,81)
y=np.linspace(-4,4,81)
X, Y = np.meshgrid(x, y)
ln =  np.sqrt(X**2 + Y**2)
U=Y/ln; V=-X/ln
plt.streamplot(X, Y, U, V,
               color=ln,              # массив цветов
               linewidth=1,           # толщина линий
               arrowstyle='->',       # вид стрелок
               arrowsize=1.5)         # размер стрелок
plt.colorbar()          # палитра цветов
ax=fig.gca()
ax.grid(True)
ax.axis('image')
```



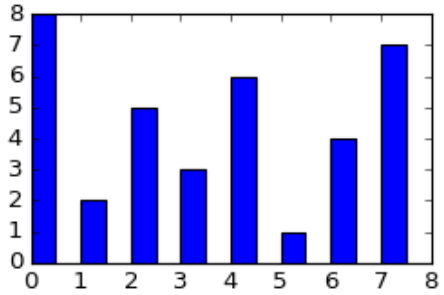
Кроме цвета, можно управлять толщиной линий, используя опцию `linewidth=массив_толщин`.

```
fig = plt.figure(facecolor='white')
x=np.linspace(-2,2,81)
y=np.linspace(-2,2,81)
X, Y = np.meshgrid(x, y)
ln =  np.sqrt(X ** 2 +Y ** 2)
U=Y; V=-Y+X-X**3
lw = (1+2*np.cos(np.pi*X*Y/4)**2)
plt.streamplot(X, Y, U, V, color=ln,
            linewidth=lw,           # толщина линий тока
            arrowstyle='->', arrowsize=1.5)
plt.colorbar()                      # палитра цветов
ax=fig.gca()
ax.grid(True)
fig.autofmt_xdate(rotation=90)
ax.axis('image');
```



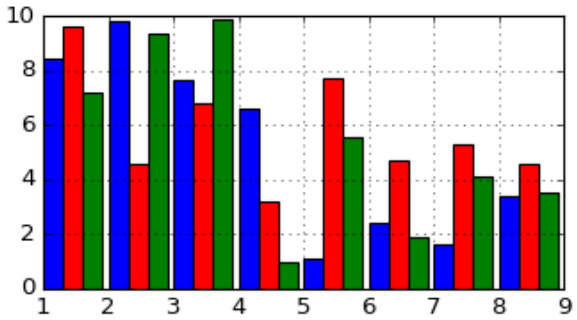
Столбчатые диаграммы дают визуальное представление об одномерном векторе/массиве данных. Каждому элементу вектора соответствует прямоугольник, высота которого зависит от значения этого элемента. Для построения столбчатой диаграммы используется функция `pyplot.bar(locs,vals[,width,...])`. Она принимает последовательность (вектор) координат `locs` левых краев столбцов, вектор значений `vals`, и ширину прямоугольников `width`, которая по умолчанию равна 0.8.

```
locs=np.arange(8)
data=[8,2,5,3,6,1,4,7]
plt.bar(locs, data, width=0.5,color='blue')
```



На одной диаграмме можно отобразить несколько векторов.

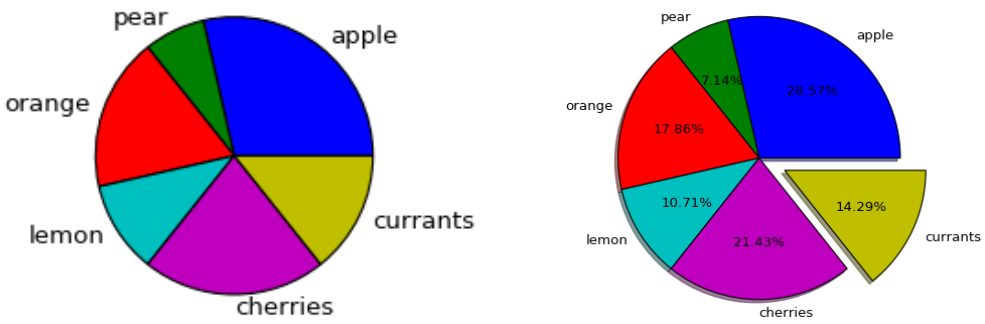
```
fig = plt.figure(facecolor='white')
n=8
data1=10*np.random.rand(n)
data2=10*np.random.rand(n)
data3=10*np.random.rand(n)
locs = np.arange(1,n+1)
wid = 0.3
plt.bar(locs, data1, width=wid,color='blue')
plt.bar(locs+wid, data2, width=wid, color='red')
plt.bar(locs+2*wid, data3, width=wid, color='green')
fig.gca().grid(True)
```



Кроме рассмотренной здесь функции `plt.bar()`, имеются другие функции построения диаграмм: `plt.barh()`, `plt.barbs()` и `broken_barh()`. С ними вы можете познакомиться самостоятельно по справочной системе.

Круговая диаграмма показывает вклада каждой компоненты вектора в итоговый результат (равный сумме всех компонент). Для построения круговой диаграммы используется функция `pyplot.pie(vals,labels[,...])`. Она принимает последовательность (вектор) значений `vals`, и их метки `labels`.

```
fig = plt.figure(facecolor='white')
data=[8,2,5,3,6,4]
lbls = ['apple', 'pear', 'orange', 'lemon', 'cherries', 'currants']
plt.pie(data, labels = lbls);          # следующий рисунок слева
fig.gca().axis('image')
```



Имеются различные аргументы, предназначенные для оформления таких диаграмм. Опция `explode`, если задается, является последовательность того же размера, что и вектор `vals`, и содержит значения радиального сдвига секторов круговой диаграммы. Опция `autopct` определяет форматирование числовых меток, которые по умолчанию показывают долю каждой компоненты вектора `vals` в процентах.

```
fig = plt.figure(facecolor='white')
data=[8,2,5,3,6,4]
epd = [0, 0, 0, 0, 0, 0.2]
lbls = ['apple', 'pear', 'orange', 'lemon', 'cherries', 'currants']
plt.pie(data, labels = lbls,explode = epd,
            autopct = '%2.2f%%',shadow=True);
fig.gca().axis('image')       # предыдущий рисунок справа
```

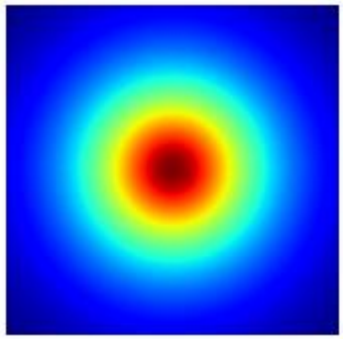
Гистограммы. Гистограммой в дискретном анализе называется геометрическое изображение дискретной функции (массива), которое строится следующим образом. Сначала множество значений разбивается на несколько примыкающих интервалов (корзин), которые откладываются на горизонтальной оси. Затем подсчитывается количество элементов массива, попавшее в каждую корзину. Над каждым интервалом рисуется прямоугольник, высота которого пропорциональна этим числам.

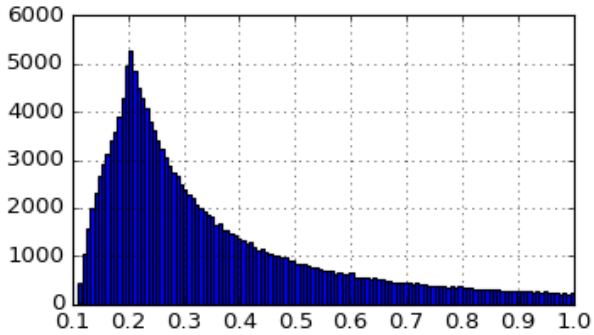
Для построения гистограмм в `matplotlib` используется функция `pyplot.hist(вектор, n)`, где `n` – количество корзин.

В следующем примере мы строим гистограмму двумерного массива `Z`, изображение которого показано на следующем рисунке слева. Двумерный массив перед передачей в функцию `hist()` преобразуется в одномерный. Множество значений массива Z (функции `f(x,y)`) разбито на 128 интервалов (корзин). Сама гистограмма показана справа.

```
def f(x, y):
    return 1/(1+x**2+y**2)

t = np.linspace(-2, 2, 401);
X, Y = np.meshgrid(t, t);
Z= f(X, Y);
fig = plt.figure(facecolor='white')
ax=fig.gca()
img=ax.imshow(Z);
ax.axis('image');
```

```
ax.axis('off')

G=Z.flatten();
fig = plt.figure(facecolor='white')
plt.hist(G, 128);
fig.gca().grid(True)
```





Заметим, что имеются еще функции построения гистограмм: `plt.hist2d()` и `plt.hlines()`.

В модуле `matplotlib.pyplot`, кроме рассмотренных выше функций, имеется много других графических функций. С ними мы предлагаем вам познакомиться самостоятельно.

В завершении, приведем еще несколько функций, которые управляют графиками.

Функция `matplotlib.pyplot.cla()` очищает графическую область (**cl**ear **a**xes). Функция `matplotlib.pyplot.clf()` очищает рисунок от всех графических объектов (**cl**ear **f**igure).

Функция `matplotlib.pyplot.clim(vmin=v1,vmax=v2)` устанавливает диапазон изменения цвета на текущем изображении.

4.2 Трехмерные графики matplotlib

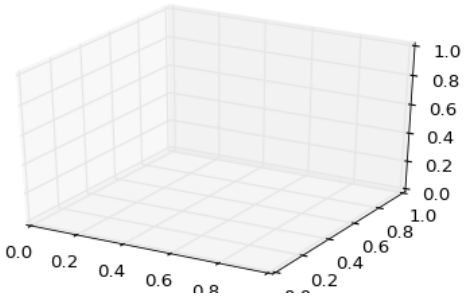
Пакетом, предназначенным для работы с трехмерными графиками, является `mpl_toolkits.mplot3d`. В основном нам будут требоваться объекты и функции из его подмодуля `Axes3D`. Кроме импортирования этого модуля, нам почти всегда нужно будет импортировать пакет `numpy` и, конечно, модуль `matplotlib` и его подмодуль `matplotlib.pyplot`.

Договоримся, что будем строить графики в IPython Console Spyder. Чтобы трехмерные графики можно было свободно вращать мышкой, такие графики следует строить в графическом окне matplotlib, а не отображать их в документе. Для этого нам потребуется выполнить магическую команду `%matplotlib qt`. В результате, в начале каждой сессии IPython Console Spyder, в которой предполагается строить трехмерную графику, следует выполнить следующие инструкции.

```
%matplotlib qt
import numpy as np
```

```
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```
Будем полагать, что перед построением любого графика этого параграфа, эти команды уже выполнены.

Для построения трехмерного графика надо создать трехмерные оси с помощью экземпляра класса `mpl_toolkits.mplot3d.Axes3D`. Его конструктор ожидает экземпляр класса `Figure`, который, в свою очередь, можно создать вызовом функции `matplotlib.pyplot.figure()`.Т.о., в начале каждого примера мы будем вызывать две команды:
```
fig = plt.figure()        # создаем пустое графическое окно
ax=Axes3D(fig)            # создаем в нем трехмерные оси
```
Фактически функция `figure()` создает пустое графическое окно, а результатом выполнения второй команды является изображение в этом окне трехмерной системы координат.



Для экономии места мы не показываем рамку и панель управления этого окна. Кроме того, это изображение может быть встроенным в IPython документ. Тогда рамки и панели вообще не будет.

При создании графического окна (объекта Figure) с помощью функции `figure(...)`, ей можно передавать аргументы, которые управляют оформлением. Например,
```
fig = plt.figure(num=номер)                    # номер окна
```
Если рисунок с этим номером существует, то команда активирует его и возвращает на него ссылку; если рисунка с таким номером нет, то он создается и возвращается ссылка на него. Если аргументу `num` передается строка, то создается графическое окно с этим заголовком.
```
fig = plt.figure(num='Hello Univer!')          # заголовок окна
```
По команде `plt.figure()` (без аргумента `num`) создается новый рисунок с номером, большим на 1, чем последний.

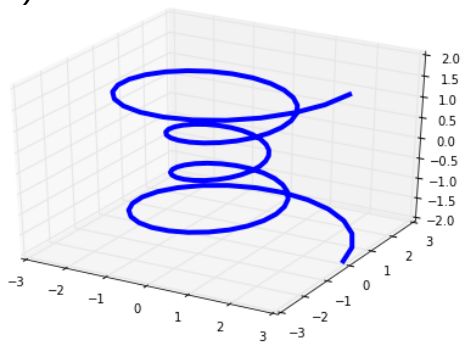
Еще одним необязательным аргументом является `figsize`, которому можно передавать кортеж из двух целых чисел, задающих размер графического окна в дюймах. Например,
```
fig = plt.figure(figsize=(8,8))                # размер в дюймах
```
Необязательный аргумент `facecolor` задает цвет фона окна. Например,
```
fig = plt.figure(facecolor='white')            # белый фон рисунка
```
Команда `fig.clear()` очищает графическое окно вместе с объектом `Axes3D`, если он был создан.

Команда `plt.close(fig)` закрывает окно, на которое ссылается переменная `fig`. Инструкция `plt.close()` закрывает окно текущего рисунка. Инструкция `plt.close(num)` закрывает окно рисунка с номером `num`. Команда `plt.close('all')` закрывает окна всех рисунков.

После создания окна рисунка и графического объекта `Axes3D` дальнейшие действия зависят от функции, которую предполагается использовать для построения трехмерного графика.

Трехмерная кривая (или ломаная) строится с помощью функции `Axes3D.plot(x,y,z[,...])`, которая принимает вектора `x,y,z` координат узлов ломаной. При большом количестве узлов ломаная неотличима от кривой.

```
fig = plt.figure()
ax=Axes3D(fig)
t=np.linspace(-4*np.pi,4*np.pi,100)
r=t**2/80+1
x=r*np.cos(t)     # праметрическое уравнение кривой
y=r*np.sin(t)
z=t/(2*np.pi)
ax.plot(x,y,z,linewidth=4)
```



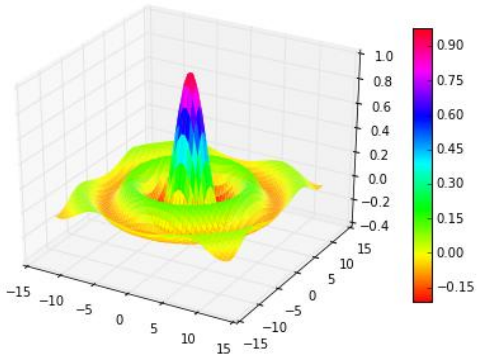
Задание свойств кривой и параметров оформления трехмерного графика аналогичны двумерным графикам.

График поверхности строится с помощью функции

`Axes3D.plot_surface(X,Y,Z,rstride=1, cstride=1[,...])`,

где `X,Y,Z` двумерные массивы, содержащие `x`, `y` и `z` координаты узлов многогранной поверхности, а параметры `rstride=1` и `cstride=1` определяют шаг по индексам (**r**ow и **c**olumn) этих массивов. Единицы означают выбор всех значений в двумерных массивах `X,Y,Z`. При большом размере матриц многогранная поверхность неотличима от криволинейной.

```
fig=plt.figure()
ax=Axes3D(fig)
u=np.linspace(-4*np.pi,4*np.pi,50)
x,y=np.meshgrid(u,u)
r=np.sqrt(x**2+y**2)
z=np.sin(r)/r
surf=ax.plot_surface(x,y,z,rstride=1,cstride=1,linewidth=0,
                     cmap=mpl.cm.hsv)
fig.colorbar(surf, shrink=0.75, aspect=15)
```



Опция `cmap=mpl.cm.hsv` задает `hsv` способ раскраски в зависимости от `z` координаты точек поверхности. Метод `fig.colorbar(...)` рисует справа от графика цветовую полосу (палитру). Ее аргумент `shrink` задает коэффициент сжатия (по высоте) полосы относительно рисунка, а аргумент `aspect` определяет пропорцию высоты и ширины полосы.

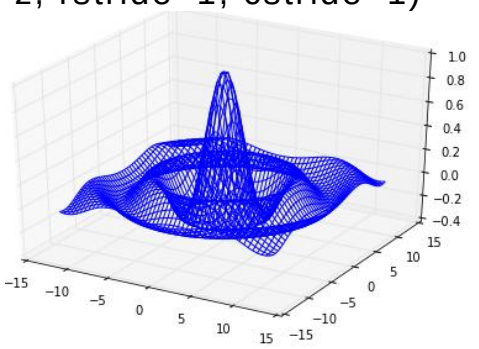
Каркасная поверхность строится с помощью функции

`Axes3D.plot_wireframe(X,Y,Z,rstride=1, cstride=1[,...])`,

где `X,Y,Z` двумерные массивы, содержащие `x`, `y` и `z` координаты узлов многогранной поверхности, а параметры `rstride=1` и `cstride=1` определяют шаг по индексам этих массивов.

Удалите в предыдущем примере две последние инструкции и добавьте одну команду

```
. . .          # вместо многоточия стоит 6 первых строк предыдущего примера
ax.plot_wireframe(x, y, z, rstride=1, cstride=1)
```



Версии графических библиотек постоянно обновляются. Объекты `Axes3D` в последней версии `matplotlib` рекомендуется создавать с использованием ключевого слова `projection='3d'` парой следующих команд:

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

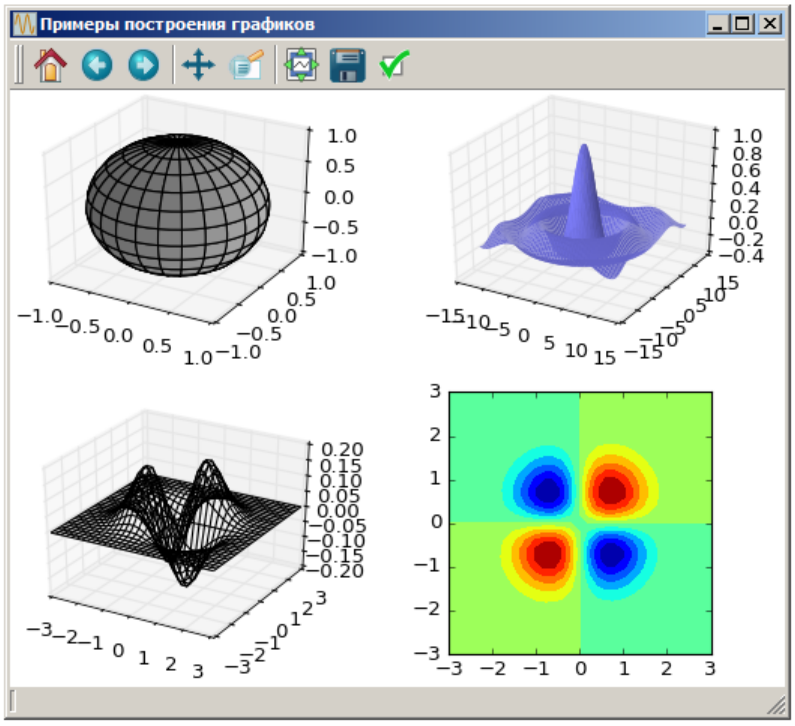
Вместо последней инструкции также можно использовать следующую:

```
ax = plt.axes(projection='3d')
```

В общем случае функция `add_subplot(...)` предназначена для построения нескольких графиков в различных областях одного окна. Но ее можно использовать и для создания единственного графика. Построение одного трехмерного графика не отличается от того, что мы делали выше. Покажем, как использовать эту функцию так, чтобы в разных областях находились как двумерные, так и трехмерные графики.

```
# Построение 4-х графиков в одном графическом окне
```

```
# Левый верхний график (сфера)
fig = plt.figure(figsize=(8,6),facecolor='white',
                 num='Примеры построения графиков')
ax = fig.add_subplot(2, 2, 1, projection='3d')
u = np.linspace(0, 2 * np.pi, 51)
v = np.linspace(-np.pi/2, np.pi/2, 51)
x = np.outer(np.cos(u), np.cos(v))
y = np.outer(np.sin(u), np.cos(v))
z = np.outer(np.ones(np.size(u)), np.sin(v))
ax.plot_surface(x, y, z, rstride=2, cstride=4, color='0.75')
# правый верхний график
u=np.linspace(-4*np.pi,4*np.pi,50)
X, Y = np.meshgrid(u, u)
r = np.sqrt(X**2 + Y**2)
Z=np.sin(r)/r
ax = fig.add_subplot(2, 2, 2, projection='3d')
ax.plot_surface(X,Y,Z, rstride=1, cstride=1,
                 linewidth=0,color=(0.5,0.5,1))
# левый нижний график (каркасная поверхность)
pt=np.linspace(-3,3,31)
X,Y=np.meshgrid(pt,pt)
Z=X*Y*np.exp(-X**2-Y**2)
ax = fig.add_subplot(2, 2, 3, projection='3d')
ax.plot_wireframe(X, Y, Z, rstride=1, cstride=1,color='k')
# правый нижний график (залитый контурный график)
ax = fig.add_subplot(2, 2, 4)
plt.contourf(X, Y, Z, 12,  cmap='jet')
plt.axis('image')
```



Приведенный код удобнее всего оформить в виде Python программы. Мы набирали его в редакторе Spyder. При этом, во время отладки выполняли код по частям, выделяя желаемый блок и выполняя его, нажав клавишу F9.

Обсудим некоторые функции из этого кода.

Команде `plt.figure(...)` переданы аргументы:

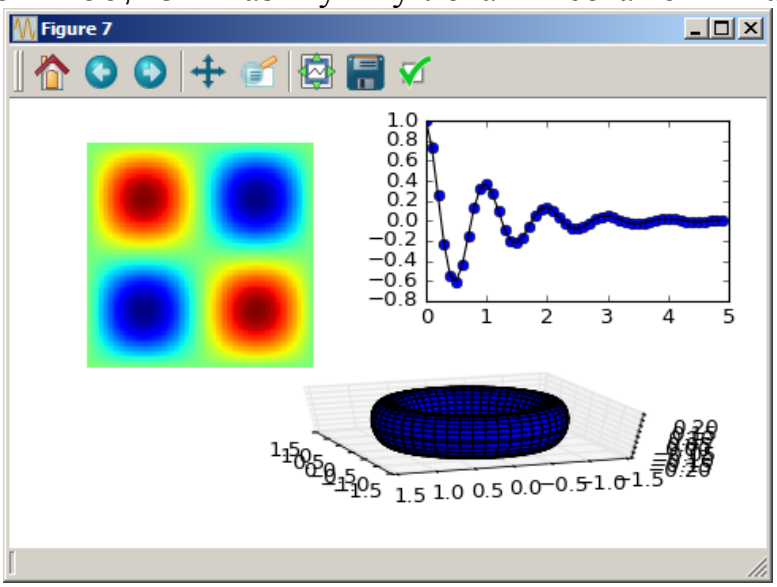
- `figsize=(8,6)` – задает размер графического окна в дюймах;
- `facecolor='white'` – задает белый фон графического окна
- `num='текст'` – если аргументу `num` передана строка, то она задает заголовок графического окна.

Команда `fig.add_subplot(2, 2, 1, projection='3d')` создает в графическом окне 4 области, организованные в 2 строки и 2 столбца (2 x 2), и делает активной первую из них. Также для этой области задается трехмерный «тип проекции» (`projection='3d'`), который позволяет соответствующему объекту `Axes` отображать трехмерные данные.

Если вы хотите расположить графические области (подокна) вручную, например «мозаикой», используйте команду `axes()`, которая позволяет задать положение осей (графических областей) как `axes([left, bottom, width, height])`, где все значения изменяются от 0 до 1. Выглядеть это может так:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

def g(x,y):
    return np.sin(np.pi*x)*np.sin(np.pi*y)

fig = plt.figure(facecolor='white')
# верхний правый график
t = np.arange(0.0, 5.0, 0.1)
plt.axes([0.55, 0.55, 0.4, 0.4])
plt.plot(t, f(t), 'bo', t, f(t), 'k')
# график плотности слева
x = np.linspace(-1,1,101)
X,Y=np.meshgrid(x,x)
Z=g(X,Y)
ax=plt.axes([0.0, 0.4, 0.5, 0.5])
plt.imshow(Z,interpolation='nearest')
ax.axis('off')
# правый нижний график (тор)
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.sin(th),
```

```
        r*np.sin(ph)
ax=Axes3D(fig,rect=[0.3,0.15,0.6,0.25])
ax.plot_surface(x,y,z,rstride=2,cstride=1)
ax.azim, ax.elev=160,45   # азимут и угловая высота точки наблюдения
```

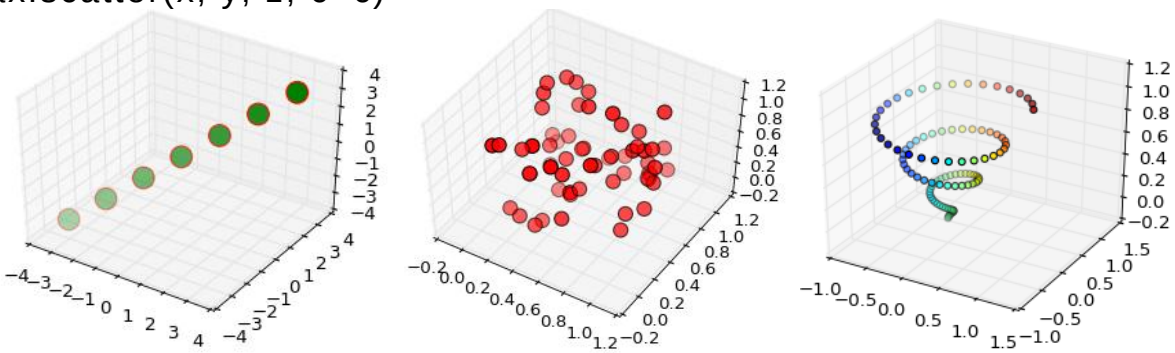


Цвет поверхности задается опцией `color=цвет`. Сам цвет можно задать строкой его названия, например, `color='lightblue'` (светло синий), или для основных цветов сокращенно одной буквой, например, `color='k'` (черный). Яркость серых цветов можно задать с помощью строки, содержащей число в интервале от `0` до `1` (1 – белый, 0 – черный), например, `color='0.75'`. Кроме того, цвет можно задавать RGB строкой, содержащей шестнадцатеричное число, перед которым стоит символ '#' (решетка). Например, `color='#E0FFFF'`. Первые две цифры задают яркость красной составляющей, вторые – зеленой, третьи – синей. Также цвет в формате RGB можно задавать кортежем, состоящим из трех чисел из интервала [0, 1], например, `color=(0.9,0.5,1)`. Также можно использовать цветовые карты, с помощью которых можно по–разному окрашивать участки трехмерной поверхности.

График рассеяния строится функцией `Axes3D.scatter(x, y, z[,...])`. Функция рисует множество точек, абсциссы которых передаются в векторе `x`, ординаты – в векторе `y`, аппликаты – в векторе `z`.

```
# следующий рисунок слева
x=y=z=np.arange(-3,4)
fig = plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, s=200, color='r', c='g')
# следующий рисунок в центре
n = 60
xs = np.random.rand(n)
ys = np.random.rand(n)
zs = np.random.rand(n)
fig = plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')
```

```
ax.scatter(xs, ys, zs, c='r', marker='o',s=200)
plt.show()
# следующий рисунок справа
fig = plt.figure(facecolor='white')
ax = plt.axes(projection='3d')
z = np.linspace(0, 1, 100)
x = z * np.sin(20 * z)
y = z * np.cos(20 * z)
c = x + y
ax.scatter(x, y, z, c=c)
```

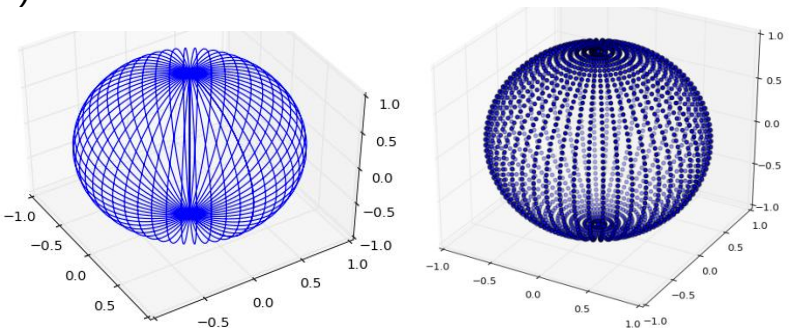


Одномерные массивы координат точек также использует функция `Axes3D.plot3D(...)`. Она имеет следующий формат:

`Axes3D.plot3D(x,y,z[,...])`,

где `x`, `y` и `z` одномерные массивы, содержащие координаты узлов. Используя функцию `numpy.ravel()`, можно двумерные массивы координат точек поверхности преобразовать в одномерные массивы и, используя их, построить поверхность. В следующем примере кроме поверхности, построенной по таким одномерным массивам, мы строим график рассеяния по тем же точкам.

```
u = np.linspace(0, 2 * np.pi, 51)
v = np.linspace(-np.pi/2, np.pi/2, 51)
x = np.outer(np.cos(u), np.cos(v))
y = np.outer(np.sin(u), np.cos(v))
z = np.outer(np.ones(np.size(u)), np.sin(v))
fig=plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot3D(np.ravel(x),np.ravel(y),np.ravel(z))      # след. рисунок слева
ax.scatter3D(np.ravel(x),np.ravel(y),np.ravel(z)) # след. рисунок справа
ax.set_xlim(-1,1)
ax.set_ylim(-1,1)
ax.set_zlim(-1,1)
```

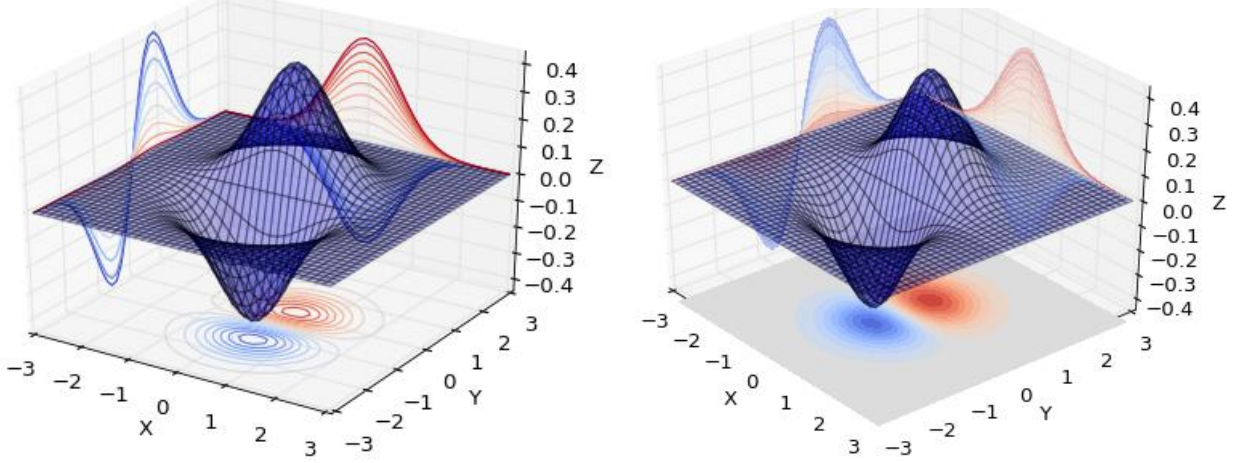


Контурный график создается функцией

`Axes3D.contour(X,Y,Z, zdir='dir',offset=val[,...])`,

где `X,Y,Z` двумерные массивы, содержащие координаты узлов многогранной поверхности. При большом размере матриц контурные линии неотличимы от кривых. Строка `'dir'` может принимать значения `'x'`, `'y'` или `'z'`. Она задает направление проецирования. Соответствующие линии постоянного значения рисуются на плоскости `x=val` (`y=val` или `z=val`), положение которой задается опцией `offset=val` . Необязательный параметр `cmap` задает цветовую палитру. Необязательный аргумент `levels` задает вектор значений, для которых строятся контурные линии.

```
pt=np.linspace(-3,3,41)
X,Y=np.meshgrid(pt,pt)
Z=Y*np.exp(-X**2-Y**2)
fig=plt.figure()
ax = fig.add_subplot(111, projection='3d')
# следующий график слева
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,alpha=0.3)
xlevels = np.array([0.,0.25,0.5,0.75,1,1.25,1.5,2])
ylevels = np.linspace(-0.5,0.5,20)
zlevels = np.linspace(-0.5,0.5,20)
ax.contour(X,Y,Z,zdir='x',offset=-3,cmap=mpl.cm.coolwarm,
          levels=xlevels)
ax.contour(X,Y,Z,zdir='y',offset=3,cmap=mpl.cm.coolwarm,
          levels=ylevels)
ax.contour(X,Y,Z,zdir='z',offset=-0.5,cmap=mpl.cm.coolwarm,
          levels=zlevels)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
```



Если заменить три инструкции предыдущего кода, содержащие функции `ax.contour(...)` следующими инструкциями:

```
ax.contourf(X,Y,Z,zdir='x',offset=-3,cmap=mpl.cm.coolwarm,
          levels=xlevels,alpha=0.4)
ax.contourf(X,Y,Z,zdir='y',offset=3,cmap=mpl.cm.coolwarm,
          levels=ylevels,alpha=0.2)
```

```
ax.contourf(X,Y,Z,zdir='z',offset=-0.5,cmap=mpl.cm.coolwarm,
            levels=zlevels)
```

то будет построено изображение, показанное на предыдущем рисунке справа.

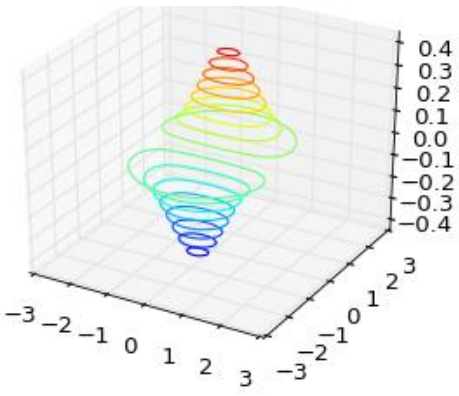
Использованные здесь функции `ax.contourf(...)` аналогичны функциям `ax.contour(...)`, но рисуют залитые контурные графики. Напомним также, что опция `alpha=a`, где $0 \le a \le 1$, задает прозрачность поверхности.

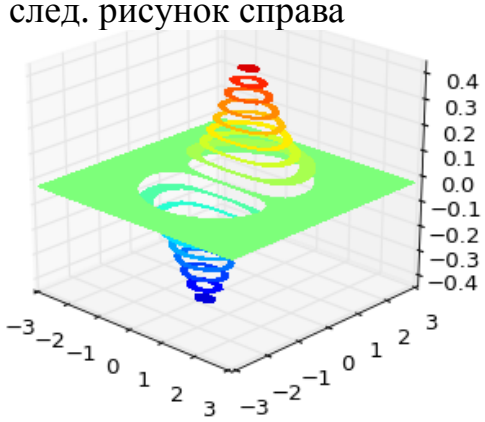
Контурные линии (линии постоянного значения) в пространстве рисует функция `Axes3D. contour3D(X,Y,Z,[,...])`, где `X,Y,Z` двумерные массивы, содержащие координаты узлов многогранной поверхности. Аналогичная функция `Axes3D. contourf3D(X,Y,Z,[,...])`, рисует «сплошные» плоские контурные области в пространстве. Если функция `Axes3D.contourf(X,Y,Z,N[,...])` не использует опцию `offset`, то она подобно функции `contourf3D(...)` рисует `N` «сплошных» плоских контурных областей в пространстве

```
pt=np.linspace(-3,3,41)
X,Y=np.meshgrid(pt,pt)
Z=Y*np.exp(-X**2-Y**2)
fig=plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')
ax.contour3D(X,Y,Z, levels=zlevels)  # след. рисунок слева
```

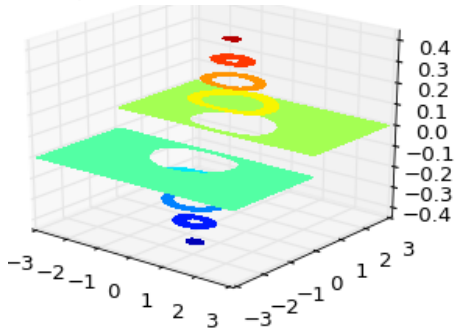
Если вместо предыдущей инструкции выполнять по одной следующие строки, то вы получите другие рисунки.

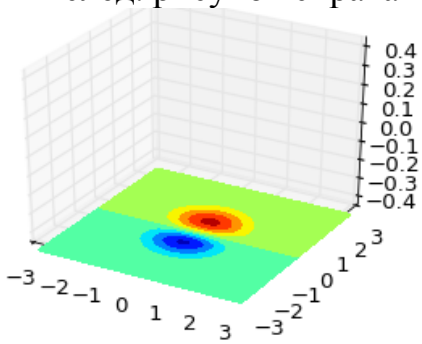
```
ax.contourf3D(X,Y,Z, levels=zlevels) # след. рисунок справа
```






```
ax.contourf(X, Y, Z, 10)                  # след. рисунок слева
ax.contourf(X, Y, Z, 10, offset=-0.5)     # след. рисунок справа
```

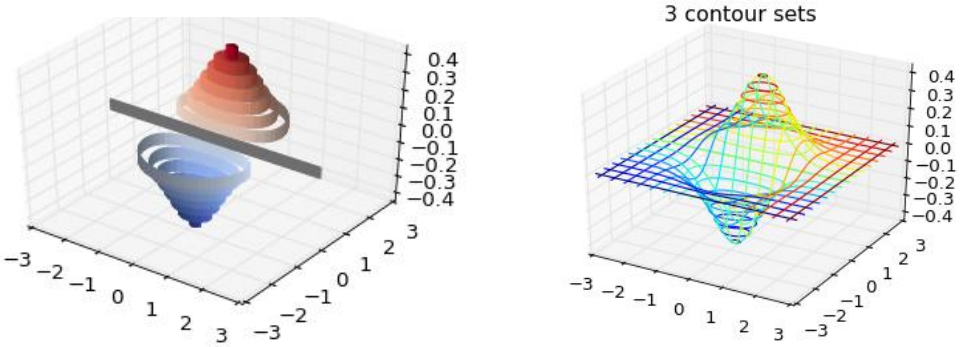




У функции `Axes3D. contourf3D(X,Y,Z,[,...])` есть опция `extend3d`. Если ее установить в `True`, то она позволяет рисовать пространственные линии

уровня в форме полос (а не кривых). Совместно с ней используется опция `stride=шаг`, которая задает шаг по матрицам данных при рисовании полос уровня.

```
pt=np.linspace(-3,3,61)
X,Y=np.meshgrid(pt,pt)
Z=Y*np.exp(-X**2-Y**2)
fig=plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')
cset = ax.contour(X,Y,Z,15,extend3d=True, cmap=mpl.cm.coolwarm,
            stride=1, zdir='z')  # следующий рисунок слева
```



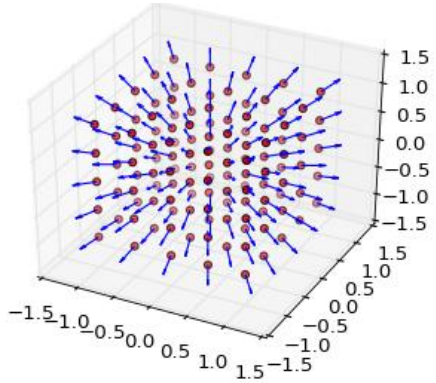
Если графические функции вызываются общим объектом `Axes3D`, то и графики рисуются в общей трехмерной области. Удалите в предыдущем коде последнюю инструкцию и введите следующие:

```
ax.contour(X, Y, Z, 15,  stride=1, zdir='x')
ax.contour(X, Y, Z, 15,  stride=1, zdir='y')
ax.contour(X, Y, Z, 15,  stride=1, zdir='z')
ax.set_title('3 contour sets ')
```

Выполните пример, и получите изображение, показанное на предыдущем рисунке справа. Функция `Axes3D.set_title(...)` печатает заголовок графика.

Функция `Axes3D.quiver(X, Y, Z, U, V, W[, ...])` рисует трехмерное векторное поле. Здесь `X, Y, Z` являются трехмерными массивами, содержащими `x, y` и `z` координаты опорных точек (точек, к которым будут «привязаны» вектора), а `U, V, W` являются массивами того же размера, содержащие компоненты векторного поля. Опция `length=число` задает длину стрелок векторного поля. Опция `pivot` может принимать значения `'tail'`, `'middle'` и `'tip'`, и задает положение опорных точек векторов (начало, середина или конец).

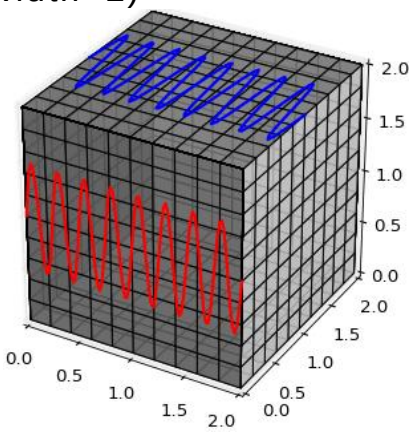
```
x, y, z = np.meshgrid(np.arange(-1, 1.5, 0.5),
                      np.arange(-1, 1.5, 0.5),
                      np.arange(-1, 1.5, 0.5))
u=x; v=y; w=z
fig = plt.figure(facecolor='white')
ax = fig.gca(projection='3d')
ax.scatter3D(np.ravel(x),np.ravel(y),np.ravel(z),c='r')
ax.quiver(x, y, z, u, v, w, length=0.3,pivot='tail')
```



Пример. Несложно нарисовать двумерные графики на гранях параллелепипеда, например, плоские кривые. Для этого достаточно построить графики трехмерных кривых, третья координата которых равна константе. В следующем примере мы рисуем куб и на его гранях две синусоиды. Вначале создаются три функции, представляющие параметрическое уравнение поверхности куба. Поскольку у пользователя функция абсолютного значения может обозначаться по–разному (например, `abs(x)`, `np.abs(x)`, `numpy.abs(x)`), то эти функции `xc()`, `yc()` и `zc()` в качестве первого аргумента принимают ссылку `Abs` на эту функцию. Затем на гранях куба дорисовываются две синусоиды.

```
# параметрическое уравнение куба
def xc(Abs,u,v):
   return Abs(u)-Abs(u-1)-Abs(u-2)+Abs(u-3)

def yc(Abs,u,v):
   return 1+Abs(v)-Abs(v-1)

def zc(Abs,u,v):
   return 1-Abs(Abs(u-1)-Abs(u-2)-Abs(u-3)+Abs(u-4)+\
            Abs(v-1)-Abs(v-2)+Abs(v)-Abs(v+1))/2+\
            Abs(2+Abs(u-1)-Abs(u-2)-Abs(u-3)+\
            Abs(u-4)+Abs(v-1)-Abs(v-2)+Abs(v)- Abs(v+1))/2

# изображение куба
u=np.linspace(0,4,41)
v=np.linspace(-1,2,31)
U,V=np.meshgrid(u,v)
X=xc(np.abs,U,V)
Y=yc(np.abs,U,V)
Z=zc(np.abs,U,V)
fig=plt.figure(facecolor='white')
ax=Axes3D(fig)
surf=ax.plot_surface(X,Y,Z,rstride=1,cstride=1,color='0.75',alpha=0.85)
ax.axis('image')

# изображение синусод на гранях куба
x = np.linspace(0, 2, 100)
y = np.sin(8*np.pi*x) / 2 + 1
zer=np.zeros(np.size(x))
```

```
ax.plot(x, y, 2,linewidth=2)
ax.plot( x, zer, y, 'r',linewidth=2)
```



Вообще то, кривые на параметрической поверхности следует строить, используя ее уравнения. Если уравнение поверхности имеет вид $x = x(u,v),\ y = y(u,v),\ z = z(u,v)$, то для построения кривой на этой поверхности следует задать ее уравнение в виде $u = u(t),\ v = v(t)$ и подставить в уравнение поверхности:

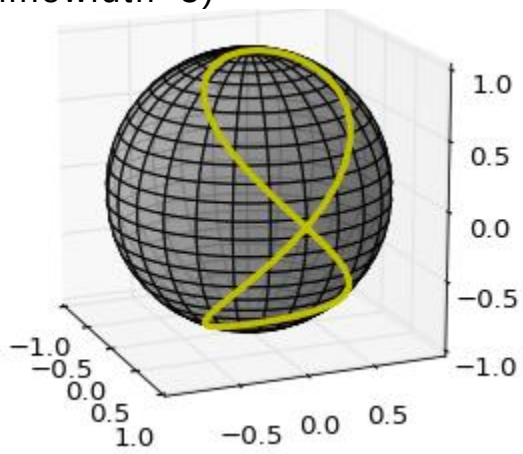
$$\tilde{x}(t) = x(u(t),v(t)),\ \tilde{y}(t) = y(u(t),v(t)),\ \tilde{z}(t) = z(u(t),v(t)).$$

Результирующие функции $\tilde{x}(t),\ \tilde{y}(t),\ \tilde{z}(t)$ будут представлять параметрическое уравнение кривой в пространстве, но кривая будет расположена на поверхности.

Пример. В следующем примере мы реализуем эти построения графическими инструментами библиотеки matplotlib. Вначале мы создаем функции, представляющие параметрическое уравнение сферы $xs(u,v), ys(u,v), zs(u,v)$, затем задаем уравнение кривой функциями $uc(t), vc(t)$. После этого мы создаем массивы X,Y,Z координат вершин многогранной поверхности, приближающей сферу, и строим ее. Затем создаем одномерные массивы XC, YC, ZC с координатами вершин ломаной, моделирующей кривую на поверхности, и строим ее.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

xs=lambda u,v: np.cos(u)*np.cos(v)
ys=lambda u,v: np.sin(u)*np.cos(v)
zs=lambda u,v: np.sin(v)
uc=lambda p: p
vc=lambda p: p

# построение поверхности
u = np.linspace(0, 2 * np.pi, 51)
v = np.linspace(-np.pi/2, np.pi/2, 51)
U,V=np.meshgrid(u,v)
X=xs(U,V)
Y=ys(U,V)
```

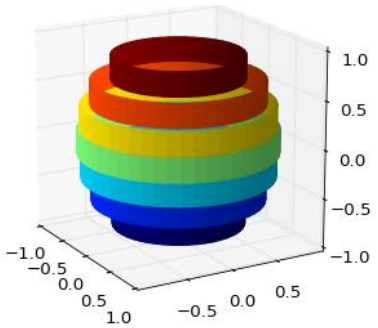
```
Z=zs(U,V)

fig=plt.figure(facecolor='white')
ax=Axes3D(fig)
surf=ax.plot_surface(X,Y,Z,rstride=2,cstride=2,color='0.75',alpha=0.85)
ax.axis('image')

# построение кривой
t = np.linspace(0, 2 * np.pi, 51)
XC=xs(uc(t),vc(t))
YC=ys(uc(t),vc(t))
ZC=zs(uc(t),vc(t))
ax.plot(XC,YC,ZC,'y',linewidth=3)
```



Вот еще несколько способов представления поверхностей

Пример (X,Y,Z из предыдущего примера сферы)

```
fig=plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')
ax.contour3D(X,Y,Z, extend3d=True,stride=1)
ax.axis('image')
```

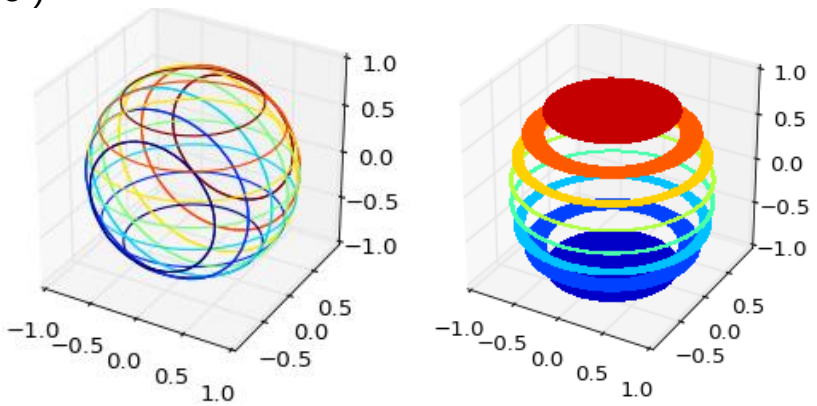


Пример (X,Y,Z из предыдущего примера сферы)

```
fig=plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')
# Следующий рисунок слева
ax.contour3D(X,Y,Z,zdir='y')
ax.contour3D(X,Y,Z,zdir='y')
ax.contour3D(X,Y,Z,zdir='z')
ax.axis('image')
```

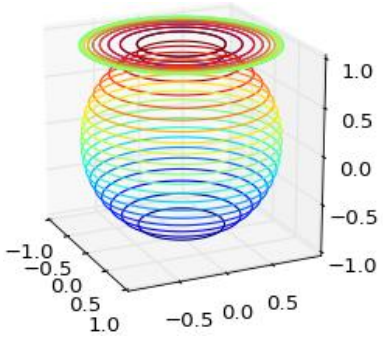
Добавьте в конце предыдущего примера следующие строки и выполните код.

```
fig=plt.figure(facecolor='white')          # новый рисунок
ax = fig.add_subplot(111, projection='3d')
ax.contourf3D(X,Y,Z,zdir='z')              # следующий рисунок справа
ax.axis('image')
```



В следующем коде дважды вызывается функция `contour(X,Y,Z,...)`, где массивы X,Y,Z взяты из предыдущего примера сферы. В первом вызове функции опция `offset` не используется, и поэтому рисуются контурные линии на поверхности сферы. Во втором вызове использование опции `offset=1` приводит к тому, что линии рисуются на плоскости `z=1`.

```
fig=plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')
ax.contour(X,Y,Z,20,zdir='z')              # 3D контурные линии на сфере
ax.contour(X,Y,Z,20,zdir='z',offset=1) # контурные линии на плоскости z=1
ax.axis('image')
```

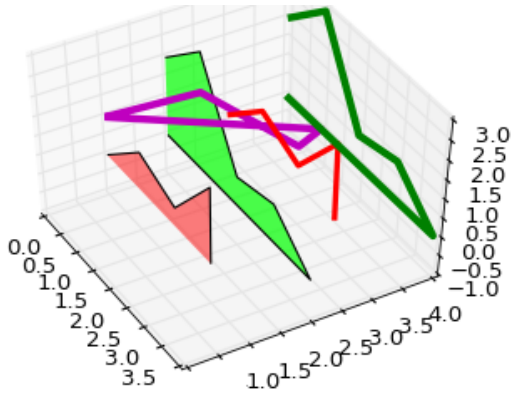


Кроме функций, строящих графики, в пакете matplotlib имеется ряд модулей с функциями, которые могут нарисовать в пространстве многоугольники, круги, напечатать текст, построить диаграмму и т.д.

В следующем примере мы демонстрируем использование функций `LineCollection` и `PolyCollection` для построения ломаных линий и многогранных поверхностей в пространстве.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection, LineCollection

verts=[[(0,0),(1,1),(2,0.5),(3,2),(3,0)],[(0,2),(1,3),(2,0.75),(3,1),
                                                    (4,0),(0,0)]]
poly = PolyCollection(verts,closed=False,
                                facecolors=[[1,0,0,0.5],[0,1,0,0.7]])
```

```
lin = LineCollection(verts,linewidths=[3,4], colors=['r','g'])
verts2=[[(0,1),(1,4),(2,3),(1,2),(0,1)]]
lin2 = LineCollection(verts2,linewidths=[4], colors=['m'])

fig = plt.figure(facecolor='white')
ax = fig.gca(projection='3d')
ax.add_collection3d(poly,zs=[1,2],zdir='y')
ax.add_collection3d(lin,zs=[3,4],zdir='y')
ax.add_collection3d(lin2,zs=[[1,0,1,2,1]])
ax.set_xlim3d(0, 4)
ax.set_ylim3d(0, 4)
ax.set_zlim3d(-1, 3)
```



Вначале фигуры создаются как двумерные, используя функции `LineCollection` и `PolyCollection`. Затем при добавлении коллекций на трехмерные оси функция `add_collection3d(...)` использует опцию `zs`, которая может быть числом z_0 (фигуры в плоскости $z=z_0$), а для ломаных может быть массивом третьих координат.

Функция `Axes3D.bar(...)` строит столбчатую диаграмму (набор столбиков) в пространстве. Она имеет следующий синтаксис:

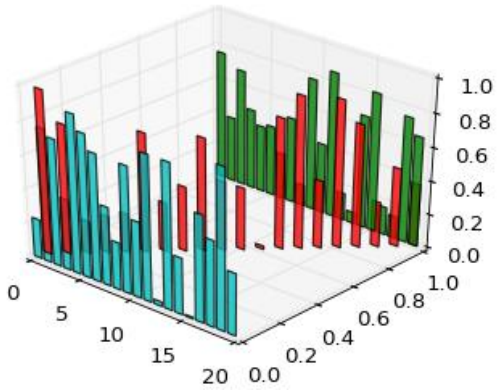
```
Axes3D.bar(x,y,z=0,zdir='z',color=массив_цветов,
                                    width=число[,...])
```

Опция `zdir` задает ориентацию прямоугольников (столбиков) диаграммы. Например, значение `zdir='y'` означает, что плоскости всех прямоугольников будут ортогональны оси Y. Вектор `x` задает абсциссу левых нижних углов столбиков, вектор `y` задает высоту столбиков, вектор `z` определяет аппликату (z–координату) левых нижних углов столбиков. Опция `width` определяет ширину столбиков, которая по умолчанию равна 0.8. Опция/вектор `color` задает цвет каждого столбика.

В следующем примере мы строим три пространственные диаграммы. Первая и вторая расположены в плоскостях `y=0` и `y=1`. Плоскости столбиков третьей диаграммы меняются от `y=0.05` до `1`.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

```
fig = plt.figure(facecolor='white')
ax = fig.add_subplot(111, projection='3d')

xs1 = np.arange(20)         # x координаты левых нижних углов столбиков
ys1 = np.random.rand(20)        # массив 20 чисел из диапазона [0,1)
zs1=np.zeros(len(xs))           # z координаты столбиков
cs1 = ['c'] * len(xs)           # цвет столбиков
ax.bar(xs1, ys1, zs=zs1, zdir='y', color=cs1, alpha=0.8, width=0.9)

xs2 = np.arange(20)
ys2 = np.random.rand(20)        # массив 20 чисел из диапазона [0,1)
zs2=np.ones(len(xs))            # z координаты столбиков
cs2 = ['g'] * len(xs)           # цвет столбиков
ax.bar(xs2, ys2, zs=zs2, zdir='y', color=cs2, alpha=0.8)

xs3 = np.arange(20)
ys3 = np.random.rand(20)
zs3 = np.linspace(0.05,1,20)    # z координаты столбиков
cs3 = ['r'] * len(xs)           # цвет столбиков
ax.bar(xs3, ys3, zs=zs3, zdir='y', color=cs3, alpha=0.8)
```

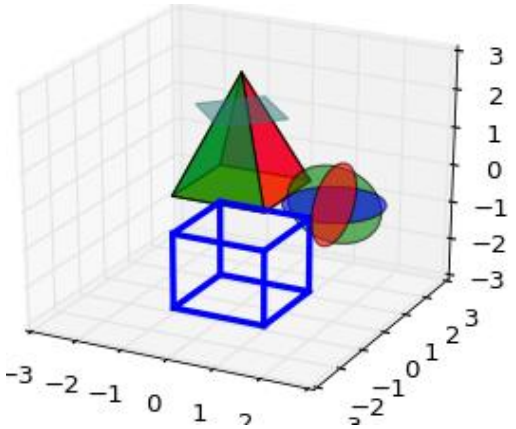


Трехмерные изображения, аналогично двумерным, могут содержать множество графических примитивов. Ими могут быть пространственные ломаные, прямоугольники, многоугольники, круги и т.д.

Пример. Рисование трехмерных графических примитивов.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.patches as patches
import mpl_toolkits.mplot3d.art3d as a3d
from matplotlib.patches import Circle
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

ax = Axes3D(plt.figure(facecolor='white'))
# три круга
c1= Circle((0, 0), 1,alpha=0.6,facecolor='r')
c2= Circle((2, 0), 1,alpha=0.6,facecolor='g')
c3= Circle((2, 0), 1,alpha=0.6,facecolor='b')
```

```
ax.add_patch(c1)
ax.add_patch(c2)
ax.add_patch(c3)
a3d.pathpatch_2d_to_3d(c1, z=2, zdir='x')
a3d.pathpatch_2d_to_3d(c2, z=0, zdir='y')
a3d.pathpatch_2d_to_3d(c3, z=0, zdir='z')
# пирамида
fc1=[[1,1,0],[0,0,3],[1,-1,0]]
fc2=[[1,-1,0],[0,0,3],[-1,-1,0]]
fc3=[[-1,-1,0],[0,0,3],[-1,1,0]]
fc4=[[-1,1,0],[0,0,3],[1,1,0]]
fc5=[[1,1,0],[1,-1,0],[-1,-1,0],[-1,1,0]]
pir=Poly3DCollection([fc1,fc2,fc3,fc4,fc5],
                                facecolors=['r','g','c','m','y'],alpha=0.75)
ax.add_collection3d(pir)
# ломаные (контур куба)
lc=a3d.Line3D((1,-1,-1,1,1,1,-1,-1,1,1),
          (1,1,-1,-1,1,1,1,-1,-1,1),
          (-1,-1,-1,-1,-1,-3,-3,-3,-3,-3),
          c='b',ls='-',linewidth=3)
ax.add_line(lc)
ax.add_line(a3d.Line3D((1,1),(-1,-1),(-1,-3),c='b',ls='-',linewidth=3))
ax.add_line(a3d.Line3D((-1,-1),(-1,-1),(-1,-3),c='b',ls='-',linewidth=3))
ax.add_line(a3d.Line3D((-1,-1),(1,1),(-1,-3),c='b',ls='-',linewidth=3))
# правильный многоугольник (здесь квадрат)
sq=patches.RegularPolygon((0,0), 4, 1,color='#337777',alpha=0.5)
ax.add_patch(sq)
a3d.pathpatch_2d_to_3d(sq, z=2, zdir='z')

ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
ax.set_zlim(-3,3)
```



■

Графических возможностей много и описать их все не представляется возможным. Кроме того, графические библиотеки постоянно обновляются, появляются новые модули, функции или новые опции у старых функций.

4.3 Анимация

Анимация – это последовательность изображений, которые быстро сменяют друг друга, в результате чего появляется движение. Для ее создания в matplotlib предназначен модуль `matplotlib.animation`. В этом параграфе мы опишем две функции этого модуля: `FuncAnimation` и `ArtistAnimation`. Функция `matplotlib.animation.FuncAnimation(...)` имеет следующий синтаксис:

```
FuncAnimation(fig, func, frames=count [,init_func=None,
                fargs=None, interval=ms, repeat=True,...])
```

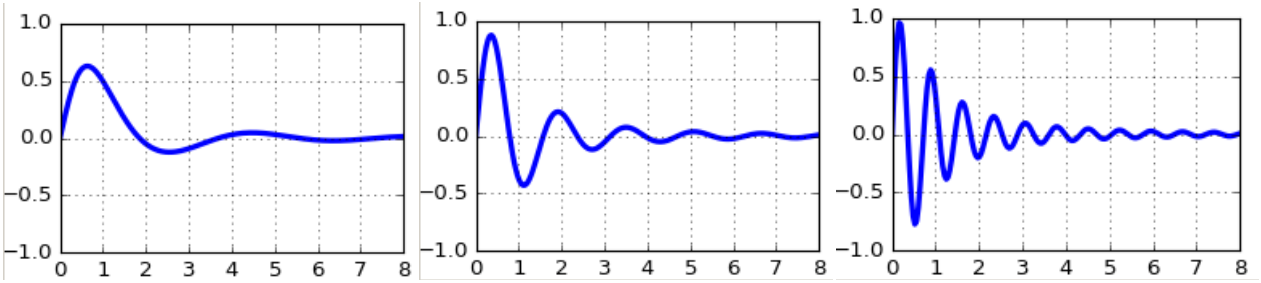
Анимация создается путем многократного вызова функции `func(...)`, которая должна рисовать кадры. В простейшем случае опция `frames` задает количество кадров, а функция `func(номер_кадра[,fargs])` в качестве первого аргумента принимает номер кадра. Функции `func`, если нужно, можно передавать аргументы через опцию `fargs`. Тогда `fargs=[arg1,...]` является списком. Функция, заданная необязательной опцией `init_func= funcname`, используется для рисования стартового кадра и инициализации графических объектов. Она вызывается один раз перед построеним первого кадра. Опция `interval` задает время в миллисекундах между кадрами. Опция `repeat=True` (значение по умолчанию) включает режим повторения анимации. В этом случае функция `funcname` будет вызываться в начале каждой «анимационной сессии».

Ниже приведен пример, создающий анимацию кривой.

Пример. *Анимация кривой*.

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

fig = plt.figure(facecolor='white')
ax = plt.axes(xlim=(0, 8), ylim=(-1, 1) )
line, = ax.plot([ ], [ ], lw=3)   # line = объект кривой
ax.grid(True)

def redraw(i):
   x = np.linspace(0, 8, 200)
   y = np.sin(i * x/10)/(1+x**2)
   line.set_data(x, y)

# результат обязательно присваивать переменной
anim =animation.FuncAnimation(fig,redraw,frames=100,interval=50)
plt.show()
```

Три различных кадра анимации показаны ниже.



Вначале загружаются необходимые модули и создается графическое окно `fig` с графической областью `ax`. Затем рисуется «пустая» ломаная. Она не имеет данных, но для нее сразу устанавливается толщина линии. Функция `plot` возвращает объект этой линии `line` класса `Line2D`. Напомним, что если кривых строится несколько, то функция `plot` возвращает кортеж (или список) таких объектов, например,

```
line1, line2 = plot(x1, y1, x2, y2)
```

Если возвращаемый объект один, то в инструкции

```
line, = ax.plot(...)
```

после имени `line` нужно ставить запятую (признак кортежа). Используя методы `set_linestyle()`, `set_marker()`, `set_drawstyle()` объекта `line`, можно менять стили оформления кривой и маркеров.

Функция `redraw(i)` рисует $i-$й кадр. Для рисования кривой используется объект `line`, созданный ранее. У него есть метод `line.set_data(x,y)`, который меняет его данные (массивы `x` и `y` координат вершин ломаной).

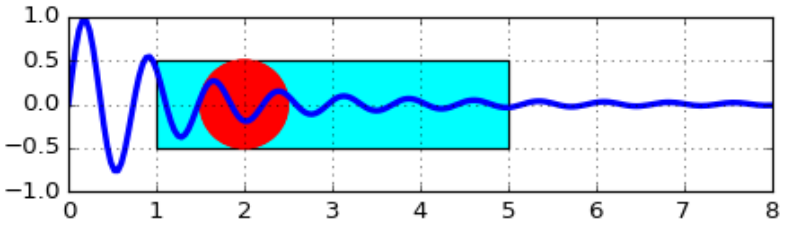
Функция `animation.FuncAnimation(...)` первым аргументом принимает графическое окно `fig`, в котором создается анимация. Второй аргумент является именем функции `redraw`, которая рисует кадры. Затем мы указываем количество кадров и интервал времени между ними.

Для добавления фона (если в функции `FuncAnimation` не используется опция `blit=True`) можно создать функцию инициализации анимации. Добавьте в предыдущий код следующие инструкции.

```
from matplotlib.patches import Circle,  Rectangle
. . .
def initpict():
    rect=Rectangle((1, -0.5),4, 1, facecolor='cyan',fill=True)
    ax.add_patch(rect)  # добавление прямоугольника на графическую область
    circle = Circle((2, 0), 0.5,  color='r')
    ax.add_artist(circle) # добавление круга на графическую область
. . .
ax.set_aspect('equal')
```

И измените следующую инструкцию, добавив вызов функции `initpict()`.

```
anim=animation.FuncAnimation(fig, redraw,init_func=initpict,
                                        frames=100, interval=50)
```



В нашем примере функция `initpict()` не должна ничего возвращать.

В результате анимация будет содержать рисунок фона, состоящий, в данном случае, из прямоугольника и круга. Однако, если мы хотим иметь неизменный рисунок фона, то его можно создать проще, используя обычные графические инструкции до вызова функции `FuncAnimation(...)`.

■

В предыдущем примере мы использовали объект, возвращаемый функцией `plot`. Все графические функции matplotlib возвращают графические объекты различных классов и эти объекты можно впоследствии использовать для изменения их свойств. В частости функция `plot` возвращает список объектов класса `matplotlib.lines.Line2D`. Например, следующие инструкции строят кривую (ломаную), ссылка на которую помещается в переменную `line`. Затем обращение к методу объекта `line` изменяет его атрибут сглаживания.

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # выключение сглаживания
```

Здесь `plt` синоним имени модуля `matplotlib.pyplot`.

Используя функцию `setp()`, можно менять свойства всего списка объектов, возвращаемых графической функцией, Например,

```
lines = plt.plot(x1, y1, x2, y2)
plt.setp(lines, color='b', linewidth=3.0)
```

Функции `setp()` и `getp()` модуля `matplotlib.pyplot` предназначены для установки и чтения свойств графических объектов. Функция `setp()` обрабатывает один объект или целый список объектов, которые ей передаются первым аргментом.

Обычно, возвращаемые графические объекты имеют метод, с помощью которого можно поменять данные, используемые для их построения. В предыдущем примере для объекта `line` класса `Line2D` использовался метод `line.set_data(x,y)`. Он заменяет массивы `x` и `y` координат вершин ломаной и, тем самым, меняет форму ломаной/кривой.

Объект, возвращаемый функцией `FuncAnimation`, должен существовать постоянно все время работы анимации. Поэтому функция `FuncAnimation` должна возвращать значение в глобальную переменную, которая будет представлять ссылку на этот объект.

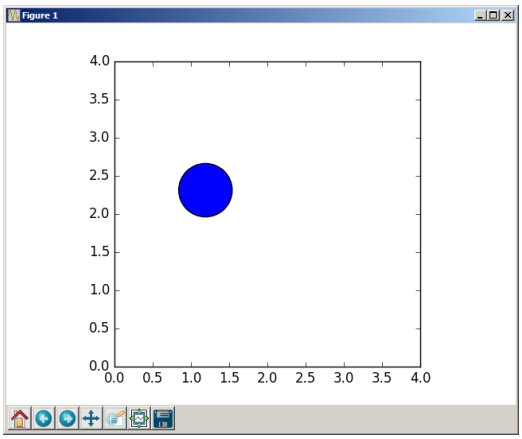
Важным аргументом функции `FuncAnimation(...)` может быть опция `blit`. Установка опции `blit=True` гарантирует, что только изменяемая часть битового массива, представляющего изображение, будет перерисовываться. Это ускоряет и улучшает качество анимации.

Если используется опция `blit=True`, то функции `func(...)` и `init_func(...)` должны возвращать последовательность графических объектов (iterable of artists). Это необходимо для того, чтобы сообщить функции `FuncAnimation`, какие графические объекты меняются. Тогда они рисуются в памяти (вне экрана) и только потом очень быстро переносятся в графическую область. Если объекты непосредственно рисуются в графическом окне, то возникает «торможение» и может возникнуть мерцание. Обычно функция инициализации графики (в примере это `initpict`) возвращает те объекты, которые должны присутствовать только на стартовом кадре, а функция анимации (в примере это `redraw`) возвращает только объекты, которые перерисовываются в каждом кадре.

Пример. *Движение бильядного шара*. В следующем примере по прямоугольному полю движется круг (модель бильярдного шара), который отражается от бортов (границ) прямоугольника. В программе используется функция `initpict`, которая возвращает объект, представляющий круг. Этот же объект возвращает функция `redraw`, сообщая функции анимации, что его изображение меняется и требует перерисовки.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.patches import Circle,Rectangle
def initpict():
    pc.center=(x0,y0)
    ax.add_patch(pc)
    return pc,   # return [ ] оставляет стартовый шар нарисованным

def redraw(i):
    global x,y,dx,dy
    if x+r>rbod:    x=rbod-r; dx=-dx
    elif x-r<lbod:  x=r; dx=-dx
    else:           x=x+dx
    if y+r>ubod or y+dy+r>ubod:    y=ubod-r; dy=-dy;
    elif y-r<dbod or y-r+dy<dbod:  y=r; dy=-dy
    else: y=y+dy
    pc.center=(x,y)
    return pc,

fig = plt.figure(facecolor='white')
fig.set_dpi(100)
rbod=4; lbod=0; ubod=4; dbod=0   # границы области
ax = plt.axes(xlim=(lbod, rbod), ylim=(dbod, ubod) )
ax.set_aspect('equal')
# начальное положение и радиус шара
x0=3; y0=1; r=0.35;
dx=0.03; dy=0.0125  # вектор начальной скорости (направления)
x=x0; y=y0
```

```
pc = plt.Circle((x, y), r, fc='b')

anim =animation.FuncAnimation(fig, redraw,
                    init_func=initpict, frames=400, interval=20,blit=True)
plt.show()
```



Повторим еще раз. Если используется опция `blit=True`, то функции `initpict` и `redraw` должны возвращать последовательность графических объектов или пустой список. При этом, если функция `init_func()`возвращает пустой список, то создаваемые ею графические объекты будут присутствовать на всех кадрах.

■

В предыдущем примере функция `redraw` в качестве аргумента принимала номер кадра. Однако, она может иметь больше аргументов. Тогда их ей можно передавать через опцию `fargs` функции `FuncAnimation`. Например, пусть функция `redraw`, кроме номера кадра, вторым аргументом принимает ссылку на объект `line` ломаной, которую она перерисовывает. Передачу ей такого аргумента можно выполнить следующим набором команд:

```
def redraw(frameNum, line):
     # использует аргумент line для изменения данных ломаной.
     . . .
line, = ax.plot([], [])
anim = animation.FuncAnimation(fig, redraw, fargs=(line,),
                                                    interval=20)
```

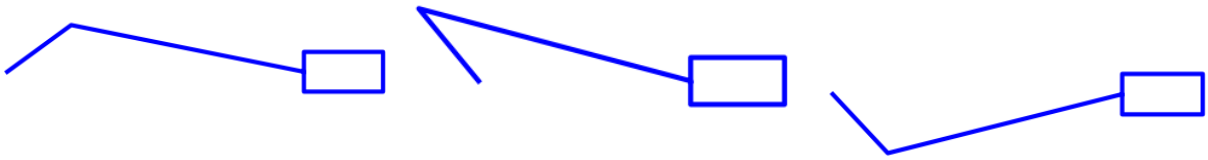
Пример. *Анимация движения кривошипно – шатунного механизма.*

Кривошипно – шатунный механизм является устройством, которое вращательное движение преобразует в поступательное или наоборот. Смоделируем движение этого механизма, изобразив его в виде ломаной.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.patches import Circle,Rectangle
def redraw(i, lin):
   alph=np.pi*i/50      # угол поворота кривошипа в i-ом кадре
   # приращения координат узлов ломаной
```

```
    dx=np.array([0,a*np.cos(alph),np.sqrt(b**2-a**2*np.sin(alph)**2),
                                                    0,dl,0,-dl, 0])
    dy=np.array([0,a*np.sin(alph),-a*np.sin(alph),-dh,0,2*dh,0,-dh])
    # массивы координат узлов ломаной
    x=np.cumsum(dx)
    y=np.cumsum(dy)
    lin.set_data(x, y)

fig = plt.figure(facecolor='white')
fig.set_dpi(100)
a=1; b=3; dl=1; dh=0.25  # геометрические размеры
ax = plt.axes(xlim=(-1.2, 5.2), ylim=(-1.5, 1.5))
line, = ax.plot([ ], [ ], lw=3)
ax.set_aspect('equal')
ax.set_xticks([])  # удаляем засечки и метки с оси X
ax.set_yticks([])  # удаляем засечки и метки с оси Y
anim =animation.FuncAnimation(fig, redraw, fargs=(line,),
                                          frames=100, interval=50)
plt.show()
```

На следующем рисунке показаны три кадра анимации в последовательные моменты времени.



Заметим, что использование второго аргумента функции `redraw(i,lin)` в данном примере необязательно.

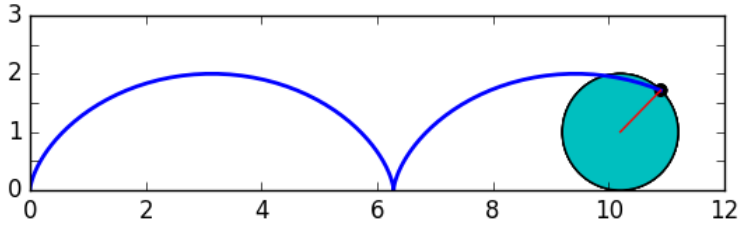
■

Приведем еще несколько примеров использования функции `FuncAnimation`.

Пример. *Циклоида*. Траектория фиксированной точки окружности, которая катится по прямой, называется циклоидой. Если радиус окружности равен 1 и скорость движения ее центра $v=1$, то параметрическое уравнение циклоиды будет иметь вид $x(t)=t-\sin t$, $y(t)=1-\cos t$. Следующий код строит катящуюся окружность, движение фиксированной точки на окружности, и заметаемую точкой траекторию (циклоиду).

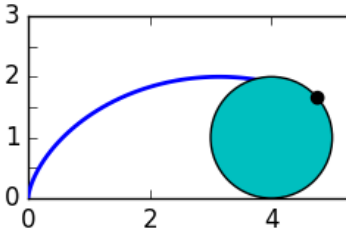
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.patches import Circle

def initpict():
    line1.set_data([], [])
    line2.set_data([], [])
    pc.center=(xC,yC)
    ax.add_patch(pc)
    pp.center=(xP,yP)
```

```
    ax.add_patch(pp)
    return pc,pp,line1,line2

def redraw(i):
    global xC, xP, yP
    n=i/10
    t=np.linspace(0,n,100)
    x=t-np.sin(t)
    y=1-np.cos(t)
    line1.set_data(x, y)
    xC=n
    pc.center=(xC,yC)
    xP=n-np.sin(n)
    yP=1-np.cos(n)
    pp.center=(xP,yP)
    line2.set_data([n,xP], [1,yP])
    return pc,pp,line1,line2

fig = plt.figure(facecolor='white')
fig.set_dpi(100)
a=1; b=3; dl=1; dh=0.25
ax = plt.axes(xlim=(0, 12), ylim=(0, 3))
ax.set_yticklabels(['0',' ','1',' ','2',' ','3'])     # меняем текст меток верт. оси
line1, = ax.plot([], [] ,lw=2)
line2, = ax.plot([], [] ,'r')
xC=0;yC=1; xP=0; yP=0
pc = plt.Circle((xC, yC), 1, fc='c')
pp = plt.Circle((xP, yP), 0.1, fc='k')
ax.set_aspect('equal')

anim =animation.FuncAnimation(fig, redraw, init_func=initpict,
                                    frames=120, interval=50, blit=True)
plt.show()
```



Обратите внимание на то, что при использовании опции `blit=True` функции `initpict` и `redraw` возвращают кортеж (или список) графических объектов. Функция `initpict` возвращает те объекты, которые должны присутствовать только на стартовом кадре. А функция `redraw` должна возвращать только объекты, которые перерисовываются в каждом кадре. Кроме того, порядок возвращаемых (перерисовываемых) объектов существенен. Попробуйте заменить инструкцию `return pc,pp,line1,line2` функции `redraw` на

инструкцию `return line1,line2, pc,pp,` и на кадрах анимации не будут видны участки траектории внутри круга и отрезок радиуса.



■

Пример. Бегущие волны в водоеме конечной глубины, неограниченном в горизонтальном направлении, можно смоделировать уравнениями

$$U = a\sin(wt - k\,x)\left(e^{k\,y} + e^{-2k\,h}e^{-k\,y}\right),\ V = a\cos(wt - k\,x)\left(e^{k\,y} - e^{-2k\,h}e^{-k\,y}\right)$$

где U и V мгновенные значения смещений частицы жидкости с равновесными координатами x, y в направлениях осей x и y. Здесь предполагается, что дно водоема находится на уровне $y = -h$, а поверхность – на уровне $y = 0$. Точки поверхности жидкости будут иметь координаты $X = x + U(x,0), Y = 0 + V(x,0)$. В примере нам удобнее считать, что дно водоема находится на уровне $y = 0$, а поверхность – на уровне $y = h$. Поэтому координаты точек поверхности жидкости в момент времени t будут равны:

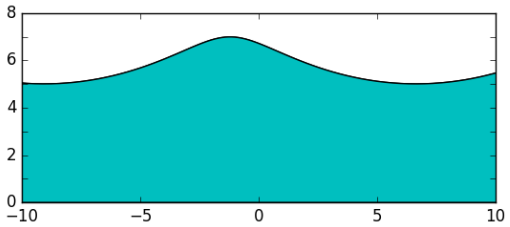
$$X = x + a\sin(wt - k\,x)\left(1 + e^{-2k\,h}\right),\ Y = h + a\cos(wt - k\,x)\left(1 - e^{-2k\,h}\right).$$

В следующем коде мы моделируем движение жидкости в соответствии с этими уравнениями. Двумерную волну изображаем с помощью залитой области между кривой (X,Y) и осью Ox (дном водоема).

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
def xs(u,t):
    return u+a*np.sin(w*t-k*u)*(1+np.exp(-2*k*h))
def ys(u,t):
    return a*np.cos(w*t-k*u)*(1-np.exp(-2*k*h))
def redraw(i):
    t=i/10  # момент времени текущего кадра
    u=np.linspace(-2*L,2*L,100)
    X=xs(u,t)
    Y=ys(u,t)+h
    # начальная и конечная точки области заливки должны находиться на оси x!
    X=np.insert(X,0,X[0])
    Y=np.insert(Y,0,0)
    X=np.append(X,X[-1])
    Y=np.append(Y,0)
    # для изменение данных передается массив из двух столбцов,
    # например, ptch.set_xy([[0,0],[3,-5],[6,-9],[9,0]])
    # преобразуем одномерные массивы координат в двумерный массив pp
    XY=np.vstack((X,Y))
    pp=XY.T
```

```
    ptch.set_xy(pp)     # меняем данные Polygon–а

L=6; w=1; a=1; h=6; k=0.4     # параметры задачи
fig = plt.figure(facecolor='white')
fig.set_dpi(100)
ax = plt.axes(xlim=(-2*(L-1), 2*(L-1)), ylim=(0, h+2))
ax.set_yticklabels(['0','','2','','4','','6','','8'])  # подписи меток верт. оси
ptch, = ax.fill([0,5,10], [0,0,0],'c' )  # создаем пустой объект;
                    # fill возвращает список/кортеж объектов класса Polygon
ax.add_patch(ptch)
ax.set_aspect('equal')
anim =animation.FuncAnimation(fig, redraw,  frames=120, interval=50)
plt.show()
```

Один из кадров анимации показан на следующем рисунке.



■

Заметим, что если опция `init_func` в функции `FuncAnimation` не используется, то для создания стартового кадра анимации применяется первый кадр функции `redraw` . Точнее вызывается функция `redraw(0)` с нулевым номером кадра. В результате вызов `redraw(0)` выполняется два раза подряд. Чтобы этого не происходило можно создать пустую функцию `initpict` и использовать ее в опции `init_func=initpict` функции `FuncAnimation`.

Функция `FuncAnimation` не ограничена двумерной графикой. Кадры анимации могут состоять из трехмерных рисунков.

Пример. Лента Мебиуса – простейшая неориентируемая поверхность. Она получается движением и вращением отрезка прямой вдоль замкнутой пространственной кривой. Пусть эта кривая будет окружностью радиуса `R`, а `n` обозначает количество полуоборотов отрезка при обходе кривой. Вдоль направляющей окружности движется центр отрезка. Уравнение поверхности можно записать в параметрическом виде

$$x(u,v)=\left(R+v\cos\left(\frac{nu}{2}\right)\right)\cos u$$

$$y(u,v)=\left(R+v\cos\left(\frac{nu}{2}\right)\right)\sin u$$

$$z(u,v)=v\sin\left(\frac{nu}{2}\right)$$

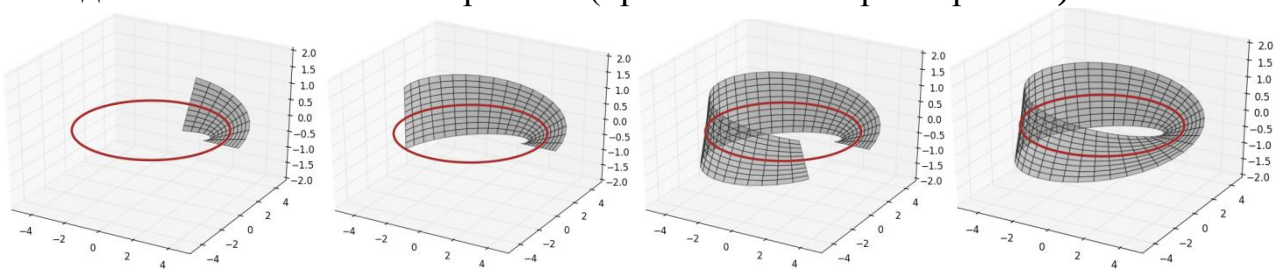
Пусть `H` обозначает ширину ленты (длину отрезка). При нечетном `n` получается неориентированная поверхность (лента Мебиуса).

Следующий код строит анимацию движения отрезка и заметаемую им поверхность.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
# координатные функции
xs=lambda u,v: (R+v*np.cos(n*u/2))*np.cos(u)
ys=lambda u,v: (R+v*np.cos(n*u/2))*np.sin(u)
zs=lambda u,v:  v*np.sin(n*u/2)

def initpict():
    global u
    u=np.linspace(0,du,1)  # очистка вектора u перед стартовым кадром

def redraw(i):      # рисование i-го кадра
    global u
    u=np.append(u,(i+1)*du)       # добавление точки к вектору u
    U,V=np.meshgrid(u,v)
    X=xs(U,V)
    Y=ys(U,V)
    Z=zs(U,V)
    ax.clear()
    # после очистки восстанавливаем видимые размеры графической области
    ax.set_xlim3d(xyLeft, xyRight)
    ax.set_ylim3d(xyLeft, xyRight)
    ax.set_zlim3d(-H,H)
    ax.plot(xc,yc,zc,color='brown',linewidth=3) # направляющая окружность
    ax.plot_surface(X, Y, Z, rstride=1,cstride=1,color='0.5',alpha=0.4)

R=4;H=2;n=1;
NumFrames=40    # количество кадров
du=2*np.pi/NumFrames
xyRight=R+1; xyLeft=-xyRight;
fig = plt.figure(facecolor='white')
fig.set_dpi(100)
ax=Axes3D(fig,xlim=(xyLeft,xyRight),ylim=(xyLeft,xyRight),zlim=(-H,H))
# координаты точек направляющей окружности одинаковые для всех кадров
t=np.linspace(0,2*np.pi,100)
vt=np.zeros(np.size(t))
xc=xs(t,vt)
yc=ys(t,vt)
zc=zs(t,vt)
# массив параметра v поверхности одинаков для всех моментов времени
v=np.linspace(-H/2,H/2,11)

anim =animation.FuncAnimation(fig, redraw,  init_func=initpict,
```

```
                frames=NumFrames, interval=20, repeat=True,blit=False)
plt.show()
```

На следующем рисунке показано несколько кадров анимации в последовательные моменты времени (при значении параметра `n=1`).



Особенность этого кода состоит в том, что в каждом кадре, создаваемом функцией `redraw`, добавляется только одна «вертикальная полоска» поверхности. Для этого перед началом анимации функция инициализации `initpict()` очищает вектор `u`, который в результате будет содержать только нулевое значение. Функция `redraw` в каждом новом кадре к вектору `u` добавляет один элемент `(i+1)·du`, больший предыдущего на величину $\mathrm{d}u = 2\pi / \mathrm{NumFrames}$. Следующие четыре команды создают массивы U,V значений параметров и массивы X, Y, Z координат точек участка поверхности ленты. Но прежде, чем ленту рисовать, инструкция `ax.clear()` очищает графическую область, а последующие три инструкции восстанавливают для нового изображения прежние размеры видимой области. После этого рисуется направляющая окружность, а затем рисуется участок ленты $0 \le u \le (i+1)\mathrm{d}u$.

Если очистку графического окна не делать, то в графической области после создания *i*-го кадра будет нарисовано *i* налагающихся друг на друга кусков ленты разной длины. Сверху будет нарисован самый длинный, но в памяти будут присутствовать все. Это значительно «утяжеляет» рисунок, который будет сложнее «крутить».

Первый кадр анимации строится инструкцией `redraw(0)`, которая вызывается из функции `FuncAnimation`. Она рисует одну полоску $0 \le u \le \mathrm{d}u$. Последующий вызов `redraw(1)` рисует две полоски ($0 \le u \le 2\mathrm{d}u$) и т.д.

Протестируйте работу программы при `n=2` и `n=3`.

■

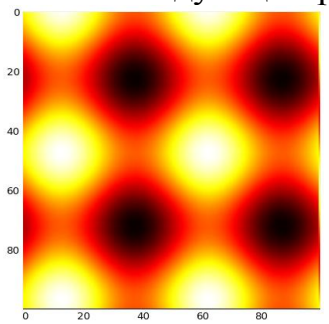
Иногда кадры анимации содержат изображения матриц, взятых из файла или сгенерированных в коде программы. Ниже приведен код, создающий анимацию изображений матриц, которые «рисуются» функцией `imshow`.

Пример.

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

fig = plt.figure(facecolor='white')
f=lambda x,y: np.sin(2*x)+np.sin(2*y)
x = np.linspace(0, 2 * np.pi, 100)
```

```
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)
im = plt.imshow(f(x, y), cmap="hot", animated=True)

def redraw(*args):
    global x, y
    x += np.pi / 26
    y += np.pi / 20.
    im.set_array(f(x, y))
    im.set_cmap('hot')
    return im,

ani = animation.FuncAnimation(fig, redraw, interval=50, blit=True)
plt.show()
```

Один из кадров анимации показан на следующем рисунке.



Напомним, что значит аргумент функции со звездочкой: `redraw(*args)`. Если перед аргументом в определении функции указан символ `*` (звездочка), то функции можно передавать произвольное количество аргументов и они будут храниться в кортеже `args`.

В приведенном коде функция `redraw`, вызываемая для рисования кадров, не использует свои аргументы. Но она меняет глобальные массивы `x` и `y`, которые в методе `set_array` используются для корректировки матрицы данных объекта `im`, созданного ранее функцией `imshow`.

Обратите также внимание на то, как первоначально создается массив `y`.

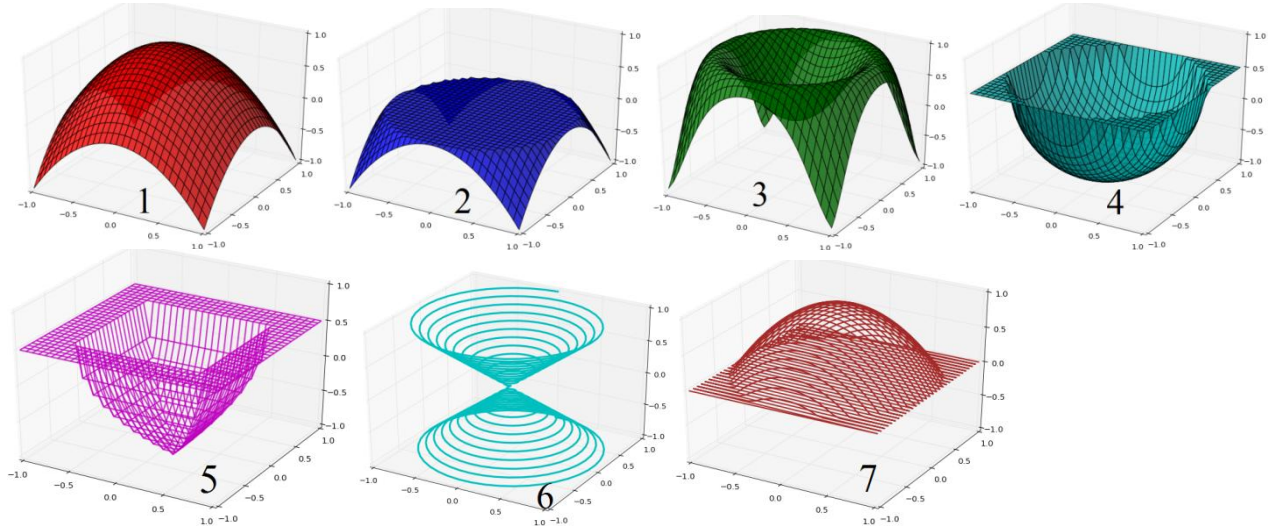
```
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)
```

В методе `reshape(-1,1)` минус единица означает количество элементов по последней размерности. В результате создается двумерный массив с одним эдементом в каждой строке и количеством строк, равным количеству столбцов (элементов) одномерного массива.

■

Другой функцией, используемой для создания анимации, является функция `ArtistAnimation`. Она имеет сходные с функцией `FuncAnimation` аргументы, и отличие состоит только во втором. Он теперь является списком графических объектов, представляющих кадры анимации. Ниже приведен пример, использующий эту функцию.

Пример. *Последовательная анимация рисунков*. В примере рисуются трехмерные объекты, показанные ниже. Из них создается список «кадров», которые затем по очереди отображаются в графической области.



Эти рисунки являются графиками поверхностей и пространственных кривых. Они строятся различными функциями, возвращающими значения в соответствующие переменные. Каждая из функций `plot_surface` возвращает объект класса `mpl_toolkits.mplot3d.art3d.Poly3DCollection` (коллекция 3D многоугольников), которые показанным в примере способом должны быть включены в список. Функция `plot_wireframe` возвращает объект класса `Line3DCollection`, а объекты, возвращаемые функциями `plot` и `plot3D`, являются пространственными кривыми. Все они включаются в анимационный список графических объектов похожим способом. Однако обратите внимание на то, как возвращаются объекты `p6` и `p7` (с запятой после имени объекта).

Код программы приведен ниже. Большая его часть состоит из инструкций создания объектов рисунков, которые затем помещаются в список кадров, передаваемый для анимации в функцию `ArtistAnimation`.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from mpl_toolkits.mplot3d import Axes3D
import copy

fig = plt.figure(facecolor='white')
ax=Axes3D(fig, xlim=(-1, 1), ylim=(-1, 1), zlim=(-1,1.0))
# поверхность 1
x = np.linspace(-1,1,31)
y = np.linspace(-1,1,31)
X,Y=np.meshgrid(x,y)
Z=1-X**2-Y**2
ZZ=copy.deepcopy(Z)   # запоминаем Z для 7-й поверхности
p1=ax.plot_surface(X,Y,Z,rstride=1,cstride=1,color='r', alpha=0.8)
# поверхность 2
Z=1-X**2-Y**2;
Z[Z>0]=0
p2=ax.plot_surface(X,Y,Z,rstride=1,cstride=1,color='b', alpha=0.8)
# поверхность 3
```

```
a=2.4; Z=np.sin(4*np.pi*(X**2+Y**2)/a**2)
p3=ax.plot_surface(X,Y,Z,rstride=1,cstride=1,color='g', alpha=0.8)
# поверхность 4
Z=1-X**2-Y**2;
Z[Z<0]=0
Z=0.5-1.5*np.sqrt(Z)
p4=ax.plot_surface(X,Y,Z,rstride=1,cstride=1,color='c', alpha=0.8)
# поверхность 5
Z=1.3-np.abs(X-Y)-np.abs(X+Y)
Z[Z<0]=0
Z=0.5-1.5*np.sqrt(Z)
p5=ax.plot_wireframe(X,Y,Z,rstride=1,cstride=1,color='m',
                                                    alpha=0.8, lw=2)
# объект 6 (кривая)
theta = np.linspace(-4 * np.pi, 4 * np.pi, 1000)
z = np.linspace(-1, 1, 1000)
r=np.abs(z)
x = r * np.sin(6*theta)
y = r * np.cos(6*theta)
p6,=ax.plot(x, y, z,c='c',lw=3)
# объект 7 (кривая в форме поверхности)
# X,Y,Z первой поверхности
ZZ[ZZ<0]=0
p7,=ax.plot3D(np.ravel(X),np.ravel(Y),np.ravel(ZZ),lw=2,c='brown')
# список объектов для анимирования
piclist=[(p1,),(p2,),(p3,),(p4,),(p5,),(p6,),(p7,)]

anim = animation.ArtistAnimation(fig, piclist, interval=1000,
                                          blit=True, repeat_delay=100)
plt.show()
```

Обратите внимание на опцию `repeat_delay=100`. Без нее и с включенной опцией `blit=True` первый кадр (первая поверхность) на повторных циклах анимации не появляется (или незаметен). Если вы хотите, чтобы анимация не повторялась многократно, то используйте опцию `repeat=False`.

■

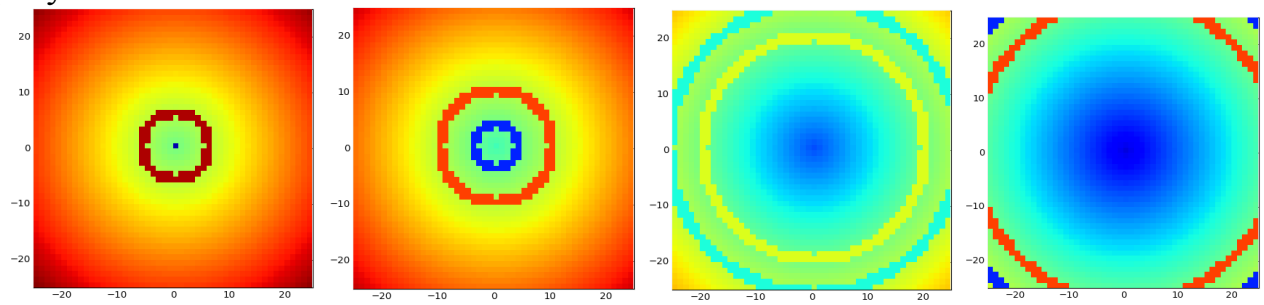
В следующем примере кадры анимации состоят из изображений матриц, которые «рисуются» функцией `pcolor`.

Пример. *Анимация изображений матриц*.

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

fig = plt.figure(facecolor='white')
ax = plt.axes(xlim=(-25, 25), ylim=(-25, 25),aspect='equal')
x = np.arange(-25, 26)
```

```
y = np.arange(-25, 26)
X,Y=np.meshgrid(x,y)
ZZ=np.sqrt(X**2+Y**2)
imlist = [ ]
for i in np.arange(36):
    Z=copy.deepcopy(ZZ)-i
    Z[np.logical_and(Z<0,Z>-2)]=(i-17)*2
    Z[np.logical_and(Z<6,Z>4)]=(17-i)*2
    imlist.append((plt.pcolor(x, y, Z , norm=plt.Normalize(-35, 35)),))
anim = animation.ArtistAnimation(fig, imlist, interval=50,
                                 repeat_delay=100,  blit=True)
plt.show()
```

Несколько кадров, созданных этой программой, приведены на следующем рисунке.



Здесь список `imlist` составляется из графических объектов, возвращаемых функцией `pcolor`. Она вызывается в одном из двух форматов:

```
pcolor(C[,...])
pcolor(X, Y, C[, ...])
```

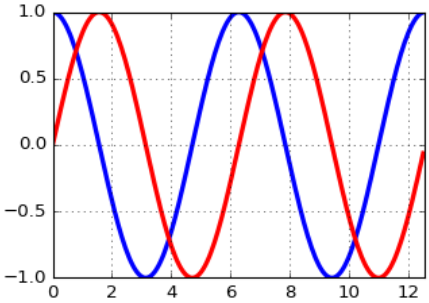
Здесь `C` – массив значений, которые представляют цвета. Если используются массивы X и Y, то они определяют (x, y) координаты закрашиваемых четырехугольников, вершины которых расположены в точках `(X[i,j],Y[i,j])`, `(X[i,j+1],Y[i,j+1])`, `(X[i+1,j],Y[i+1,j])`, `(X[i+1,j+1], Y[i+1,j+1])`. При этом индекс столбца ссответствует `x` координате, а индекс строки – `y` координате. Размеры массивов `X` и `Y` должны быть на единицу больше соответствующих размеров матрицы `C`. Однако, если размеры одинаковы, то последняя строка и столбец матрицы `C` игнорируются. В качестве `x` и `y` массивов допустимо использовать одномерные массивы. Аргумент `norm=plt.Normalize(-35,35)` функции `pcolor` используется для нормировки данных (приведению их к диапазону [0, 1]).

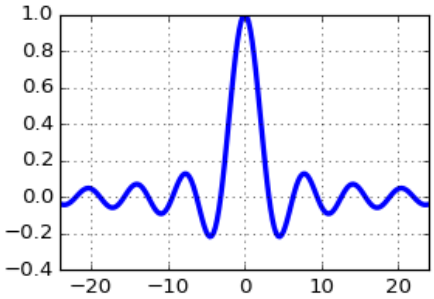
4.4 Графические функции модуля mpmath

В набор научных библиотек входит пакет `mpmath`, который позволяет выполнять вычисления стандартных математических функций (элементарных и специальных) с произвольной точностью. В нашем пособии мы этой темы не будем касаться. Но в этот модуль включено несколько функций, которые умеют строить графики. Их особенность состоит в том, что для построения графиков функций требуются только их аналитические представления.

Функция `mpmath.plot(...)` строит график одной или нескольких функций одной вещественной переменной. При этом используется идентификатор имени функции (без аргументов).

```
from mpmath import *
plot([cos,sin],[0,4*pi])  # следующий рисунок слева
```

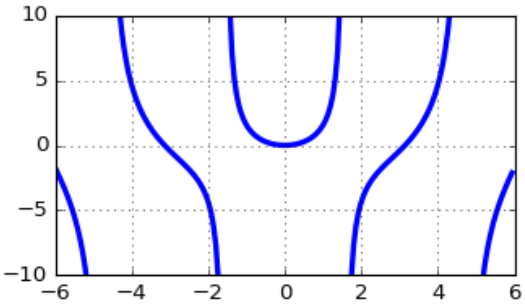





```
f=lambda x: sin(x)/x
plot(f, [-24, 24])        # предыдущий рисунок справа
```

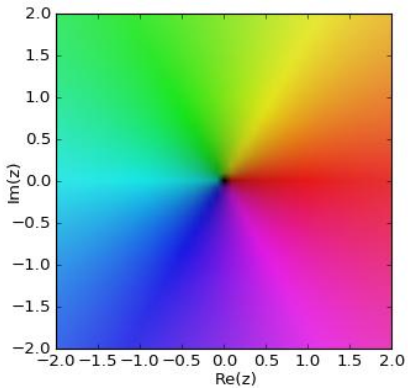
Имеются опции, управляющие пределами изменения значений по осям, а также опция `singularities`, которая указывает особые точки графика.

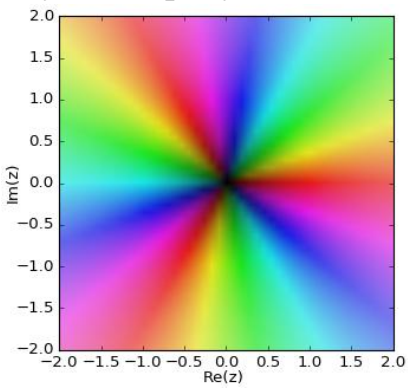
```
f=lambda x: tan(x)*x
plot(f,xlim=[-6,6],ylim=[-10,10],singularities=[-3*pi/2,-pi/2, pi/2,3*pi/2])
```



Функция `mpmath.cplot(f,re=[-5,5],im=[-5,5],points=2000, color=None,...)` строит график комплекснозначной функций `f` над прямоугольной областью комплексной плоскости, задаваемой парой интервалов `re` и `im`. По умолчанию аргумент функции представляется цветом в `hue` палитре, а модуль значения представляется яркостью.

```
cplot(lambda z: z, [-2,2], [-2,2])        # следующий рисунок слева
```

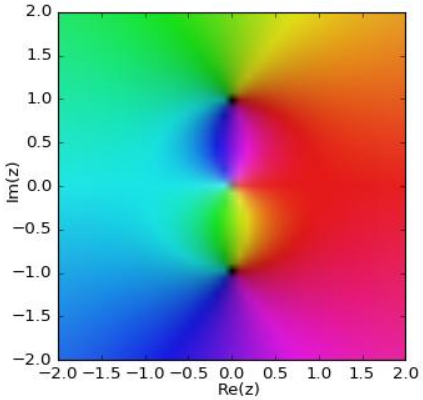


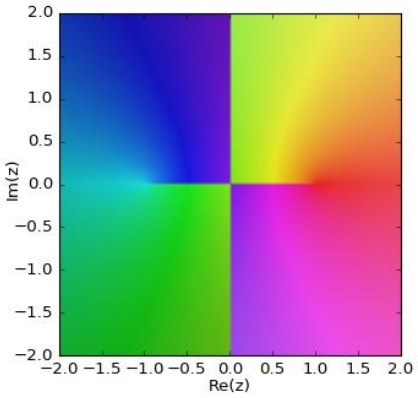



```
cplot(lambda z: z**3, [-2,2], [-2,2])     # предыдущий рисунок справа
```

Вот как выглядит график функции Жуковского $f(z)=\frac{1}{2}\left(z+\frac{1}{z}\right)$.

```
cplot(lambda z: (z+1/z)/2, [-2,2], [-2,2])      # следующий рисунок слева
```



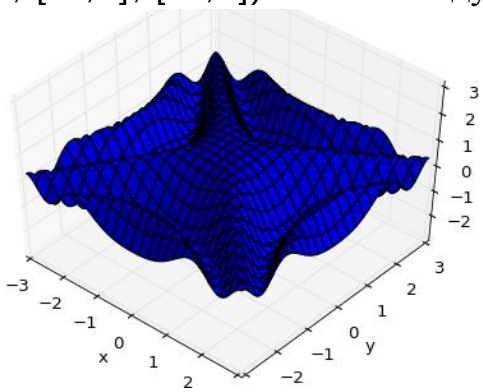


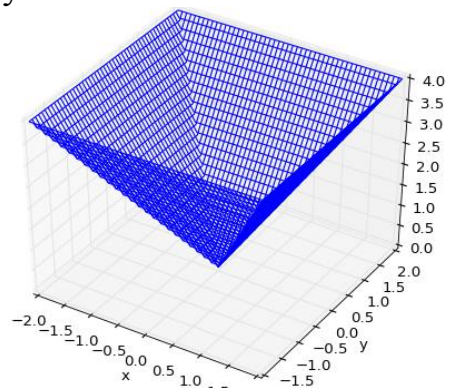
На предыдущем рисунке справа показан график функции $z = w + \sqrt{w^2 - 1}$ обратной к функции Жуковского.

```
cplot(lambda w: w+sqrt(w**2-1),[-2,2],[-2,2],points=20000)
```

Функция `mpmath.splot(f,u=[-5,5],v=[-5,5],points=100, keep_aspect=True, wireframe=False,...)` строит график (поверхность) функции двух переменных.

```
f=lambda x, y: cos(x**2-y**2)*exp(-(x**2+y**2)/16)
splot(f, [-3,3], [-3,3])          # следующий рисунок слева
```

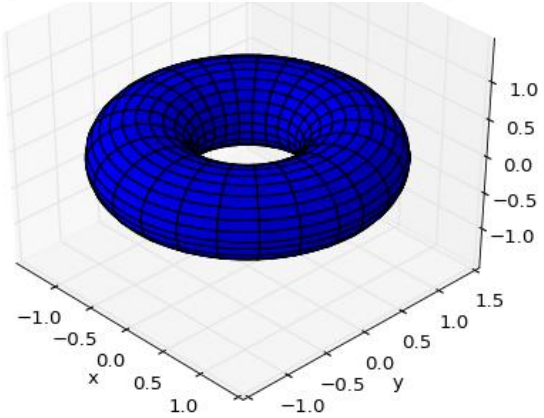


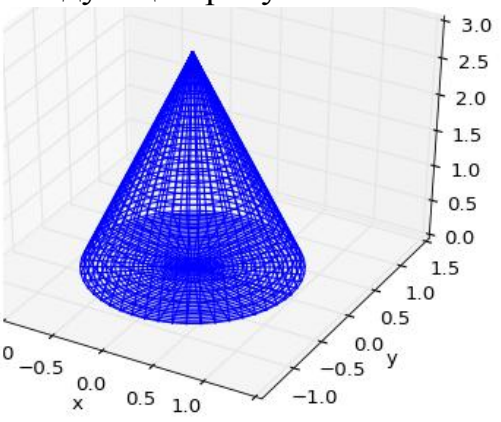



```
f=lambda x, y: abs(x-y)+abs(x+y)
splot(f, [-2,2],[-2,2],points=200,wireframe=True) # пред. рисунок справа
```

Если `f` возвращает список из трех функций, то функция `splot()` рассматривает их как компоненты параметрического уравнения поверхности.

```
r=0.5
f=lambda ph,th: [(1+r*cos(ph))*cos(th),(1+r*cos(ph))*sin(th),r*sin(ph)]
splot(f,[0,2*3.1415],[0,2*3.1415])      # следующий рисунок слева
```





```
R,H=1,3          # радиус и высота конуса
fx=lambda t,tau: R*cos(tau)*(abs(t)-2*abs(t-1)+abs(t-2))/2
fy=lambda t,tau: R*sin(tau)*(abs(t)-2*abs(t-1)+abs(t-2))/2
fz=lambda t,tau: H*(1+abs(t-1)-abs(t-2))/2
f=lambda t,tau: [fx(t,tau),fy(t,tau),fz(t,tau)]
splot(f,u=[0,2],v=[0,2*3.141592],wireframe=True,points=200)
```

В этом примере мы по параметрическим уравнениям нарисовали конус, который показан на предыдущем рисунке справа.

5. Символьные вычисления

5.1 Основы символьных вычислений

Python может выполнять арифметические операции не только как калькулятор. После импортирования функций модуля SymPy, он может выполнять символьные преобразования.

```
>>> from sympy import *
```

Поскольку переменные в Python не имеют никакого значения до тех пор, пока им не будет присвоено значение, то просто использовать их в символьных вычислениях нельзя. Поэтому перед использованием переменные и функции должны быть объявлены как символьные. Это отличает Python SymPy от таких программ как Mathematica или Maple, где переменные, если им не присвоено никаких значений, автоматически рассматриваются как символьные. Объявление символьных переменных выполняется функцией symbols. Например

```
>>> x,y,a,b = symbols('x y a b')
```

Здесь мы создали четыре символьных переменных. Предыдущие значения переменных затираются.

Символьные выражения конструируются из символьных переменных

```
>>>f=a**3*x + 3*a**2*x**2/2 + a*x**3 + x**4/4
```

Переменная **f** автоматически становится символьной.

Имеется другая функция var(), которая также создает символьные переменные, помещая их в глобальное пространство имен.

```
>>> var('u,v')
(u, v)
```

Разница между symbols() и var() в том, первая функция возвращает ссылку на символьный объект, которую, чтобы использовать, нужно присвоить какой – либо переменной. Вторая, без присваивания создает символьную переменную. Например, после выполнения команды >>>var('Hello') символьная переменная с именем Hello будет уже существовать.

```
>>> var('Hello')
Hello
>>> type(Hello)
<class 'sympy.core.symbol.Symbol'>
```

При создании символьных выражений следует аккуратно обращаться с числовыми константами. Иногда они могут быть проинтерпретированы как

числовые константы Python, а не SymPy. Если надо быть уверенным, что константа является символьной, поместите ее внутрь функции `S()`.

```
>>> expr = x**2 + sin(y) + S(1)/2; expr
x**2 + sin(y) + 1/2
```

Сравните

```
>>> type(1)
<class 'int'>
>>> type(S(1))        # символьная константа единица
<class 'sympy.core.numbers.One'>
```

Функция `sympy.S(expr[,...])` конвертирует произвольное выражение `expr` в тип, который можно использовать в выражениях SymPy. Например, она преобразует целые, рациональные, вещественные числа Python в аналогичные типы SymPy. Ее аргумент `expr` может быть любым объектом SymPy, объектом стандартного Python типа (int, long, float, Decimal), строкой, булевской переменной, списком, множеством или кортежем из элементов приведенных типов.

```
>>> type(S(x**2))
sympy.core.power.Pow
>>> S(x**2).subs(x,3)
9
```

Также функцию `S()` можно использовать и для объявления символьных переменных.

```
>>> p = S('p')
>>> type(p)
sympy.core.symbol.Symbol
```

Фактически инструкция `S()` является синонимом вызова функции `sympify()`, которая часто используется для преобразования строки в символьное выражение. Но, как сказано выше, ее (функцию `S()`) можно использовать также для преобразования констант Python в символьные константы.

```
>>> sympify(0.5)
0.500000000000000
>>> type(_)      # тип последнего результата
<class 'sympy.core.numbers.Float'>
```

Разница между константой Python и символьной состоит в том, что символьная постоянная может быть вычислена с любой степенью точности. Для этого можно использовать метод `симв_перем.n(кол_цифр)`. Например, следующая инструкция вычисляет переменную `z1` с процессорной точностью (примерно 15 значащих цифр).

```
>>> z1=1/3; z1
0.3333333333333333
```

Символьная константа может быть вычислена с произвольной точностью.

```
>>> z2=S(1)/3; z2
1/3
>>> z2.n(30)
0.333333333333333333333333333333
```

Сравните также следующие два выражения.

```
>>> x**(1/2)
x**0.5
>>> x**(S(1)/2)
sqrt(x)
```

Можно объявлять символьные переменные с индексом.

```
>>> x=symbols('x:5'); x      # диапазон индексов от 0 до 4
(x0, x1, x2, x3, x4)      #x[0],x[1],... явл. симв. переменными x0,x1,...
>>> x=symbols('x5:10'); x # диапазон индексов от 5 до 9
(x5, x6, x7, x8, x9)
```

Можно создать набор идущих подряд «однобуквенных» символьных имен.

```
>>> symbols('a:d')
(a, b, c, d)
```

При создании символьным переменным можно назначить тип, например, целый.

```
>>> k, m, n = symbols('k m n', integer=True)
```

Иногда без подобных дополнительных предположений почти очевидные символьные преобразования не работают.

```
>>> sqrt(x**2)   # это x, если x≥0
sqrt(x**2)
```

Функция `Symbol('имя',...)` (на самом деле это конструктор класса) позволяет наложить на символьную переменную какое–нибудь условие.

```
>>> x = Symbol('x', positive=True)
>>> sqrt(x**2)
x
```

Функция `var()` также имеет опции наложения ограничений на символьные переменные.

```
>>> var('x y', positive = True, integer = True)
```

Если вы предполагаете в текущей сессии использовать символьную математику постоянно, то можно импортировать общепринятые символьные имена из модуля `sympy.abc`.

```
>>> import sympy.abc
>>> dir(sympy.abc)
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
..., 'a','alpha','b','beta','c', 'chi', 'd','delta', 'division',
'e', 'epsilon', 'eta', 'exec_', 'f', 'g', 'gamma', 'greeks', 'h',
'i', 'iota', 'j', 'k', 'kappa','l','lamda', 'm', 'mu', 'n', 'nu',
'o', 'omega','omicron','p', 'phi', 'pi', 'print_function', 'psi',
'q', 'r', 'rho', 's', 'sigma', 'string', 'symbols', 't', 'tau',
'theta', 'u', 'upsilon', 'v', 'w', 'x', 'xi', 'y', 'z', 'zeta']
```

Если импортировать содержимое модуля `sympy.abc` командой

```
>>> from sympy.abc import *
```

то все имена, перечисленные выше (мы несколько имен пропустили, заменив их символом многоточия) можно использовать в символьных выражениях.

Однако мы не рекомендуем это делать, поскольку будут перегружены имена некоторых символьных констант, например, таких как pi или E (основание натурального логарифма).

Напомним, что удалить имя переменной из пространства имен можно командой del имя1, имя2,...

```
>>> type(x)
<class 'sympy.core.symbol.Symbol'>
>>> del x,y
>>> x
Ошибка!
```

Если вы выполняли инструкцию «from sympy.abc import *», то повторно загрузите модуль sympy.

```
>>> from sympy import *
```

Этим вы восстановите значения стандартных констант pi и E, а также имена некоторых функций.

При записи символьного выражения может автоматически выполняться его упрощение.

```
>>> a,b,c,d,x,y,z,u,v,w = symbols('a b c d x y z u v w')
>>> x - z + 23 -z- 14 + sin(pi)+2*z
x + 9
```

Для вычисления символьного выражения при некоторых конкретных значениях переменных используется метод subs(...).

```
>>> f=a**3*x + 3*a**2*x**2/2 + a*x**3 + x**4/4
>>> f.subs(a,1)
x**4/4 + x**3 + 3*x**2/2 + x
```

Здесь в выражение f вместо переменной a была подставлена единица.

Если метод subs принимает два аргумента, то они интерпретируются как subs(old,new), т.е. старый идентификатор old заменяется новым new. Аргумент метода subs() может быть последовательностью, которая должна содержать пары (old,new). В следующей команде в том же символьном выражении f выполнена двойная подстановка a=1, x=2.

```
>>> f.subs([(a,1),(x,2)])
20
>>> f=x*(1+log(x))
>>> f.subs(x, pi).evalf()
6.73786765331895
```

Подставляемый аргумент может быть словарем, чьи пары ключ:значение должны соответствовать старым и новым значениям old:new.

```
>>> f.subs({x:1})
a**3 + 3*a**2/2 + a + 1/4
```

Вместо символьной переменной подставлять можно символьное выражение.

```
>>> expr=x**2+2*x+1
>>> expr.subs(x,1/x)
1 + 2/x + x**(-2)
```

Обратим ваше внимание на следующую особенность работы с переменными (символьными и обычными переменными Python). Выполним следующий код

```
>>> x = symbols('x')
>>> expr=x+1
>>> x=2
>>> print(expr)
x + 1
```

Присваивание символьной переменной `x` значения `2` (этим меняем также тип `x`) не повлияло на значение символьного выражения `expr`. Здесь действует правило: если переменная изменилась, то созданное ранее выражение, содержащее эту переменную, не пересчитывается автоматически. Это правило срабатывает и для обычных переменных Python. Например,

```
>>> x='Hello'
>>> expr=x+'world'
>>> expr
'Helloworld'
>>> x='AAA'
>>> expr
'Helloworld'
```

Символьные вычисления также имеют отношение к выражениям, в которые входят одни числа. Например, *SymPy* может проводить вычисления с дробями и, приводя их к общему знаменателю, и получать точный ответ.

```
>>> S(1)/3+S(2)/5
11/15
```

Сравните.

```
>>> 1/3+2/5
0.7333333333333334
```

Для создания рациональных дробей можно использовать функцию `Rational(числитель,знаменатель)`. С рациональными числами математические операции выполняются без десятичного округления.

```
>>> z=Rational(1, 3)+Rational(2, 5); z
11/15
```

Для создания рациональной дроби можно также использовать функцию `Integer(...)`. Например,

```
>>> Integer(1)/Integer(3)
1/3
>>> 1/3
0.3333333333333333
>>> z=Integer(1)/Integer(3)+Rational(2, 5); z
11/15
```

Основной принцип символьных вычислений – не делать никаких приближений, кроме затребованных. Так выражение

```
>>> sqrt(12)          # функция sqrt() модуля sympy
```

будет преобразовано к виду

```
2*sqrt(3)
```

Python преобразует выражение, но оставляет в записи ответа квадратный корень и не выполняет никаких округлений. Однако, если хотя бы одно из чисел символьного выражения задано с десятичной точкой, то и результат будет приближенным

```
>>> sqrt(12.0)
3.46410161513775
```

У любого символьного объекта имеется метод `evalf(...)`(**eval**uate **f**loat), который возвращает его десятичное представление (аналогично методу `n()`).

```
>>> sqrt(12).evalf()   # функция sqrt() модуля sympy
3.46410161513775
>>> E.evalf()
2.71828182845905
```

Метод `evalf([n,...])` допускает использование аргумента, который задает точность результата (`n` = количество значащих цифр)

```
>>> sqrt(12).evalf(40)
3.464101615137754587054892683011744733886
>>> pi.evalf(20)
3.1415926535897932385
```

Функция `N(expr,args)` эквивалента команде `simplify(expr).evalf(args)`, где `simplify(expr)` означает упрощение символьного выражения `expr`.

```
>>> N(sqrt(2),30)
1.41421356237309504880168872421
```

Сравните результаты следующих команд.

```
>>> N(sqrt(2)*pi)
4.44288293815837
>>> float(sqrt(2)*pi)
4.442882938158366
>>> complex(sqrt(2)*pi)
(4.442882938158366+0j)
```

Функция `nsimplify(value,list,tolerance=1e-9)` преобразует десятичное число `value` в ближайшее рациональное или в комбинацию рациональных чисел и констант, перечисленных в списке `list`.

```
>>> nsimplify(0.33333,tolerance=0.001)
1/3
>>> nsimplify(6.28,[pi],tolerance=0.01)
2*pi
>>> nsimplify(atan(1),[pi])
pi/4
```

Следует помнить об одной особенности вещественной арифметики. Обычно она не возвращает точный результат. Рассмотрим следующий пример.

```
>>> from sympy import *
>>> one=S('one')
>>> one = cos(1)**2 + sin(1)**2
>>> one.evalf()        # равно 1
1.00000000000000
```

Но

```
>>> (one-1).evalf()    # должно быть равно 0
-0.e-124
```

Если вы знаете, что результат содержит погрешность вычислений, то ее можно удалить с помощью опции `chop=True` метода `evalf()`. В этом случае очень маленькое значение вещественной или мнимой части результата заменяется нулем.

```
>>> (one - 1).evalf(chop=True)
0
```

Помните, что выражение, записанное в виде строки, отличается от символьного выражения. Функция `sympify` (не путать с `simplify`), о которой говорилось выше, используется для преобразования строки в символьное выражение.

```
>>> str = "x**4-8*x**2+16"
>>> expr = sympify(str)
>>> expr
x**4-8*x**2+16
>>> expr.subs(x, 2)
0
>>> factor(expr)        # разложение на множители символьного выражения
(x-2)**2*(x+2)**2
```

Еще одним приемом, о котором следует знать, это способ извлечения частей символьных выражений.

```
>>> x,y,z=symbols('x y z')
>>> z=x+y+1
```

Атрибут `args` возвращает кортеж аргументов функции верхнего уровня.

```
>>> z.args
(1, x, y)
>>> z=x**2*y**2/4
>>> z.args
(1/4, x**2, y**2)
```

Метод `as_ordered_terms()` преобразует выражение в упорядоченный список операндов.

```
>>> z=x+y+1
>>> z.as_ordered_terms()
[x, y, 1]
```

После выделения кортежа или списка вы можете по индексу получить объект любого операнда.

После выполнения инструкции `from sympy import *` становится доступен специальный символ `oo` (две буквы 'o') – бесконечность, с которым тоже можно выполнять некоторые операции.

```
>>> oo+1
oo
>>> 1000000<oo
True
```

```
>>> 1/oo
0
```
Символ `oo` (бесконечность) в основном используется функцией `limit()`, а также функцией `integrate()` при задании пределов интегрирования.

5.2 Алгебраические вычисления

Алгебраические преобразования. С символьными выражениями можно выполнять алгебраические преобразования. Для этого следует использовать подходящие функции пакета `SymPy`. Например, функция `factor` раскладывает алгебраическое выражение на множители.

```
>>> x,y = symbols('x y')
>>> factor(x**2-y**2)
(x - y)*(x + y)
```
Можно ли x^4+4 разложить на множители без комплексных чисел. Давайте проверим.
```
>>> factor(x**4+4)
(x**2 - 2*x + 2)*(x**2 + 2*x + 2)
>>> expr=sum([ x**i for i in range(0, 6)]); expr
x**5 + x**4 + x**3 + x**2 + x + 1
>>> factor(expr)
(x + 1)*(x**2 - x + 1)*(x**2 + x + 1)
```
Функция `expand` обратная к функции `factor`. Она раскрывает скобки в алгебраическом выражении.
```
>>> f=(a+b)**3
>>> expand(f)
a**3 + 3*a**2*b + 3*a*b**2 + b**3
>>> expand((1-x)*(1+x+x**2+x**3+x**4+x**5))
-x**6 + 1
```
Формула $\log(xy) = \log x + \log y$ справедлива для вещественных положительных `x` и `y`, но для отрицательных и комплекснозначных `x` и `y` она неверна. Поэтому следующее разложение не срабатывает.
```
>>> expand(log(x*y))
log(x*y)
```
Можно проигнорировать подобное ограничение и заставить функцию выполнить привычное преобразование.
```
>>> expand(log(x*y), force=True)
log(x) + log(y)
```
Многие другие функции, выполняющие алгебраические преобразования, поддерживают опцию `force=True`. Однако лучшим решением является наложение подходящих ограничений на символьные переменные.
```
>>> var('a,b', positive=True)
(a, b)
>>> expand(log(a*b))
log(a) + log(b)
```

```
>>> expand(prod([(x-i) for i in range(1,5)]))
x**4 - 10*x**3 + 35*x**2 - 50*x + 24
```
Здесь использована функция `prod()` модуля sympy, которая вычисляет произведение своих аргументов.
```
>>> from sympy import prod, S
>>> prod([S(5), S(3)])
15
>>> type(_)
<class 'sympy.core.numbers.Integer'>
>>> prod(range(1,5))
24
>>> prod([4, 5], 2)
40
```
В выражении собрать коэффициенты при одинаковых степенях переменной можно с помощью функции `collect(expr,var)`.
```
>>> x,y,z=symbols('x y z')
>>> expr = x*y + x - 3 + 2*x**2 - z*x**2 + x**3
>>> collect(expr, x)
x**3 + x**2*(-z + 2) + x*(y + 1) - 3
>>> a,b,c,d=symbols('a b c d')
>>> collect(a*x+b*x+c*y+d*y,x)
c*y + d*y + x*(a + b)
>>> collect(a*x+b*x+c*y+d*y,(x,y))
x*(a + b) + y*(c + d)
```
Функция `cancel()` приводит рациональное выражение к общему знаменателю и сокращает числитель и знаменатель на общий множитель, если он есть.
```
>>> cancel(a/b+c/d)
(a*d + b*c)/(b*d)
>>> cancel((x**2 + 2*x + 1)/(x**3 + x**2))
(x + 1)/x**2
>>> cancel((x - 7)/(x +3)-(x-12)/(x-2))
50/(x**2 + x - 6)
```
Наоборот, функция `apart()` выполняет разложение дробей.
```
>>> apart((4*x**3 + 21*x**2 + 10*x + 12)/(x**4 + 5*x**3 + 5*x**2 + 4*x))
(2*x - 1)/(x**2 + x + 1) - 1/(x + 4) + 3/x
```
Универсальная функция `simplify()` пытается упрощать алгебраические выражения, используя различные комбинации приведенных выше функций.
```
>>> from sympy import *
>>> var('x')
>>> simplify(sin(x)**2 + cos(x)**2)
1
>>> a,b = symbols('a b')
>>> z=simplify((a**4 - 2*a**2*b**2 + b**4)/(a-b)**2);z
a**2 + 2*a*b + b**2
```
Как видите, функция `simplify` не до конца справилась со своей задачей. Поэтому ей требуется помощь.

```
>>> factor(z)
(a + b)**2
```
Имеются функции, предназначенные для преобразования тригонометрических выражений.
```
>>> expr=sin(2*x)+cos(3*x)
>>> expand_trig(expr)
2*sin(x)*cos(x) + 4*cos(x)**3 - 3*cos(x)
>>> tr=expand_trig(cos(4*x)-2*cos(x)*cos(3*x)); tr
-2*(4*cos(x)**3-3*cos(x))*cos(x)+8*cos(x)**4-8*cos(x)**2+1
>>> trigsimp(tr)
2*(-cos(2*x) + 1)**2 + 3*cos(2*x) - cos(4*x) - 3
>>> simplify(tr)
-cos(2*x)
>>> trigsimp(sin(x)**4 - 2*cos(x)**2*sin(x)**2 + cos(x)**4)
cos(4*x)/2 + 1/2
```
Символьные выражения являются объектами и имеют различные методы (аналогичные функциям), позволяющие их преобразовывать.
```
>>> a,b,c,d,x,y,z = symbols('a b c d x y z')
>>> expr=6+11*x-3*x**2-2*x**3
>>> z=expr.factor();z
-(x - 2)*(x + 3)*(2*x + 1)
>>> z.expand()
-2*x**3 - 3*x**2 + 11*x + 6
>>> ((a+b+c)**2).expand()
a**2 + 2*a*b + 2*a*c + b**2 + 2*b*c + c**2
>>> w=a/b+c/d
>>> u=w.together(); u
(a*d + b*c)/(b*d)
>>> u.simplify()
a/b + c/d
>>> z=a*x+b*x+c*y+d*y
>>> z.collect(x)
c*y + d*y + x*(a + b)
>>> z.collect((x,y))
x*(a + b) + y*(c + d)
```
Комплексная арифметика. Для ввода комплексных чисел можно использовать стандартную нотацию, используя для мнимой единицы символ 'I' (заглавная буква I).
```
>>> I*I
-1
>>> a,b =symbols('a b',real=True)
>>> z=symbols('z')
>>> z=a+b*I
```
Функции `re(z)` и `im(z)` вычисляют вещественную и мнимую часть комплексного числа `z`.

```
>>> re(z)
a
>>> im(z)
b
```
Функция `Abs(z)` и `arg(z)` вычисляют модуль и аргумент комплексного числа `z`.
```
>>> Abs(z)
sqrt(a**2 + b**2)
>>> arg(1+2*I)
atan(2)
```
Функция `conjugate(z)` выполняет комплексное сопряжение.
```
>>> conjugate(z)
a - I*b
```
Арифметические операции с комплексными числами выполняются обычным образом.
```
>>> z1=4+3*I
>>> z2=1-2*I
>>> z1+z2
5 + I
>>> simplify(z1*z2)
10 - 5*I
>>> simplify(z1/z2)
-2/5 + 11*I/5
>>> var('w')
>>> w=z**3;
>>> w=expand(w);
>>> collect(w,I)
a**3 - 3*a*b**2 + I*(3*a**2*b - b**3)
>>> simplify(re(w))
a*(a**2 - 3*b**2)
>>> simplify(im(w))
b*(3*a**2 - b**2)
```
Скажем несколько слов об отображении формул. Среда выполнения IPython и IPython console в Spyder поддерживают инструкцию `init_printing(...)`, которая позволяет настраивать способ отображения математических выражений в своем окне. Наиболее часто используются опции:
```
pretty_print= True / False
use_unicode = True / False
use_latex   = True / False
```
Например, инструкция
```
init_printing(use_latex=True)
```
включает Latex режим отображения. С другой стороны, инструкция
```
init_printing(pretty_print=False)
```
включает стандартный текстовый режим отображения.

В нашем пособии для представления результатов вычисления мы, обычно, используем текстовый режим. Но иногда, для отображения формул,

будем использовать Latex режим (в среде IDLE он недоступен). Вот, например, как выглядит решение квадратного уравнения в режиме Latex.

```
from sympy import *
init_printing(use_latex=True)
a,b,c,x  = symbols('a b c x')
solve(a*x**2+b*x+c,x)
```

$$\left[\left(-b+\sqrt{-4ac+b^2}\right)/2a,\quad -\left(b+\sqrt{-4ac+b^2}\right)/2a\right]$$

Вернемся к комплексной арифметике. SymPy умеет работать с комплексными функциями.

```
var('x',real=True)
exp(I*x).expand(complex=True)
I*sin(x) + cos(x)
```

В IPython, включив Latex режим, вы получите результат в привычной математической записи.

```
init_printing(use_latex=True)
exp(I*x).expand(complex=True)
```

$$i\sin(x)+\cos(x)$$

```
re(_)
```

$$\cos(x)$$

Решение уравнений. Большинство алгебраических преобразований мы можем выполнить самостоятельно. Однако решение уравнений является боле сложной задачей. И в этом вопросе помощь *SymPy* может быть значительной. Здесь мы рассматриваем функции пакета, с помощью которых можно решать алгебраические уравнения символьно. В тех случаях, когда функции пакета SymPy не могут получить решение, вам следует обращаться к функциям других пакетов, реализующих подходящие численные алгоритмы.

Для записи уравнений в SymPy не используются знаки равенства ‘=’ или двойного равенства ‘==’, а используется функция `Eq(expr1,expr2)`. Если можно проверить, равны ли выражения `expr1` и `expr2`, то функция `Eq()` возвращает `True`.

```
>>> from sympy import *
>>> Eq(3, 4)
False
```

У функции `Eq()` есть опция `evaluate=False`, которая принудительно может отменить проверку равенства.

```
>>> Eq(3, 4,evaluate=False)
Eq(3, 4)
>>> x,y=symbols('x y')
>>> Eq(x,y)
Eq(x, y)
```

Функции, которые предназначены для решения уравнений, используют функцию `Eq()` для записи этих уравнений. Кроме того, в SymPy все функции «решатели» в том месте, где может стоять функция `Eq()`, а стоит символьное выражение `expr`, автоматически полагают, что оно равно нулю (т.е.

используется Eq(expr,0)). Поэтому в «решателях» вместо уравнения вида Eq(expr1,expr2) можно использовать символьное выражение expr1-expr2.

Функцией, предназначенной для символьного решения алгебраического уравнения, является solveset(). Она имеет следующий синтаксис.

solveset(equation,variable=None[,domain=S.Complexes]),

где equation может быть записано в форме Eq() или быть выражением, которое полагается равным нулю. Второй аргумент задает имя искомой переменной, третий – определяет область, в которой ищутся корни (вещественная ось или комплексная плоскость). Результат функция возвращает в виде множества.

```
>>> solveset(Eq(x**2, 4), x)
{-2, 2}
>>> solveset(Eq(x**2 - 7*x+12, 0), x)
{3, 4}
>>> solveset(Eq(x**2 - 5*x+12, 0), x)
{5/2 - sqrt(23)*I/2, 5/2 + sqrt(23)*I/2}
>>> type(_)
<class 'sympy.sets.sets.FiniteSet'>
```

Для одного уравнения результатом будет объект типа FiniteSet.

```
>>> solveset(x**4-1, x)
{-1, 1, -I, I}
>>> type(_)
<class 'sympy.sets.sets.FiniteSet'>
```

В некоторых случаях результат может иметь другой тип.

```
>>> solveset(x**2 - x**2, x, domain=S.Reals)
(-oo, oo)
>>> type(_)
<class 'sympy.sets.fancysets.Reals'>
>>> solveset(cos(x) - 1, x, domain=S.Reals)
ImageSet(Lambda(_n, 2*_n*pi), Integers())
```

Если решения нет, то возвращается пустое множество.

```
>>> solveset(exp(x), x)
EmptySet()
```

Если решения найти не удалось, то функция возвращает ConditionSet.

В пакете есть функция real_roots(), которая возвращает только вещественные корни.

```
>>> x=S('x')
>>> real_roots(x**2-4)
[-2, 2]
>>> p = (x-3)**2*(x-2)*(x-1)*x*(x+1)*(x**2 + x + 1)
>>> real_roots(p)
[-1, 0, 1, 2, 3, 3]
```

В последнем примере два комплексных корня множителя (x**2 + x + 1) не были учтены функцией real_roots().

```
>>> real_roots(x**2+x+1)
[]
```
Функция `roots()` возвращает символьное выражение для корней полинома одной переменной. Результат возвращается в форме словаря с парами `корень:кратность`.
```
>>> a,b,c,x=symbols('a,b,c,x')
>>> roots(x**2-c**2,x)
{-c: 1, c: 1}
>>> roots(x**3 - 5*x**2 + 3*x + 9)
{3: 2, -1: 1}
```
Если последнее выражение разложить на множители, то получим
```
>>> factor(x**3 - 5*x**2 + 3*x + 9)
(x - 3)**2*(x + 1)
```
Имеется функция `nroots(p[,n=15,...])`, которая находит приближенное значение корней полинома `p` с точностью `n` значащих цифр.
```
>>> nroots(x**2-2, 20)
[-1.4142135623730950488, 1.4142135623730950488]
```
Для построения аналитических решений алгебраических уравнений и систем предназначена функция `solve(...)`. Первым аргументом она также принимает уравнение (или список уравнений) в форме `Eq()` или выражение (список выражений), которое полагается равным нулю.
```
>>> x,y = symbols('x y')
>>> solve(x**2+x-30,x)
[-6, 5]
```
Второй аргумент – имя искомой переменной необязателен, но его лучше всегда указывать. Рассмотрим пример.
```
>>> a,b,c,x = symbols('a b c x')
>>> eq = Eq(a*x**2+b*x+c, 0)
>>> solve(eq)
[{a: -(b*x + c)/x**2}]
```
Уравнение решено относительно переменной `a`. Поэтому лучше всегда указывать имя искомой переменной.
```
>>> solve(eq,x)
[(-b+sqrt(-4*a*c+b**2))/(2*a),-(b+sqrt(-4*a*c+b**2))/(2*a)]
```
В следующем примере решим систему линейных уравнений.

$$\begin{cases} x - 2y = 0 \\ 3x + y = 7 \end{cases}$$

```
>>> solve([x-2*y,3*x+y-7],x,y)
{y:1,x:2}
```
Форма возвращаемого результата зависит от поставленной задачи и может быть списком, словарем, множеством и т.д. Кроме того, форма результата может быть задана опциями. Например, опция `dict=False` отменяет вывод результата в форме словаря (и включает вывод в форме списка).
```
>>> solve([Eq(x**2-y**2,0),Eq(x+y,2)],x,y,dict=False)
[(1, 1)]
```

Функция `solve()` может решать некоторые трансцендентные уравнения.

```
>>> solve(exp(x**2+x-6) - 1, x)
[-3, 2]
>>> solve(exp(x) + 1, x)    # комплексный корень
[I*pi]
```

Функция `solve()` имеет много вариантов использования, с которыми вы можете познакомиться самостоятельно по справочной системе.

Для решения систем линейных уравнений предназначена функция `linsolve(...)`. Запишем последнюю систему уравнений в виде

$$\begin{cases} x - 2y = 0 \\ 3x + y - 7 = 0 \end{cases},$$

Функции `linsolve(...)` можно передавать левые части уравнений этой системы.

```
>>> linsolve([x-2*y,3*x+y-7],x,y)
{(2, 1)}
```

Другой способ решения этой задачи с помощью функции `linsolve()` состоит в использовании расширенной матрицы системы

$$\begin{pmatrix} 1 & -2 & 0 \\ 3 & 1 & 7 \end{pmatrix}.$$

```
>>> linsolve(Matrix(([1,-2,0],[3,1,7])),(x,y))
{(2, 1)}
```

Матричная алгебра. Символьные матрицы создаются функцией `Matrix()` и являются объектами, у которых имеются атрибуты и методы.

```
>>> from sympy import *
>>> a,b,c,d = symbols('a b c d')
>>> M=Matrix([[a,b],[c,d]])
```

При работе с символьными матрицами следует убедиться, что они (матрицы) отображаются в вашей среде корректно. Например, в IPython, если просто напечатать имя матрицы в режиме `init_printing(use_latex=True)`, происходит сбой и матрица не отображается. Для печати матрицы ее имя следует передать аргументом функции «милой» печати `pprint(имя_матрицы)`. Можно также отключить `Latex` режим следующей командой `init_printing(use_latex=False)`. Тогда, чтобы напечатать матрицу, в командной строке можно просто ввести ее имя и выполнить инструкцию (без использования функций печати).

Доступ к элементам символьных матриц выполняется по индексам с использованием квадратных скобок.

```
>>> M[0,0]
a
```

Использование одномерного списка в аргументе функции `Matrix()` приводит к созданию вектора–столбца (матрицы с одним столбцом).

```
>>> v1=Matrix([1,2,3]); v1
```

```
Matrix([
[1],
[2],
[3]])
```
После создания матрицы вы можете вызывать методы, которые будут возвращать затребованные преобразования.
```
>>> M.det()                    # вычисление определителя
a*d - b*c
>>> M.T                        # транспонирование матрицы
Matrix([
[a, c],
[b, d]])
>>> B=simplify(M.inv()); B     # вычисление обратной матрицы
Matrix([
[ d/(a*d - b*c), -b/(a*d - b*c)],
[-c/(a*d - b*c),  a/(a*d - b*c)]])
>>> simplify(M*B)              # проверка
Matrix([
[1, 0],
[0, 1]])
>>> M=Matrix([[1,4],[2,3]])
>>> M.eigenvals()              # вычисление собственных чисел
{5: 1, -1: 1}       # с.число 5 имеет кратность 1; с.число -1 имеет кратность 1
>>> M.eigenvects()
[(-1, 1, [Matrix([[-2],[ 1]])]),
 ( 5, 1, [Matrix([[1],[1]])])]
```
Здесь видно, что собственному числу `-1` с кратностью 1 соответствует собственный вектор `(-2,1)`; собственному числу `5` с кратностью 1 соответствует собственный вектор `(1,1)`.

Операции между символьными матрицами выполняются с помощью привычных операторов.
```
>>> x=symbols('x:2:2'); x
(x00, x01, x10, x11)
>>> y=symbols('y:2:2'); y
(y00, y01, y10, y11)
>>> A=Matrix([[x[0],x[1]],[x[2],x[3]]]); A
Matrix([
[x00, x01],
[x10, x11]])
>>> B=Matrix([[y[0],y[1]],[y[2],y[3]]])
>>> A+B
Matrix([
[x00 + y00, x01 + y01],
[x10 + y10, x11 + y11]])
>>> A*B
```

```
Matrix([
[x00*y00 + x01*y10, x00*y01 + x01*y11],
[x10*y00 + x11*y10, x10*y01 + x11*y11]])
>>> A**2
Matrix([
[ x00**2 + x01*x10, x00*x01 + x01*x11],
[x00*x10 + x10*x11, x01*x10 + x11**2]])
>>> A**(-1)          # вычисление обратной матрицы
Matrix([
[1/x00+x01*x10/(x00**2*(x11-x01*x10/x00)),
                                -x01/(x00*(x11- x01*x10/x00))],
[  -x10/(x00*(x11 - x01*x10/x00)),  1/(x11 - x01*x10/x00)]])
>>> A=Matrix([[1,2],[3,4]])
>>> B=Matrix([[1,-1],[-1,3]])
>>> A/B              # умножение на обратную матрицу (A*B^-1)
Matrix([
[ 5/2, 3/2],
[13/2, 7/2]])
>>> A**(-1)          # вычисление обратной матрицы
Matrix([
[ -2,    1],
[3/2, -1/2]])
```

Многие операции можно выполнить, используя методы объектов «матрица».

```
>>> v1=Matrix([1,2,3]);
>>> v2=Matrix([1,1,1])
>>> v1.dot(v2)       # скалярное произведение
6
>>> v1.cross(v2)     # векторное произведение
Matrix([
[-1],
[ 2],
[-1]])
```

В модуле SymPy имеются конструкторы специальных матриц: нулевых (zeros), единичных (ones), диагональных (diag) и т. д, а также большое количество других функций, выполняющих матричные вычисления.

5.3 Реализация основных понятий математического анализа

Кроме символьных алгебраических преобразований функции пакета `SymPy` умеют выполнять многие операции математического анализа.

Вычисление пределов. Для аналитического вычисления пределов используется функция `limit(...)`.

```
>>> from sympy import *
>>> x,y,z=symbols('x y z')
>>> limit(sin(x)/x, x, 0)
1
```

```
>>> limit(x*log(x),x,0)
0
```
Для записи символа ∞ (бесконечность) в пакете SymPy используется запись oo (две буквы 'o').
```
>>> expr = x**2*exp(-x)
>>> limit(expr, x, oo)
0
>>> limit((1+z/x)**x,x,oo)
exp(z)
>>> limit((sin(x+z)-sin(x))/z,z,0)        # производная sin(x)
cos(x)
```
У limit()есть невычисляемый эквивалент – оператор Limit(), который возвращает символьный объект типа 'sympy.series.limits.Limit' (невычисленный предел).
```
>>> expr = Limit((1-cos(x))/x**2, x, 0)
>>> expr
Limit((-cos(x) + 1)/x**2, x, 0)
```
Для вычисления символьного объекта, созданного невычисляемым оператором, нужно использовать метод doit().
```
>>> expr.doit()
1/2
```
Для вычисления одностороннего предела следует передать в функцию limit() еще один аргумент '+' или '-'.
```
>>> limit(1/x, x, 0, '+')
oo
>>> limit(1/x, x, 0, '-')
-oo
>>> limit(tan(x),x,pi/2,'-')
oo
```
Дифференцирование. Для символьного дифференцирования предназначена функция diff(...). Первым аргументом она принимает символьное выражение, которое будет дифференцироваться, а вторым – переменную, по которой вычисляется производная.
```
>>> from sympy import *
>>> diff(sin(x)*exp(x), x)
exp(x)*sin(x) + exp(x)*cos(x)
```
Часто полезно использовать функцию «милой» печати pprint(), которая, в зависимости от используемой среды выполнения, пытается отобразить результат в привычном математическом виде.
```
>>> pprint(diff(sin(x)*exp(x), x))  # в среде IDLE
 x          x
e *sin(x)+e *cos(x)
```
В IPython console Spyder и Jupyter Notebook функция pprint() результат выводит еще «милее».

Функцию diff() можно использовать для вычисления производных второго и более высокого порядков.

```
>>> diff(x**4, x, x)     # два раза дифференцируем по x
12*x**2
>>> diff(x**4, x, 3)     # третья производная по x
24*x
```

Можно дифференцировать выражение по нескольким переменным, т.е. вычислять смешанные частные производные.

```
>>> f = exp(x*y**2)          #  f(x,y)=e^{xy^2}
>>> diff(f, x, y)            #  ∂^2 f / ∂x ∂y
2*y*(x*y**2 + 1)*exp(x*y**2)
>>> diff(f, x, 3,y, 2)       #  ∂^5 f / ∂x^3 ∂y^2
2*y**4*(2*x**2*y**4 + 13*x*y**2 + 15)*exp(x*y**2)
```

В следующем примере мы вычисляем градиент функции 3–х переменных.

```
>>> x,y,z=symbols('x y z')
>>> f=x**2+y**2+z**2
>>> g=[diff(f,var) for var in [x,y,z]]; g
[2*x, 2*y, 2*z]
```

У символьных выражений есть также метод diff().

```
>>> f.diff(x,2,y)            #  ∂^3 f / ∂x^2 ∂y
2*y**3*(x*y**2 + 2)*exp(x*y**2)
```

Оба способа вычисления производных эквивалентны.

У diff()есть невычисляемый эквивалент – оператор Derivative(). Он возвращает выражение типа 'sympy.core.function.Derivative'.

```
>>> f1=Derivative(f,x,y,y); f1    #  ∂^3 f / ∂x ∂y^2
Derivative(exp(x*y**2), x, y, y)
```

Чтобы вычислить выражение, созданное невычисляемым оператором, нужно использовать метод doit().

```
>>> f1.doit()
2*(2*x**2*y**4 + 5*x*y**2 + 1)*exp(x*y**2)
```

Иногда полезно иметь символьные формулы, содержащие произвольные функции. Чтобы сказать системе, что переменная является именем символьной функции можно использовать инструкцию

```
f = Function('f')
```

После этого идентификатор f становится именем функции и с ним можно выполнять операции, определенные для функций. Следующие несколько примеров мы выполним в IPython console в интегрированной среде Spyder, и приведем результат в таком виде, в котором он возвращается в Latex режиме.

Чтобы включить этот режим в IPython следует выполнить инструкцию `init_printing(use_latex=True)`. В IDLE Python результат будет возвращаться в текстовом виде, поскольку Latex режим не поддерживается.

```
from sympy import *
init_printing(use_latex=True)
f=Function('f')
f(x).diff(x)
```

$$\frac{d}{dx}f(x)$$

Другой способ создать символьные функции состоит в использовании функции `symbols()` с опцией `cls=Functions`.

```
f, g, h = symbols('f g h', cls=Function)
```

Получим некоторые классические формулы дифференцирования.

```
(f(x)*g(x)*h(x)).diff(x)
```

$$f(x)g(x)\frac{d}{dx}h(x) + f(x)h(x)\frac{d}{dx}g(x) + g(x)h(x)\frac{d}{dx}f(x)$$

```
diff(f(x)/g(x),x)
```

$$-\frac{\frac{d}{dx}g(x)}{g^2(x)}f(x) + \frac{\frac{d}{dx}f(x)}{g(x)}$$

Вот как можно продифференцировать сложную функцию.

```
diff(exp(f(x)),x)
```

$$e^{f(x)}\frac{d}{dx}f(x)$$

Символьная функция может иметь несколько аргументов.

```
f(x,y).diff(x,2)
```

$$\frac{\partial^2}{\partial x^2}f(x, y)$$

```
f(x,y).diff(x,y)
```

$$\frac{\partial^2}{\partial x \partial y}f(x, y)$$

Пример. Найдем максимум функции $f(x) = x^3 - 2x^2 + x$.

Вначале определим экстремальные точки функции, приравняв нулю ее производные. Затем вычислим значения вторых производных в экстремальных точках.

```
x=symbols('x')
f=x**3-2*x**2+x
f1=diff(f,x);f1
sols=solve(f1,x); sols
[1/3, 1]
diff(f1,x).subs({x:sols[0]})
-2
diff(f1,x).subs({x:sols[1]})
2
```

В точке 1/3 вторая производная отрицательна, а это означает, что это точка максимума. Т.о. локальный максимум равен

```
>>> f.subs({x:sols[0]})
4/27
```

Интегрирование. Для символьного интегрирования предназначена функция `integrate(...)`. С ее помощью можно вычислять как определенные, так и неопределенные интегралы. Первым аргументом она принимает символьное выражение, которое будет интегрироваться, вторым – переменную интегрирования или кортеж, состоящий из имени переменной и ее нижнего и верхнего предела. Если второй аргумент – имя, то вычисляется неопределенный интеграл, т.е. первообразная подынтегральной функции.

```
>>> from sympy import *
>>> x,y,a,b = symbols('x y a b')
>>> integrate(1/x, x)
log(x)
>>> integrate(cos(x)**2, x)
x/2 + sin(x)*cos(x)/2
>>> integrate(log(x), x)
x*log(x) - x
```

Проверим результат дифференцированием.

```
>>> _.diff(x)
log(x)
```

Обратите внимание, что SymPy не включает в результат постоянную интегрирования. Вы можете добавить константу самостоятельно, или можете сформулировать задачу как решение соответствующего дифференциального уравнения (ОДУ). Во втором случае произвольная постоянная будет включаться в результат.

Подынтегральное выражение может содержать несколько символьных переменных, по одной из которых будет выполняться интегрирование.

```
>>> x,y,a,b = symbols('x y a b')
>>> y=integrate((a+x)**3, x)
>>> pprint(y)
          2  2           4
 3     3*a *x       3   x
a *x + ------- + a*x  + --
          2             4
```

Чтобы увидеть выражение в более привычном виде, мы использовали функцию `pprint(...)`. В зависимости от используемой вами оболочки, результат может выглядеть более привычно, например, так $a^3x + 3a^2x^2/2 + ax^3 + x^4/4$.

Если первообразную не удалось найти, то возвращается невычисляемый объект `Integral`.

```
>>> integrate(x**x, x)
Integral(x**x, x)
```

Для вычисления определенного интеграла в функцию `integrate()` вторым аргументом нужно передать кортеж вида `(var,lower_limit,upper_limit)`, содержащий имя переменной интегрирования `var`, а также нижний `lower_limit` и верхний `upper_limit` пределы интегрирования.

```
>>> integrate(sin(x), (x, 0, pi))          # ∫_0^π sin x d x
```
2

Аналогично вычисляются несобственные интегралы с бесконечными пределами интегрирования.

```
>>> integrate(exp(-x), (x, 0, oo))         # ∫_0^∞ e^(-x) d x
```
1

Здесь вместо символа ∞ нужно вводить oo (две буквы 'o').

```
>>> integrate(exp(-x**2), (x, -oo, oo))    # ∫_-∞^∞ e^(-x^2) d x
```
sqrt(pi)

Функцию integrate() можно использовать для вычисления повторных интегралов.

```
>>> integrate(x*y,x,y)                     # ∫(∫(x·y)d x)d y
```
x**2*y**2/4

```
>>> integrate(x*y,(x,0,1),(y,0,1))         # ∫_0^1 (∫_0^1 (x y)d x) d y
```
1/4

```
>>> integrate(exp(-x**2-y**2),(x,-oo,oo),(y,-oo,oo)) # ∫_-∞^∞ ∫_-∞^∞ e^(-x^2-y^2) d x d y
```
pi

Внутренние пределы интегрирования могут зависеть от переменной интегрирования внешнего интеграла. Вот пример вычисления повторного интеграла по верхней половине единичного круга.

```
>>> integrate(x**2*y,(y,0,sqrt(1-x**2)),(x,-1,1))   # ∫_-1^1 d x ∫_0^√(1-x^2) x^2 y d y
```
2/15

У integrate()есть невычисляемый эквивалент – функция Integral(). Она возвращает объект типа 'sympy.integrals.integrals.Integral'. Чтобы потом вычислить интеграл, нужно использовать метод doit().

```
>>> expr = Integral(log(x)**2, x); expr
Integral(log(x)**2, x)
>>> expr.doit()
x*log(x)**2 - 2*x*log(x) + 2*x
```

Суммы и ряды. Для символьного вычисления конечных и бесконечных сумм используется функция summation. В формате summation(f,(i,imin,imax)) она вычисляет сумму вида $\sum_{i=i_{\min}}^{i_{\max}} f(i)$ (приращение индекса i равно 1).

```
>>> var('i,j,n')
```

```
>>> summation(2*i-1,(i,1,12))
144
>>> summation((i+j)**2,(i,1,3),(j,1,4))
266
```
Пределы суммирования могут быть символьными.
```
>>> var('a,b')
>>> s = summation(6*n**2 + 2**n, (n, a, b)); s
-2**a+2**(b + 1)-2*a**3+3*a**2-a+2*b**3+3*b**2+b
```
Можно суммировать выражения, содержащие более чем одну переменную; другие переменные будут трактоваться, как константы.
```
>>> summation(x**i/factorial(i),(i,1,7))
x**7/5040 + x**6/720 + x**5/120 + x**4/24 + x**3/6 + x**2/2 + x
```
Результат вычисления суммы может быть преобразован в более простое выражение.
```
>>> summation(i**2, (i, 1, n))
n**3/3 + n**2/2 + n/6
>>> summation(i**4,(i,1,n))
n**5/5 + n**4/2 + n**3/3 - n/30
>>> factor(_)
n*(n + 1)*(2*n + 1)*(3*n**2 + 3*n - 1)/30
>>> simplify(summation((-1)**i*i/(4*i**2-1),(i,1,n)))
((-1)**n - 2*n - 1)/(4*(2*n + 1))
```
Пределы суммирования могут быть бесконечными. Такие суммы принято называть рядами и *SymPy* умеет их вычислять.
```
>>> summation(1/i**2,(i,1,oo))
pi**2/6
>>> summation(x**n/factorial(n),[n,0,oo])
exp(x)
```
Если найти результат суммирования в символьной форме не удается, *SymPy* возвращает невычисляемую форму оператора суммирования `Sum()`.
```
>>> summation(i/(1+factorial(i)),(i,1,oo))
Sum(i/(factorial(i) + 1), (i, 1, oo))
```
Невычисляемую форму можно использовать сразу, чтобы впоследствии использовать метод `doit()`.
```
>>> sn = Sum(1/i**2, (i, 1, n))
>>> sn.doit()
harmonic(n, 2)
>>> limit(sn.doit(), n, oo)
pi**2/6
```
Но вычисление предела эквивалентно вычислению бесконечной суммы.
```
>>> summation(1/i**2, (i, 1, oo))
pi**2/6
```
SymPy умеет разлагать функции в степенные ряды. Для построения разложения функции (выражения) `f(x)` в ряд в окрестности точки $x=x_0$ до слагаемых порядка $(x-x_0)^n$ предназначена функция
```
series(expr[, x=None, x0=0, n=6,...])
```

и метод f(x).series(x[,x0=0,n=6]). Если x0 и n опущены, то им по умолчанию присваиваются значения x0=0 и n=6.

```
>>> g=series(exp(x)); g
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
>>> sin(x).series(x,0,7)
x - x**3/6 + x**5/120 + O(x**7)
>>> log(x).series(x,1,4)
-1-(x - 1)**2/2+(x - 1)**3/3+x+O((x-1)**4,(x,1))
```

В IPython console Spyder последняя инструкция возвращает результат в виде

$$-1-(x-1)^2/2+(x-1)^3/3+x+\mathcal{O}((x-1)^4;x\to 1)$$

Чтобы не выводить остаточный член O(...), можно использовать метод expr.series(...).removeO().

```
>>> sin(x).series(x,0,7).removeO()
x**5/120 - x**3/6 + x
```

Функцию series можно использовать для переразложения полинома по степеням x−a при любом a.

```
>>> P=x**5+2*x**4-3*x**3+6*x**2-12*x+17
>>> P.series(x,0,5)
17 - 12*x + 6*x**2 - 3*x**3 + 2*x**4 + O(x**5)
>>> P.series(x,1,5).removeO()
4*x + 7*(x - 1)**4 + 15*(x - 1)**3 + 19*(x - 1)**2 + 7
```

С рядами можно выполнять некоторые арифметические операции.

```
>>> f=series(sin(x)); f
x - x**3/6 + x**5/120 + O(x**6)
>>> g=series(exp(x)); g
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
>>> f+g
1 + 2*x + x**2/2 + x**4/24 + x**5/60 + O(x**6)
>>> simplify(f*g)
x + x**2 + x**3/3 - x**5/30 + O(x**6)
>>> simplify(g**2)
(120+120*x+60*x**2+20*x**3+5*x**4+x**5+O(x**6))**2/14400
```

Ряд можно дифференцировать.

```
>>> f.diff(x)
1 - x**2/2 + x**4/24 + O(x**5)
```

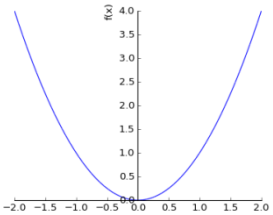
5.4 Графические возможности пакета SymPy

При работе с символьными функциями мы привыкли представлять их наглядно в виде графиков. В пакете SymPy имеется модуль sympy.plotting, который содержит функции, с помощью которых можно строить двумерные и трехмерные графики символьных выражений.

График функции, заданной явным уравнением y=f(x), строится с помощью функции sympy.plotting.plot(f(x)[,...]). Первый аргумент –

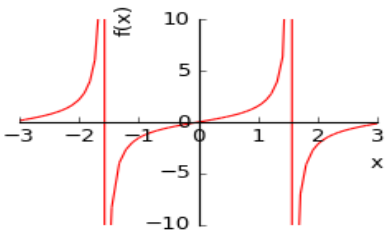
выражение `f(x)`, график которого надо построить. Второй аргумент, если он задан, определяет интервал изменения переменной `x`.

```
from sympy import symbols
from sympy.plotting import plot
x = symbols('x')
p1=plot(x**2,(x,-1,1))
```



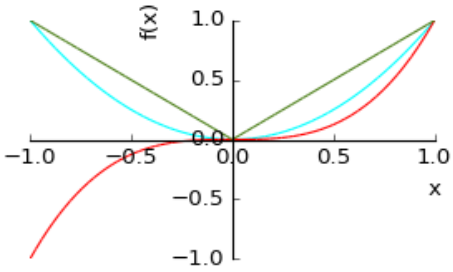
Система автоматически выбирает масштаб по вертикальной оси и выбирает достаточное количество точек, чтобы кривая была гладкой. Параметры графика можно изменять с помощью опций: `title='строка'; xlabel='строка'; ylabel='строка'; legend=True` или `False; xscale='linear'` или `'log'; yscale='linear'` или `'log'; axis= True` или `False; axis_center=(x0,y0)` или `'center', 'auto'; xlim=(`x_{min}`,`x_{max}`); ylim=(`y_{min}`,`y_{max}`); aspect_ratio=(`r_x`,`r_y`)` или `'auto'; autoscale=True` или `False; margin=float([0,1]); line_color=`цвет. Назначение большинства опций понятно из их названия.

```
p1=plot(tan(x),(x,-3,3),ylim=(-10,10),
line_color='r')
```



Кроме того, функция `plot()` возвращает объект, который имеет свои атрибуты и методы, и которые можно использовать для управления свойствами графика.

```
p1 = plot(x**2,(x,-1,1),show=False)
p1.line_color = 'cyan'
p2 = plot(x**3,(x,-1,1),show=False,
                                line_color='r')
p3 = plot(Abs(x),(x,-1,1),show=False,
                    line_color=(0.3,0.5,0.1))
p1.extend(p2)
p1.extend(p3)
p1.show()
```

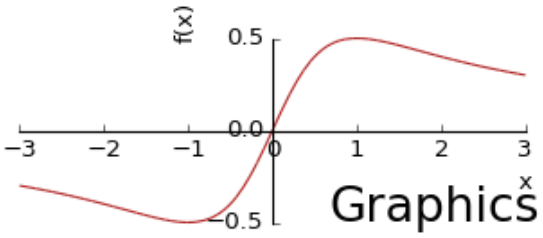


В этом примере с помощью атрибута `line_color` мы задали цвет первого графика, который из-за опции `show=False` был создан невидимым. Объекты графиков `p2` и `p3` вначале также созданы только в памяти. Затем с помощью метода `p1.extend()` они добавлены на первый график. Инструкция `p1.show()` рисует графики всех кривых.

Графические функции SymPy используют библиотеку matplotlib, поэтому управлять многими свойствами графиков можно, используя функции этой библиотеки. Например, фон окна или его заголовок можно задать, используя методы объекта Figure.

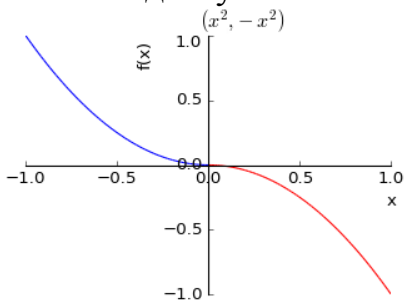
```
from sympy import *
from sympy.plotting import plot
```

```
x = symbols('x')
y = x/(1 + x**2)
plot(y, (x, -3, 3), ylim=(-1,1),
                    line_color='firebrick')
fig=plt.gcf()
fig.set_facecolor('white')
fig.text(0.6,0.3,'Graphics',fontsize=24)
```



В одной функции `plot()` можно строить несколько графиков. Для каждой кривой нужно задавать цвет отдельно, обращаясь к ним по индексу.

```
x = symbols('x')
f1 = x**2
f2 = -x**2
str=latex(S('x**2 , -x**2 ',evaluate=False))
p = plot((f1, (x, -1, 0)), (f2, (x, 0, 1)),
             title='$'+str+'$', show=False)
p[0].line_color = 'blue'
p[1].line_color = 'red'
p.show()
```



Если диапазон изменения переменной одинаковый для всех функций, то график нескольких кривых можно построить проще.

```
plot(x, x**2, x**3, (x, -1, 1))
```

Можно принудительно задать точки, по которым будет строиться график. Для этого опцией `adaptive=False` надо отключить режим автоматического выбора точек на отрезке, и опцией `nb_of_points=количество` задать желаемое количество точек.

```
x = symbols('x')
plot(sin(x),(x,-3.5,3.5), adaptive=False,
                          nb_of_points=10)
fig=plt.gcf()
fig.set_facecolor('white')
```

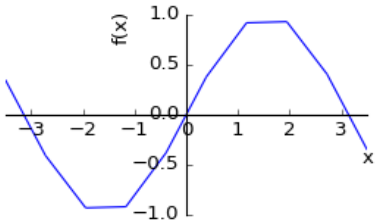
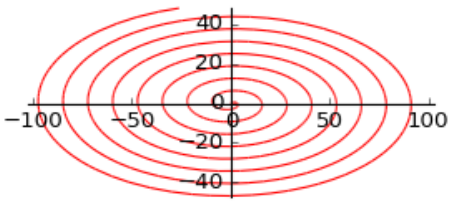


График кривых, задаваемых параметрическими уравнениями `x=x(t), y=y(t)`, строится с помощью функции

`plot_parametric(x(t),y(t),(t,tmin,tmax)[,...]).`

Первые два аргумента представляют выражения `x(t),y(t)`, которые определяют координаты *x* и *y* как функцию параметра *t*. Третий аргумент определяет диапазон изменения параметра `t`.

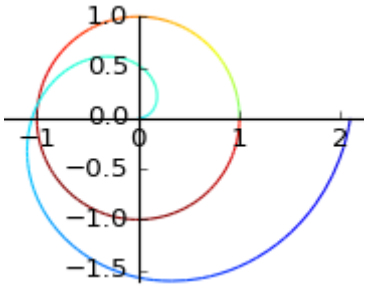
```
from sympy.plotting import plot_parametric
t = symbols('t')
plot_parametric(2*t*sin(t), t*cos(t),
                    (t, 0, 50), line_color='r')
fig=plt.gcf()
ax=fig.gca()
ax.axis('equal')
```



Можно строить несколько кривых с общим диапазоном изменения параметра. Атрибут `line_color`, задающий цвет, должен быть функцией, возвращающей

числовое значение. Если функция принимает один аргумент, то он рассматривается как параметр кривой. Если функция цвета принимает два параметра, то они рассматриваются как пространственные координаты точек.

```
p=plot_parametric((cos(t), sin(t)),
                  (t*cos(t)/3, t*sin(t)/3),
                  (t,0,2*pi),ylim=(-2,1.5),
                  margin=0.2,show=False)
p[0].line_color=lambda t: sin(t)
p[1].line_color=lambda x,y: x
p.show()
```



Для каждой кривой на графике можно задать свой диапазон изменения параметра.

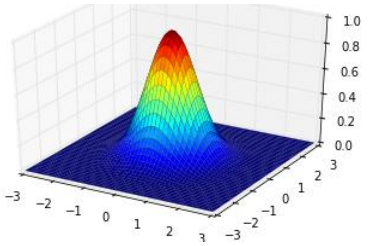
```
pp= plot_parametric((cos(t), sin(t),(t,0,2*pi)),\
          (t*cos(t)/10, t*sin(t)/10,(t,0,50)),\
          ylim=(-5,5),margin=0.2)
```

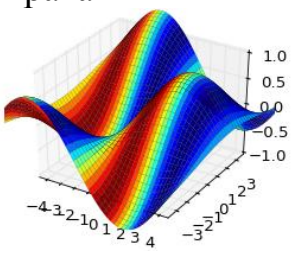
Для построения графиков функций 2 – x переменных предназначена функция `plot3d(expr,(x,`x_{min}`,x,`x_{max}`),(y,`y_{min}`,`y_{max}`)[,...])`.

```
from sympy import *
from sympy.plotting import plot3d
var('x y')
plot3d((exp(-(x**2+y**2))),(x,-3,3),(y,-3,3))  # следующий график слева
```

Атрибуту `surface_color` можно присвоить функцию, которая должна возвращать вещественное число, и которая будет представлять цвет точек поверхности.

```
g = plotting.plot3d(sin(x/2)*cos(y/2), (x, -6, 6), (y, -6, 6), show=False)
g[0].surface_color = lambda x, y: sin(x)
g.show()                    # следующий график справа
```



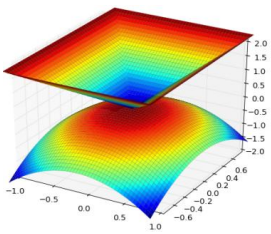


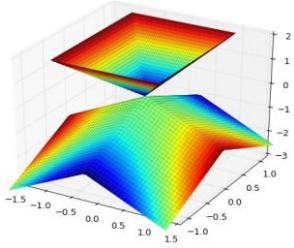
Чтобы атрибут `surface_color` подействовал, сначала должен быть сознан невидимый объект поверхности, потом для него задается значение атрибута цвета и, только потом, поверхность должна быть отображена.

На одном графике можно строить несколько поверхностей, имеющих общие диапазоны по `x` и `y`.

```
plotting.plot3d(-x**2-y**2,Abs(x-y)+Abs(x+y),(x,-1,1),(y,-1,1))
```

Этот график показан на следующем рисунке слева



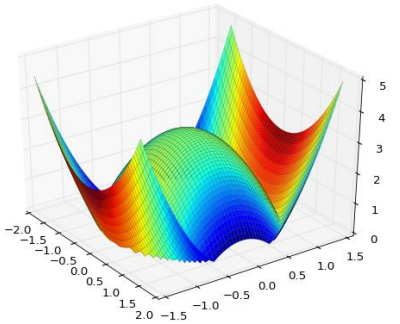


Можно задавать диапазоны изменения независимых переменных для каждой функции.

```
p=plot3d((-Abs(x)-Abs(y),(x,-1.5,1.5),(y,-1.5,1.5)),
         (Abs(x-y)+Abs(x+y),(x,-1,1),(y,-1,1)), show=False)
p[0].surface_color = lambda x, y: x**2-y**2
p[1].surface_color = lambda x, y: Abs(x-y)+Abs(x+y)
p.show()   # предыдущий график справа
```

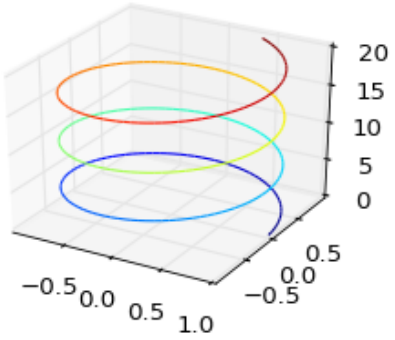
У функции `plot3d()` есть опции `nb_of_points_x`, `nb_of_points_y`, которые задают количество точек разбиения области изменения независимых переменных `x` и `y`. Имеется также опция `surface_color`, которая должна быть функцией двух переменных.

```
f=lambda x,y:Abs(3-x**2-2*y**2)
p = plotting.plot3d(f(x,y), (x, -1.6, 1.6), (y, -1.6, 1.6), \
         nb_of_points_x=60,nb_of_points_y=60,\
         surface_color = lambda x, y: y**2-x**2)
p.show()
```



Функция `plot3d_parametric_line()` строит график пространственной кривой, заданной параметрически. В следующем примере строится график одной кривой.

```
from sympy.plotting import plot3d_parametric_line
u = symbols('u')
p=plot3d_parametric_line(cos(u),sin(u),u,(u,0,20),
                         line_color=lambda x, y, z: z / 10)
p=plot3d_parametric_line(cos(u),sin(u),u,(u,0,20),show=False)
p[0].line_color = lambda x, y, z: z / 10
p.show()
```

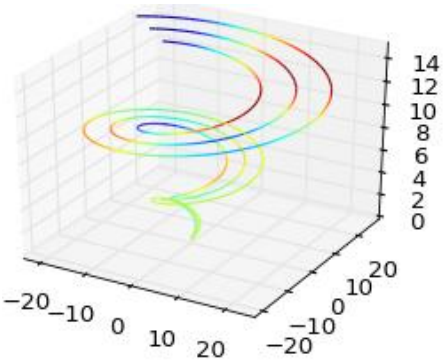


Здесь мы использовали опцию `line_color`, которая должна быть функцией одной, двух или трех переменных.

Можно строить графики нескольких кривых.

```
p = plot3d_parametric_line((t*cos(t), t*sin(t), t, (t, 0, 15)),
```

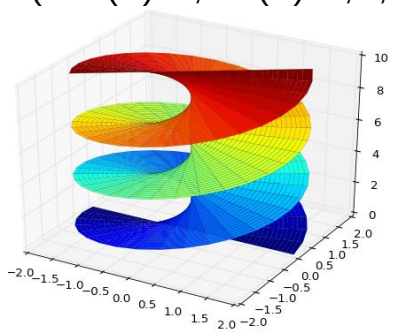
```
(1.5*t*cos(t), 1.5*t*sin(t), t, (t, 0, 15)),
(2*t*cos(t), 2*t*sin(t), t, (t, 0, 15)), show=False)
p[0].line_color = lambda t: t              # t – параметр кривой
p[1].line_color = lambda x, y: x*y         # x,y пространственные координаты
p[2].line_color = lambda x, y, z: x*y*z # все пространственные координаты
p.show()
```



У функции `plot3d_parametric_line()` имеется опция `nb_of_points`, задающая количество точек на кривой.

Функция `plot3d_parametric_surface()` строит график поверхности, заданной параметрическими уравнениями.

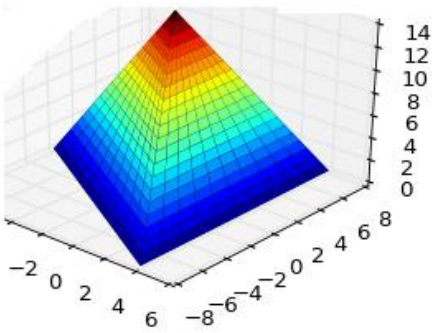
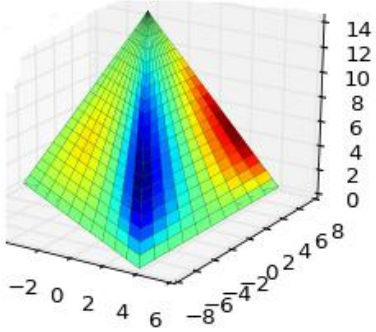
```
from sympy.plotting import plot3d_parametric_surface
u, v = symbols('u v')
plot3d_parametric_surface(cos(u)*v,sin(u)*v,u,(u,0,10),(v,-2,2))
```



У функции `plot3d_parametric_surface()` имеются опции `nb_of_points_u`, `nb_of_points_v`, назначение которых понятно из их названия. Использование этих опций для поверхностей с ребрами позволяет эти ребра правильно отображать. В следующем примере мы строим треугольную пирамиду по ее параметрическим уравнениям и корректное задание значений `nb_of_points_u`, `nb_of_points_v` позволяет улучшить изображение поверхности. Имеется также опция (и атрибут) `surface_color` , которая должна принимать ссылку на функцию.

```
u, v = symbols('u v')
X=(2*Abs(v)-3*Abs(v-1)+Abs(v-3))*(-4+3*Abs(u)-3*Abs(u-1)-
3*Abs(u-2)+3*Abs(u-3))/2
Y=-(2*Abs(v)-3*Abs(v-1)+Abs(v-3) )*\
(3*Abs(u-1)-3*Abs(u-2)+Abs(u-3)-Abs(u));
Z=4*(2+Abs(v-1)-Abs(v-3));
plot3d_parametric_surface(X,Y,Z,(u,0,3),(v,0,3),
    nb_of_points_u=31,nb_of_points_v=31) # следующ. рисунок слева
```

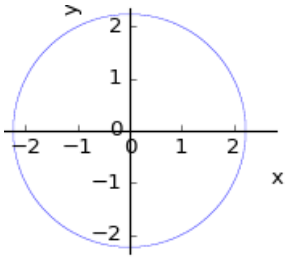
```
plot3d_parametric_surface(X,Y,Z,(u,0,3),(v,0,3),\
    nb_of_points_u=31,nb_of_points_v=31,\
    surface_color = lambda x, y, z: x*y*z)    # следующ. рисунок справа
```

Функция `plot_implicit()` строит кривые по их неявным уравнениям, или рисует залитые области, заданные неравенствами. Использование функции поясним на примерах.

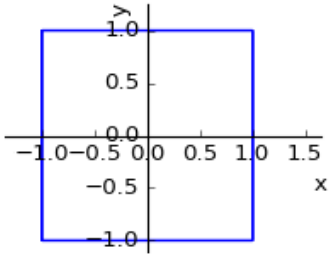
В следующем примере мы рисуем кривую по ее неявному уравнению $x^2 + y^2 = 5$. Напомним, что уравнения в SymPy записываются с использованием функции `Eq()`. Например, последнее уравнение записывается в форме `Eq(x**2+y**2, 5)`.

```
x, y = symbols('x y')
plot_implicit(Eq(x**2 + y**2, 5))
fig=plt.gcf()
ax=fig.gca()
ax.axis('equal')
ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
```



Можно задать диапазон изменения независимых переменных. В следующем примере мы строим квадрат по его неявному уравнению. Иногда отключение адаптивной сетки опцией `adaptive=False` улучшает изображение.

```
W=2-x**2-y**2-sqrt((1-x**2)**2+(1-y**2)**2)
plot_implicit(W,(x,-1.2,1.2),(y,-1.2,1.2),
                              adaptive=False)
fig=plt.gcf()
ax=fig.gca()
ax.axis('equal')
```

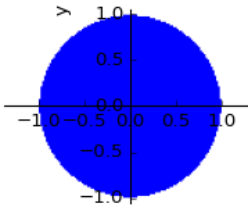


Опцией `points=количество` можно задать количество точек в области изменения переменных `x` и `y`. Опция `depth` задает глубину рекурсии при поиске точек неявно заданной кривой. Эту опцию имеет смысл использовать в паре с опцией `adaptive=False`.

Для построения областей в качестве первого аргумента функции `plot_implicit()` следует использовать неравенства.

```
from sympy.plotting import plot_implicit
```

```
plot_implicit(x**2+y**2<1)
fig=plt.gcf()
fig.set_facecolor('white')
ax=fig.gca()
ax.axis('equal')
```

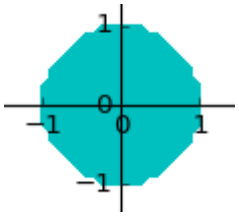


В этом примере заметно влияние опции depth. В предыдущем примере добавьте опцию `depth=2` в функцию `plot_implicit()` и граница области (круга) станет более гладкой.

```
plot_implicit(x**2+y**2<1, depth=2)
```

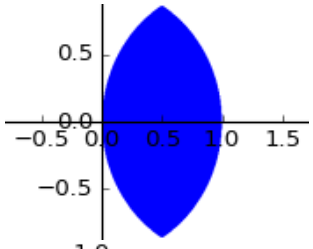
Опциями `adaptive=True/False` и `points=количество` можно управлять густотой сетки, используемой в алгоритме построения кривой или области. А опция `line_color` используется для задания цвета области.

```
plot_implicit(x**2+y**2<1,adaptive=False,
                  points=100, line_color='c')
fig=plt.gcf()
ax=fig.gca()
ax.axis('equal')
ax.set_xlim(-2,2)
ax.set_ylim(-2,2)
```

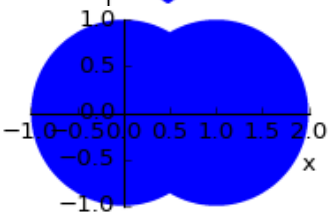


Для построения более сложных областей следует использовать булевы операции между неравенствами. В следующих примерах мы демонстрируем применение функций `And()`, `Or()` и `Not()`. Фактически мы здесь строим диаграммы Эйлера для соответствующих операций.

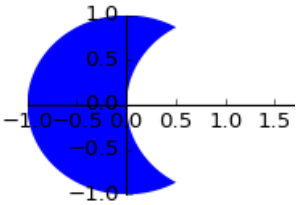
```
var('x y U V')
U=x**2+y**2<1
V=(x-1)**2+y**2<1
plot_implicit(And(U, V),(x, -1, 2), (y, -1, 1))
```



```
var('x y U V')
U=x**2+y**2<1
V=(x-1)**2+y**2<1
plot_implicit(Or(U, V),(x, -1, 2), (y, -1, 1))
```

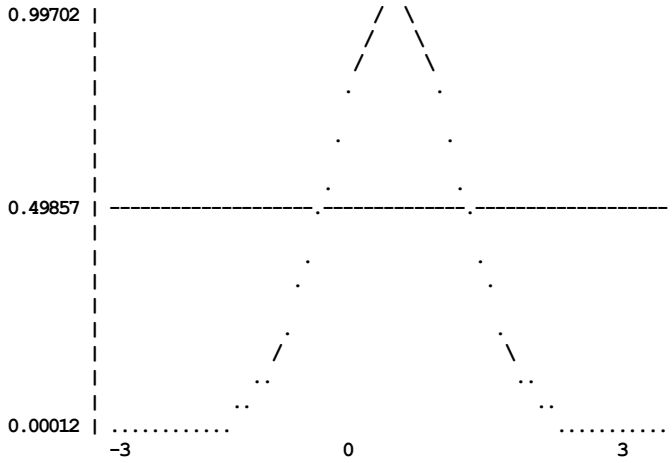


```
var('x y U V')
U=x**2+y**2<1
V=(x-1)**2+y**2<1
plot_implicit(And(U,Not(V)),(x, -1, 2), (y, -1, 1))
```



Упомянем еще об одной «экзотической» функции `textplot(expr,`x_{min}`,`x_{max}`)`, которая имитирует построение графика символьного выражения `expr` на отрезке $[x_{min}, x_{max}]$ в текстовом режиме.

```
from sympy import *
var('x')
textplot(E**-x**2,-3,3)
```



5.5 Символьное решение дифференциальных уравнений

Ранее мы говорили, что для записи уравнений SymPy используется функция Eq(). Фактически запись Eq(выражение1,выражение2) создает специальный объект, который передается функциям, применяемым для решения дифференциальных уравнений (солверам).

```
>>> from sympy import *
>>> x = symbols('x')
>>> y=Function('y')          # объявление символьной функции
>>> pprint(dsolve(Eq(y(x).diff(x,x)+y(x),exp(x)),y(x))) #y''+y=e^x
                                    x
                                   e
y(x) = C1*sin(x) + C2*cos(x) + ----
                                   2
```

Решением является выражение $C_1 \sin x + C_2 \cos x + \frac{1}{2} e^x$. Обратите внимание, что в решении появились две неопределенные константы C1 и C2.

Для решения обыкновенных дифференциальных уравнений (ОДУ) вначале объявляется символьная функция, которая будет представлять решение. Для этого имя этой функции определяется как символьная переменная в функции symbols() с опцией cls=Function. Например, команда f = symbols('f', cls=Function) создает одну такую функцию. Можно использовать эквивалентную ей запись f=Function('f'). После этого выполняется инструкция решения ДУ. Объект Eq() может создаваться сразу внутри вызова функции dsolve(), или отдельно перед ее вызовом.

Решим ОДУ $f'' - 2f' + f = \sin x$. Вначале создаем символьную функцию f, а затем объект diffeq, представляющий уравнение.

```
>>> f = symbols('f', cls=Function)
>>> diffeq = Eq(f(x).diff(x, x) - 2*f(x).diff(x) + f(x), sin(x))
```

```
>>> pprint(diffeq)
                      2
         d           d
f(x) - 2*--(f(x)) + ---(f(x)) = sin(x)
         dx           2
                     dx
```

Для решения используется функция `dsolve(diffeq,f(x))` с первым аргументом – объектом уравнения. Вторым аргументом является искомое выражение.

```
>>> dsolve(diffeq, f(x))
Eq(f(x), (C1 + C2*x)*exp(x) + cos(x)/2)
```

Последний пример иллюстрирует обычную последовательность инструкций, используемых для решения ОДУ. Она состоит в следующем:

- объявляется символьная независимая переменная `x` и символьная функция `f`;
- создается объект, представляющий уравнение;
- решается дифференциальное уравнение с помощью функции `dsolve()`.

Решим несколько примеров.

```
>>> x=Symbol('x')
>>> f=Function('f')
>>> eq=Eq(f(x).diff(x,x)+f(x),0)
>>> pprint(dsolve(eq,f(x)))                    # f''+f=0
f(x) = C1*sin(x) + C2*cos(x)
```

Договоримся, что далее решать ОДУ будем в IPython в режиме отображения Latex, т.е. будем считать, что во всех последующих примерах этого параграфа включен режим «красивой» печати.

```
init_printing(use_latex=True)
```

Функция `dsolve()` может использовать несколько различных методов решения ОДУ. Чтобы получить список методов, которые можно использовать для решения уравнения, следует использовать инструкцию `classify_ode(уравнение, выражение/функция)`.

```
classify_ode(eq, f(x))
('separable',
 '1st_exact',
 'almost_linear',
 '1st_power_series',
 'lie_group',
 'separable_Integral',
 '1st_exact_Integral',
 'almost_linear_Integral')
```

Опция `hint='method_name'` функции `dsolve()` задает метод решения.

```
f = Function('f')
eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
rez=dsolve(eq, hint='1st_exact'); rez
```

$$\left[f(x) = -acos\left(\frac{C_1}{\cos(x)}\right) + 2\pi, \quad f(x) = acos\left(\frac{C_1}{\cos(x)}\right)\right]$$

```
rez=dsolve(eq, hint='separable_Integral'); rez
```

$$\int^{f(x)} \frac{\sin(y)}{\cos(y)}\,dy = C_1 + \int -\frac{\sin(x)}{\cos(x)}\,dx$$

Опции `hint` можно присвоить значение `all` . Тогда вы получите словарь решений в форме `метод:решение`.

```
rez=dsolve(eq, hint='all');
```

Значение `all_Integral` также возвращает все интегралы ОДУ. Опции `hint` можно присвоить значение `'best'`. Тогда функция вернет простейшее (по ее мнению) решение ОДУ.

```
rez=dsolve(eq, hint='best'); rez
```

$$f(x) = -x^2/2 - x^4/3 + C_1 + \mathcal{O}(x^6)$$

Опция `ics` позволяет задавать граничные условия. Она должна принимать форму `{f(x0): y0,f(x).diff(x).subs(x, x1):y1[,...]}`. В настоящий момент эта опция реализована только для метода решения в виде рядов для ОДУ 1 – го порядка (`hint='1st_power_series'`). В этом случае условие должно принимать вид `{f(x0): y0}`. При решении в виде рядов опция `n` определяет порядок независимой переменной, до которого строится степенной ряд решения. Опция `simplify` позволяет использовать символьные алгоритмы преобразования полученного выражения.

Пример. Решим задачу Коши $u''(t) - u'(t) - 2u = -2$, $u(0) = 0, u'(0) = 1$.

```
import numpy as np
from sympy import *
from IPython.display import *
import matplotlib.pyplot as plt
init_printing(use_latex=True)

var('t C1 C2')
u = Function("u")(t)          # здесь u – выражение (не функция)
de = Eq(u.diff(t,t)-u.diff(t)-2*u,-2)
display(de)
```

$$-2u(t) - \frac{d}{dt}u(t) + \frac{d^2}{dt^2}u(t) = -2$$

Функция `display()` отображает объекты Python в любой его оболочке. Она загружается из модуля `IPython.display`. Вид результата зависит от оболочки и, возможно, режима отображения. Здесь мы просто напечатали наше дифференциальное уравнение.

Теперь решаем ДУ и печатаем результат.

```
des = dsolve(de,u)          # здесь u – выражение, а не функция
display(des)
```

$$u(t) = C_1 e^{-t} + C_2 e^{2t} + 1$$

Найдем постоянные C1 и C2, которые удовлетворяют начальным условиям. Вначале выполним подстановку `t=0` в правую часть полученного выражения `des`. Здесь метод `rhs` (**r**ight **h**and **s**ide) выбирает правую часть символьного выражения.

```
eq1=des.rhs.subs(t,0);
eq1
```

$$C_1 + C_2 + 1$$

Дифференцируем правую часть выражения `des`, и делаем подстановку `t=0`.

```
eq2=des.rhs.diff(t).subs(t,0)
eq2
```

$$-C_1 + 2C_2$$

Выражения `eq1` и `eq2` должны равняться 0 и 1 (начальным значениям). Решаем систему уравнений относительно неизвестных `C1` и `C2`.

```
seq=solve([eq1,eq2-1],C1,C2)
seq
```

$$\{C_1 : -1, \quad C_2 : 0\}$$

Строим результирующее выражение, которое представляет решение нашей задачи, и «красиво» его печатаем.

```
rez=des.rhs.subs([(C1,seq[C1]),(C2,seq[C2])])
F = Lambda(t,rez)
display(Latex('$u(t) = ' + str(latex(F(t))) + '$'))
```

$$u(t) = 1 - e^{-t}$$

Здесь функция `sympy.Lambda(x,expr)` создает символьную лямбда – функцию аналогично лямбда – функции Python `'lambda x: expr'`. Функция нескольких переменных создается с использованием синтаксиса `Lambda((x,y,...),expr)`.

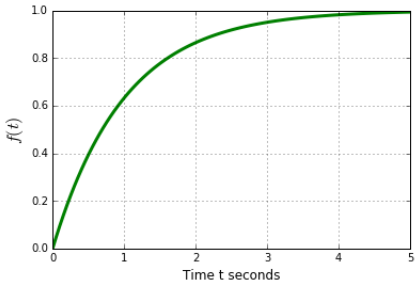
Функция `sympy.latex()` преобразует символьное выражение в строку TeX кода. А функция `Latex()` (из модуля `IPython.display`) преобразует строку, содержащую TeX код, в объект, который может отобразить функция `display()`.

Затем преобразуем символьное выражение в лямбда–функцию, поддерживающую работу с массивами модуля `numpy`.

```
f=lambdify(t, rez, "numpy")
```

Теперь, используя функции модуля `matplotlib.pyplot`, строим график решения.

```
x = np.linspace(0,5,100)
plt.grid(True)
plt.xlabel('Time t seconds',fontsize=12)
plt.ylabel('$f(t)$',fontsize=16)
plt.plot(x,f(x),color='#008000', linewidth=3)
plt.show()
```



■

Приведем примеры решения систем дифференциальных уравнений. При этом, для обозначения производных в уравнениях можно использовать как функцию `Derivative()`, так и метод `diff()`. Заметим также, что в функции `dsolve` указывать имена искомых функций не требуется – это будет выполняться автоматически.

Пример. Решим систему ОДУ

$$\begin{cases} x'(t) = x \cdot y \cdot \sin t \\ y'(t) = y^2 \cdot \sin t \end{cases}$$

```
# включен режим   init_printing(use_latex=True)
from sympy import *
t = symbols('t')
x, y = symbols('x, y', function=True)
eq = (Eq(Derivative(x(t),t),x(t)*y(t)*sin(t)), \
            Eq(Derivative(y(t),t),y(t)**2*sin(t)))
rez=dsolve(eq)
display(list(rez))
```

$$\left[x(t) = -\frac{e^{C_1}}{C_2 e^{C_1} - \cos(t)}, \quad y(t) = -\frac{1}{C_1 - \cos(t)}\right]$$

Пример. Решим систему ОДУ

$$\begin{cases} y'(x) = -z(x) \\ z'(x) = -y(x) \end{cases}$$

```
from sympy import *
x = symbols('x')
y, z = symbols('y, z', function=True)
eq = (Eq(Derivative(y(x),x), -z(x)), Eq(Derivative(z(x),x), -y(x)))
rez=dsolve(eq)
display(list(rez))
```

$$[y(x) = -C_1 e^{-x} - C_2 e^{x}, \quad z(x) = -C_1 e^{-x} + C_2 e^{x}]$$

Пример. Решим систему ОДУ

$$\begin{cases} x'(t) = -x + z \\ y'(t) = -y - z \\ z'(t) = y - z \end{cases}$$

```
t = symbols('t')
x,y,z = symbols('x,y,z', function=True)
eq1=Eq(x(t).diff(t),-x(t)+z(t))
```

```
eq2=Eq(y(t).diff(t),-y(t)-z(t))
eq3=Eq(z(t).diff(t),y(t)-z(t))
rez=dsolve((eq1,eq2,eq3))
display(rez[0])
display(rez[1])
display(rez[2])
```

$$x(t) = C_1 e^{-t} + C_2 e^{-t}\sin(t) - C_3 e^{-t}\cos(t)$$

$$y(t) = -C_2 e^{-t}\sin(t) + C_3 e^{-t}\cos(t)$$

$$z(t) = C_2 e^{-t}\cos(t) + C_3 e^{-t}\sin(t)$$

■

В пакете `sympy` имеется модуль `sympy.solvers.pde`, содержащий функции, предназначенные для символьного решения дифференциальных уравнений в частных производных (ДУЧП). Поскольку таких уравнений известно немного, то возможности этого модуля ограничены. Основной функцией этого пакета является функция `sympy.solvers.pde.pdsolve(eq,...)`, где `eq` является любым допустимым ДУЧП.

Пример. Решим ДУЧП

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = \frac{1}{x \cdot y}.$$

```
from sympy import *
from sympy.solvers.pde import pdsolve
f = Function('f')
u = f(x, y)
ux = u.diff(x)
uy = u.diff(y)
eq=Eq(ux+uy,1/(x*y))
pdsolve(eq)
```

$$f(x, y) = \frac{1}{x-y}((x-y)F(x-y) - \log(x) + \log(y))$$

Общее решение дифференциального уравнения в частных производных, если оно вообще может быть найдено, должно включать в себя произвольные функции. В последнем решении такой функцией является $F(x-y)$.

Пока неизвестная функция и ее производные входят в уравнение линейно можно надеяться получить общее решение.

```
f = Function('f')
u = f(x, y)
ux = u.diff(x)
uy = u.diff(y)
eq=Eq(x*ux+y*uy,exp(x*y))
pdsolve(eq)
```

$$f(x, y) = F(y/x) + Ei(xy)/2$$

Здесь `F(...)` произвольная функция, а `Ei(x)` - интегральная показательная функция.

Если общее решение не может быть найдено, то система выводит сообщение

```
NotImplementedError: psolve: Cannot solve ...
```

В пакете SymPy имеется большое количество функций, которые предназначены для решения специальных типов обыкновенных дифференциальных уравнений. Однако, в настоящее время, возможности символьного решения дифференциальных уравнений весьма ограничены и на практике чаще используются функции, реализующие численные методы. С ними мы познакомимся в следующей главе.

5.6 Совместное использование символьной и численной математики

Имеется несколько причин включать символьные вычисления в числовой код. Например, это случается тогда, когда мы хотим сравнить известное аналитическое решение с решением, полученным на сетке. Довольно часто перед выполнением численных расчетов, нам надо выполнить какие–либо символьные преобразования, например, вычислить производные аналитически заданной функции. То же касается вычисления интегралов или дифференциальных уравнений, если конечно их решения можно получить в аналитическом виде. В любом случае мы можем вставлять символьные выражения в код, и использовать их не только для вычисления значений.

В этом параграфе мы опишем способы использования возможностей пакета SymPy совместно с другими пакетами научного Python. Они сводятся к преобразованию символьных выражений в функции Python или в функции, способные работать с данными другого модуля. Встречаются ситуации, когда из функции Python нужно получить символьное выражение.

Пусть дано символьное выражение `expr`.

```
>>> x=symbols('x')
>>> expr=x**4+4
```

Преобразуем его в функцию Python.

```
>>> fn=lambda t : expr.subs({x:t})
>>> fn(1)
5
```

Достаточно выполнить подстановку аргумента лямбда – функции `t` в символьное выражение `expr` в теле функции.

Подстановка `subs()` в символьное выражение удобна, если его значение надо вычислить в одной точке. Но если вы намерены вычислять значение выражения во многих точках, то имеется более эффективное решение – функция `lambdify()`. Она возвращает лямбда – функцию и при этом конвертирует SymPy имена (функций) в имена функций подходящего научного модуля, например, `numpy`. Формат вызова этой функции следующий:

```
fun=lambdify(args, expr, modules=None[,...])
```

Здесь `args` является именем аргумента лямбда – функции или кортежем имен. Второй параметр `expr` является символьным выражением, которое будет преобразовываться. Третий параметр `modules`, если он задан, может быть

строковым именем одного из модулей: 'math', 'mpmath', 'numpy', 'numexpr', 'sympy'. Третий аргумент может быть также словарем с парами замен вида sympy_name : func_name. Если третий аргумент не задан, то выполняется замена (насколько это возможно), функциями модуля math, numpy или mpmath (в таком порядке).

```
>>> import numpy as np
>>> from sympy import *
>>> x=symbols('x')
>>> expr = sin(x)+cos(x)
>>> g= lambdify(x, expr, "numpy")
>>> f=np.arange(5)
>>> g(f)
array([ 1. ,1.38177329, 0.49315059, -0.84887249, -1.41044612])
```

Таким образом, если мы хотим, чтобы преобразованная из символьного выражения функция, работала с массивами модуля numpy, то ее следует преобразовать в лямбда – функцию, используя опцию modules='numpy' (в модуле numpy свои элементарные и специальные функции, которые умеют работать с массивами).

Рассмотрим теперь обратную задачу преобразования функции Python в символьное выражение.

Пусть выражение, возвращаемое функцией, содержит только стандартные математические операции над аргументом функции. Тогда для получения символьного выражения, вычисляемого по той же формуле, достаточно вызвать функцию с символьным аргументом. Например, пусть дана функция Python.

```
>>> fn=lambda t: t**4+4
>>> fn(2)
20
```

Для получения символьного выражения достаточно вызвать эту функцию с символьным аргументом.

```
>>> x=symbols('x')
>>> z=fn(x)
>>> type(z)
<class 'sympy.core.add.Add'>
>>> factor(fn(x))       # разложение на множители символьного выражения
(x**2 - 2*x + 2)*(x**2 + 2*x + 2)
```

Как видите, результатом вызова функции fn с символьным аргументом является символьное выражение, с которым можно работать как с любым символьным выражением. В частности, результат можно дифференцировать.

```
>>> def f(x):
    return x**4
>>> x=S('x')
>>> f(x).diff(x)
4*x**3
>>> f(x).diff(x,2)
12*x**2
```

Однако этот прием не работает, когда выражение, возвращаемое функцией, содержит функции, вызываемые из модулей. Например, такое простое решение не работает, когда функция пользователя содержит функции модуля `math`. Дело в том, что подстановка символьных переменных в качестве аргументов функций модуля `math` приводит к ошибке. Но в модуле SymPy имеется функция `lambdify()`, которая умеет преобразовывать различные функции, в частности модуля `math`, в аналогичные функции модуля SymPy (или любого другого модуля, содержащего одноименные функции).

```
>>> from math import sin
>>> fn=lambda t: sin(t)**2
>>> fn(1)
0.7080734182735712
>>> from sympy import *
>>> x=symbols('x')
>>> g= lambdify(x, fn(x), "sympy")
>>> g(x)
sin(x)**2
>>> g(x).diff(x)
2*sin(x)*cos(x)
```

Таким образом, функция `lambdify()` позволяет преобразовывать функции Python и символьные выражения друг в друга.

Иногда мы сталкиваемся с задачей преобразования строкового выражения в Python функцию. Для этого можно использовать функцию `eval()`.

```
>>> eq = "x**2"
>>> func = lambda x: eval(eq)
>>> func(5)
25
```

А для преобразования строкового выражения в символьное подходит функция `sympify()`.

```
>>> from sympy import *
>>> f=sympify('x**4'); f
x**4
>>> x=S('x')
>>> f.diff(x), f.diff(x,x)
(4*x**3, 12*x**2)
```

Если символьное выражение содержит только знаки математических операций, то для его преобразования в строку можно использовать функцию `str()`.

```
>>> f=x**3-2*x**2+x
>>> str(f)
'x**3 - 2*x**2 + x'
```

Для выполнения кода, который содержится в строке, вы можете применить функцию `eval()`, но для этих целей имеется специальная функция `exec()`.

```
>>> mc = 'print("hello world")'
>>> exec(mc)
hello world
```

Функцию `eval()` следует применять тогда, когда нужно получить значение.

```
>>> a=eval('2+2');a
4
```

Пример. Построить график символьного выражения одной символьной переменной, а также графики его первой и второй производных.

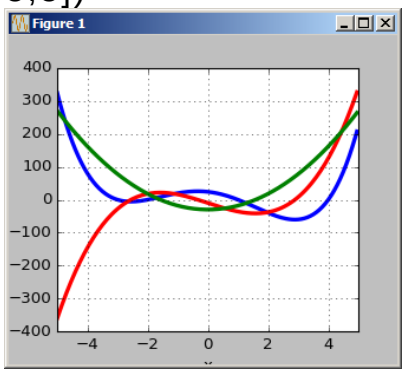
Первое решение состоит в использовании графической функции `plot()` пакета SymPy.

```
>>> import sympy as sp
>>> x=sp.symbols('x')
>>> f0=x**4 - 15*x**2 - 10*x + 24
>>> f1=sp.diff(f0,x);f1
4*x**3 - 30*x - 10
>>> f2=sp.diff(f0,x,2); f2
6*(2*x**2 - 5)
>>> sp.plot(f0,f1,f2,(x,-5,5))
```

Поскольку управление графикой в графических функциях SymPy ограничено, то, возможно, вы пожелаете использовать графику других пакетов, а для этого потребуется преобразование символьного выражения.

Второе решение состоит в преобразовании символьного выражения и его производных в лямбда – функцию и построении ее графика функцией `plot()` модуля `mpmath`.

```
>>> import sympy as sp
>>> import mpmath as mp
>>> x=sp.symbols('x')
>>> f0=x**4 - 15*x**2 - 10*x + 24
>>> f1=sp.diff(f0,x)
>>> f2=sp.diff(f0,x,2)
>>> g0=sp.lambdify(x,f0,'mpmath')
>>> g1=sp.lambdify(x,f1,'mpmath')
>>> g2=sp.lambdify(x,f2,'mpmath')
>>> mp.plot([g0,g1,g2],[-5,5])
```

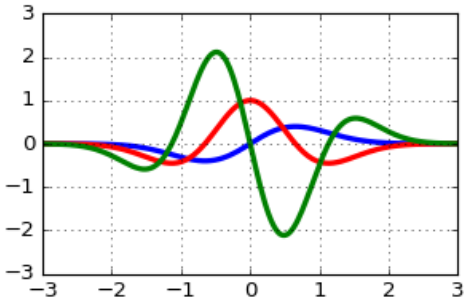


Здесь мы создали символьное выражение `f0= x**4-15*x**2-10*x+24` и вычислили его первую и вторую производные `f1, f2`. Затем символьные выражения, представляющие исходную функцию `f0` и ее первую и вторую производные, преобразовали в лямбда – функции с помощью функции `sympy.lambdify()`, которой третьим аргументом указали желаемый тип

результата (‘mpmath’). Для построения графиков использовали функцию plot() модуля mpmath.

Третье решение состоит в преобразовании символьного выражения и его производных в лямбда – функцию и построении ее графика функцией plot() модуля matplotlib.pyplot.

```
>>> import matplotlib.pyplot as plt
>>> import sympy as sp
>>> import numpy as np
>>> x=sp.symbols('x')
>>> f0=sp.sin(x)*sp.exp(-x**2)
>>> f1=sp.diff(f0,x)
>>> f2=sp.diff(f0,x,2)
>>> h0=sp.lambdify(x,f0,'numpy')
>>> h1=sp.lambdify(x,f1,'numpy')
>>> h2=sp.lambdify(x,f2,'numpy')
>>> t = np.linspace(-3, 3, 121)
>>> plt.plot(t,h0(t),'b',t,h1(t),'r',t,h2(t),'g', linewidth=3)
>>> plt.show()
```



Здесь, имея символьные выражения f0, f1 и f2, мы построили лямбда – функции h0(x), h1(x), h2(x). Обратите внимание на третий аргумент функций lambdify(), которому присвоено значение 'numpy'. Это нужно для того, чтобы имена функций sin и exp, используемые в символьных выражениях f0, f1 и f2, были преобразованы в соответствующие функции модуля numpy. Затем лямбда – функции h0(x), h1(x), h2(x) мы используем для создания массивов значений, которые нужны для построения графиков функцией matplotlib.pyplot.plot().

Пример. Нарисовать в одном графическом окне символьный график и график matplotlib, не прибегая к преобразованиям выражений.

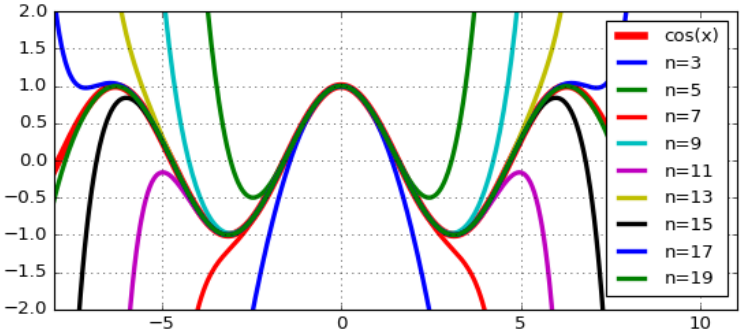
```
%matplotlib qt
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

x=sp.symbols('x')
p1=sp.plot(x**3-x,(x,-1.5,1.5))   # sympy график
fig=plt.gcf()
fig.set_facecolor('white')
ax=fig.gca()                      # получение осей sympy графика
```

```
t=np.linspace(-1.5,1.5,16)
z=t**3-t
ax.plot(t,-z,'-sr', linewidth=3)     # matplotlib график
ax.grid(True)
ax.axis('equal')
```

Вначале мы построили график символьного выражения `x**3-x`. Затем получили ссылку на объект `axes` этого графика. Этот объект принадлежит модулю `matplotlib.pyplot` и имеет метод `plot()` этой библиотеки.

Пример. Построить график функции $y = \cos x$ и графики ее сумм Тейлора различного порядка.

```
%matplotlib qt
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
x = symbols('x')
f = sp.cos(x)

fvec = np.vectorize(lambda val: f.subs(x, val).evalf())
z = np.linspace(-8,8, num=300)
fig = plt.figure(facecolor='white')
plt.plot(z, fvec(z), linewidth=5,color='r',label='cos(x)')
ax = fig.gca()
for order in range(3, 20, 2):
    s = f.series(x, 0, order)
    svec = np.vectorize(lambda val: s.removeO().subs(x, val))
    ax.plot(z, svec(z), '-', label='n=%i' % order, linewidth=3)
ax.set_xlim(-8, 11)
ax.set_ylim(-2, 2)
ax.legend(fontsize=12)
ax.grid(True)
```

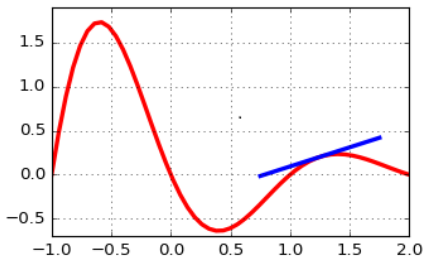


Пример. *График функции и ее касательной.*

Уравнение касательной к кривой $y = f(x)$ в точке x_0 имеет вид $y = f(x_0) + f'(x_0)(x - x_0)$. Чтобы его создать нужно использовать символьное дифференцирование, и правую часть уравнения задавать как символьное выражение. Для построения графика будем использовать функцию `matplotlib.pyplot.plot`, которая не умеет работать с символьными

выражениями. Поэтому требуется преобразование символьных выражений, представляющих уравнения кривой и ее касательной, к виду, пригодному для функции `plot`. Ниже приведен код, учитывающий эту особенность.

```
import numpy as np
import matplotlib.pyplot as plt
from sympy import *
x = symbols('x')
x0=1.25                 # точка касания
y=-exp(-x)*sin(pi*x)    # символьное уравнение кривой
y1=y.diff(x)
y0=y.subs(x,x0)
y10=y1.subs(x,x0)       # значение производной в точке
func=lambdify(x, y, "numpy")                          # уравнение кривой
tangent=lambdify(x, y0+y10*(x-x0), "numpy")    # уравнение касательной

dlt=0.5                 # полуширина (по оси x) отрезка касательной
Xt=np.linspace(x0-dlt,x0+dlt,21)
# если производная ноль, то символьное уравнение
# касательной вырождается в константу (число)
if y10==0:
    Yt=y0*np.ones(np.size(Xt))
else:
    Yt=tangent(Xt)

Xf=np.linspace(-1,2,51)
Yf=func(Xf)
plt.plot(Xf,Yf,linewidth=3,c='r')
plt.plot(Xt,Yt, linewidth=3,c='b')
plt.ylim([1.1*Yf.min(),1.1*Yf.max()])
plt.grid(True)
plt.show()
```



Отметим еще раз, что существенным моментом этого кода является использование функции `lambdify` для преобразования двух символьных выражений в функции `func` и `tangent` модуля `numpy`.

```
func=lambdify(x, y, "numpy")
tangent=lambdify(x, y0+y10*(x-x0), "numpy")
```

Другим ньюансом нашего кода является проверка условия `if y10==0 ...`. Если производная `y10` в точке `x0` обращается вноль, то инструкция `tangent=lambdify(x, y0+y10*(x-x0), "numpy")` создаст не функцию `tangent(x)`, а константу. Тогда инструкция `Yt=tangent(Xt)` создаст не

вектор значений Yt, а одно число, и график не будет нарисован (вы получите сообщение об ошибке).

Пример. *Касательная и нормаль к эллипсу, заданному неявным уравнением* $\frac{x^2}{a^2}+\frac{y^2}{b^2}-1=0$.

Если кривая задана неявным уравнением $F(x,y)=0$ и точка (x_0,y_0) принадлежит кривой, то уравнение касательной к кривой в этой точке имеет вид

$$F'_x(x_0,y_0)(x-x_0)+F'_y(x_0,y_0)(y-y_0)=0, \quad (1)$$

а уравнение нормали

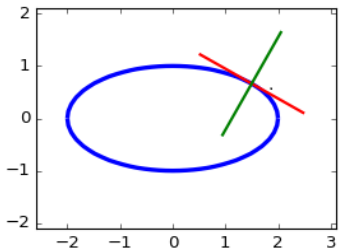
$$F'_x(x_0,y_0)(y-y_0)-F'_y(x_0,y_0)(x-x_0)=0. \quad (2)$$

В следующем коде в начале создается символьное выражение $F(x,y)$. Абсцисса точки касания x0 задается, а ордината y0 определяется путем решения уравнения $F(x_0,y)=0$. Учитывая, что функция solve может вернуть список из одного или двух корней, для y0 мы выбираем последнее значение, поскольку оно есть всегда. Значения $F'_x(x_0,y_0)$ и $F'_y(x_0,y_0)$ находятся символьным дифференцированием с последующей подстановкой координат точки касания (x_0,y_0). После этого, с помощью функции lambdify мы создаем numpy функцию $F(x,y)$ и функции правых частей уравнений (1) и (2), которые и используются при построении графика.

```
import numpy as np
import matplotlib.pyplot as plt
from sympy import *
x,y = symbols('x y')
a=2; b=1
F=x**2/a**2+y**2/b**2-1
Fx=F.diff(x)
Fy=F.diff(y)
x0=1.5                          # x координата точки касания
y0=solve(F.subs(x,x0),y)[-1]    # находим y координаты точек касания
                                # и выбираем последнюю из списка
Fx0=Fx.subs([(x,x0),(y,y0)])    # значение производной Fx в точке касания
Fy0=Fy.subs([(x,x0),(y,y0)])    # значение производной Fy в точке касания
# создание numpy функций из символьных выражений
Fun=lambdify((x,y), F, "numpy")
Tan=lambdify((x,y), Fx0*(x-x0)+Fy0*(y-y0), "numpy")
Norm=lambdify((x,y), Fx0*(y-y0)-Fy0*(x-x0), "numpy")
# построение графиков эллипса, касательной и нормали
t=np.linspace(-a,a,31)
Xf,Yf=np.meshgrid(t,t)
plt.figure()
plt.contour(Xf, Yf, Fun(Xf,Yf), [0] , linewidths=3,colors='b' )
tx=np.linspace(x0-1.0,x0+1.0,21)
```

```
y0=float(y0)       # преобразование символьного значения в числовое
ty=np.linspace(y0-1,y0+1,21)
Xt,Yt=np.meshgrid(tx,ty)
plt.contour(Xt, Yt, Tan(Xt,Yt), [0] , linewidths=2,colors='r' )
plt.contour(Xt, Yt, Norm(Xt,Yt), [0] , linewidths=2,colors='g' )
#plt.gcf().gca().axis([-3,3,-3,3]);
plt.gcf().gca().axis('equal');
plt.gcf().set_facecolor('white')
plt.show()
```

Полученное изображение эллипса, касательной и нормали к нему, приведено на следующем рисунке.



6. Научные вычисления с пакетом SciPy

Пакет SciPy является библиотекой математических процедур. Многие из них являются Python оболочками, обеспечивающими доступ к библиотекам, написанным на других языках программирования. Поскольку функции этих библиотек хранятся в откомпилированном виде, то эти процедуры обычно работают очень быстро. В этой главе мы переходим к описанию возможностей модулей SciPy.

6.1 Численное интегрирование.

6.1.1 Вычисление интегралов.

Многие прикладные задачи сводятся к определению интегралов – одинарных, двойных, тройных или многократных. Однако аналитически вычислить удается немногие из них. Тогда интеграл можно найти численно. В этом параграфе мы рассмотрим некоторые из функций пакета `scipy.integrate`, предназначенные для численного интегрирования.

Приведем список наиболее часто используемых функций интегрирования из модуля `scipy.integrate`.

Функция	**Описание**
`quad`	однократное численное интегрирование
`dblquad`	вычисление двойного интеграла
`tplquad`	вычисление тройного интеграла
`nquad`	вычисление n – кратного интеграла
`cumtrapz`	вычисление первообразной

Вычислим определенный интеграл $\int_{-1}^{1} e^{-x} \sin x \, dx$.

Первым шагом является создание функции, вычисляющей подынтегральное выражение

```
def f( x ):
    return np.exp(-x)*np.sin(x)
```

Функция `f(x)` должна содержать комбинацию стандартных математических функций, имена которых имеются в модуле `numpy` и `scipy` (или их подмодулях, например, в `scipy.special`). Она также может содержать вызовы функций пользователя.

Для вычисления интеграла предназначена функция `quad`. Первым аргументом ей нужно передать ссылку на функцию `f` (имя функции), а вторым и третьим — нижний (`a`) и верхний (`b`) пределы интегрирования.

```
I=quad(f, a, b)
```

Таким образом, вычислить интеграл можно следующим образом:

```
from scipy.integrate import quad
import numpy as np
def f( x ):
   return np.exp(-x)*np.sin(x)
I=quad(f, -1, 1)
print(I)
(-0.6634936666312413, 1.2783578503385453e-14)
```

Обычно (как в этом примере) функция `quad` возвращает кортеж, содержащий значение интеграла и абсолютную погрешность. Однако при задании опции `full_output=True` функция `quad` будет возвращать словарь с дополнительной информацией.

По умолчанию функция `quad` вычисляет приближенное значение интеграла с абсолютной и относительной погрешностями `epsabs=1.49e-08, epsrel=1.49e-08`. Здесь `epsabs` и `epsrel` имена опций, задающих эти погрешности.

Функция `f(x,...)` может принимать много аргументов, но интегрирование будет выполняться только по первому. Для передачи подынтегральной функции дополнительных аргументов используется опция `args`, которой следует передавать кортеж с дополнительными аргументами.

```
f = lambda x,y : x**2+y**2
y,err = quad(f, 0, 3, args=(1,))   # вычисление ∫₀³(x²+y²)dx при y=1
print(y)
12.0
```

Вычислим интеграл $\int_0^1 \left(a x^2 + b x + c\right) d x$ с параметрами $a = 3, b = 2, c = 1$. Это можно сделать, используя следующий код:

```
from scipy.integrate import quad
def f(x, a, b,c):
   return a*x**2+b*x+c
I = quad(f, 0, 1, args=(3,2,1))
```

```
print(I)
(3.0, 3.3306690738754696e-14)
```
Иногда удобно не передавать вспомогательные аргументы через опцию `args`, а использовать лямбда – оболочку вокруг подынтегральной функции. Вот, например, как можно вычислить предыдущий интеграл.
```
def f(x, a, b,c):
    return a*x**2+b*x+c
I = quad(lambda x: f(x,3,2,1), 0, 1)
print(I)
(3.0, 3.3306690738754696e-14)
```
Можно численно находить интеграл на бесконечном участке вещественной оси. Это значит, что верхний и/или нижний пределы интегрирования могут быть бесконечными. Для обозначения пределов $\pm\infty$ используется идентификатор `numpy.inf`. Например, вычислим интеграл $\int_0^{\infty} e^{-x^2}\,dx$.
```
i1 = quad((lambda x: np.exp(-x**2)), 0, np.inf)
print(i1)
(0.8862269254527579, 7.101318390915439e-09)
```
Этот интеграл вычисляется аналитически. Сравним значения.
```
x=smp.symbols('x')
s1=smp.integrate(smp.exp(-x**2),(x,0,smp.oo))  # ∫_0^∞ e^{-x^2} dx = √π/2
print(s1)
print(s1.evalf())
sqrt(pi)/2
0.886226925452758
```
Вычислим еще один несобственный интеграл символьно и численно.
```
from scipy.integrate import quad, dblquad, tplquad
import numpy as np
i1 = quad(lambda x: 1/(1+x**2), 0, np.inf)        # ∫_0^∞ 1/(1+x^2) dx
print(i1)
(1.5707963267948966, 2.5777915205989877e-10)
```
Сравним с точным значением
```
x=smp.symbols('x')
s1=smp.integrate(1/(1+x**2),(x,0,smp.oo))        # ∫_0^∞ 1/(1+x^2) dx = π/2
print(s1)
pi/2
```
Подынтегральная функция сама может содержать интеграл. Например, вычислим интеграл $\int_0^1\int_0^x \left(x^2+y^2\right) dy\,dx$. Это можно сделать, используя следующий код:

```
def f(y,u):                          # аргумент y указываем первым
    return y**2+u**2
def g(x):
    return quad(f,0,x,args=(x,))[0]  # возвращаем только значение интеграла
I=quad(g,0,1)
print('I=',I)
I=(0.3333333333333333, 3.700743415417188e-15)
```

Проверим результат, используя символьное интегрирование.

```
x,y=smp.symbols('x y')
si=smp.integrate(x**2+y**2,(y,0,x),(x,0,1))
print(si)
1/3
```

Однако такие интегралы проще вычислять, если использовать функцию dblquad. Она применяется для вычисления повторных интегралов вида $\int_a^b \int_{g(x)}^{h(x)} f(x,y)\,d\,y\,d\,x$. Функция dblquad имеет следующий синтаксис:

```
scipy.integrate.dblquad(ffun, a, b, gfun, hfun[,...])
```

Здесь ffun является именем подынтегральной функции (двух переменных). При этом имя внутренней переменной интегрирования y при создании функции ffun должно быть указано первым. Это значит, что инструкция определения функции должна иметь вид:

```
def ffun(y,x):        # порядок аргументов важен
    ...
```

Аргументы a и b задают значения нижнего и верхнего пределов интегрирования по переменной x (пределы внешнего интеграла), gfun и hfun являются именами функций, определяющих нижний и верхний пределы интегрирования по переменной y (пределы внутреннего интеграла). Имеются также опции args, epsabs, epsrel, значение которых такое же, как и для функции quad.

Вот, например, как следует вычислять предыдущий интеграл. Вначале определим функции f, g и h.

```
f=lambda y,x: x**2+y**2
g=lambda x: 0
h=lambda x: x
```

Обратите внимание на то, что даже если функции g и h нижнего и верхнего пределов интегрирования являются константами (как g(x) в нашем случае), их следует задавать функциями. Затем используем функцию dblquad.

```
I=dblquad(f, 0, 1, g, h)          # ∫_0^1 ∫_0^x (x^2+y^2) d y d x
print(I)
(0.3333333333333333, 3.700743415417188e-15)
```

Как и для однократного интеграла, функция dblquad возвращает два числа: значение интеграла и абсолютную погрешность (absolute uncertainty).

Вычислим еще один повторный интеграл $\int_{-1}^{1}\int_{0}^{\sqrt{1-x^2}}\sqrt{1-x^2-y^2}\,dy\,dx$.

```
f=lambda y,x: np.sqrt(1-x**2-y**2)
g=lambda x: 0
h=lambda x: np.sqrt(1-x**2)

I=dblquad(f, -1, 1, g, h)          # ∫_{-1}^{1}∫_{0}^{√(1-x²)} √(1-x²-y²) dy dx

print(I)
(1.0471975511965979, 1.162622832669544e-14)
```

Этот интеграл представляет объем четверти шара (интеграл по полукругу от функции $z=\sqrt{1-x^2-y^2}$) и нам известно его точное значение $\pi/3$. Сравним это значение с полученным приближенным значением интеграла.

```
print(np.pi/3)
1.0471975511965976
```

Как видим, значения практически совпадают.

Функцию `dblquad` можно использовать для вычисления двойных интегралов, предварительно «обнулив» функцию вне области интегрирования. Однако этот подход имеет свои недостатки. Например, чтобы проинтегрировать функцию $f(x,y)=1-x^2-y^2$ по области единичного круга можно проинтегрировать функцию

$$F(x,y)=\begin{cases}1-x^2-y^2, & x^2+y^2\le 1\\ 0, & x^2+y^2>1\end{cases}$$

по прямоугольнику $-1\le x\le 1, -1\le y\le 1.$

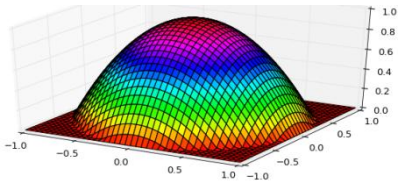
```
from scipy.integrate import dblquad
import numpy as np
def F(y,x):
    if x**2+y**2<1:
        return 1-x**2-y**2
    else:
        return 0
g=lambda x: -1
h=lambda x: 1

I = dblquad(F, -1,1,g,h, epsabs=1e-6, epsrel=1e-6)
print(I)
(1.5707950459207884, 1.2636992237532934e-06)
```

Сравним с точным значением $\pi/2$.

```
print(np.pi/2)
1.5707963267948966
```

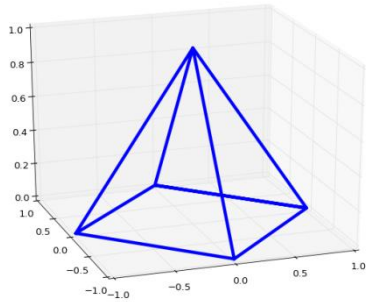
Обратите внимание на то, что абсолютная и относительная погрешность нами задана меньшей, чем значения по умолчанию. Использование значений по умолчанию приводит к сообщению о медленной сходимости интеграла и получению неправильного результата (проверьте). Задание значений

epsabs=1e-7, epsrel=1e-7 сообщения не вызывает, но результат (1.3121922832006239, 1.5164007161061477e-08) все еще неверный.
Ниже показан график «обрезанной» функции $F(x,y)$.



Тем не менее, поверхность интегрирования может содержать ребра. Вот пример вычисления объема правильной четырехугольной пирамиды со стороной a и высотой h $\left(V=\frac{1}{3}ha^2 \right)$. Ее изображение при $h=1$ и $a=\sqrt{2}$ (ломаную, проходящую по ребрам пирамиды) можно построить следующим кодом.

```
fig = plt.figure()
ax=Axes3D(fig)
x=[-1,0,0,-1,0,0,1,0,1,0]
y=[0,0,-1,0,1,0,0,1,0,-1]
z=[0,1,0,0,0,1,0,0,0,0]
ax.plot(x,y,z,linewidth=3)
plt.show()
```



Уравнение поверхности пирамиды имеет вид $z=1-|x|-|y|$, а область интегрирования ограничена снизу и сверху кривыми $g(x)=|x|-1$ и $h(x)=1-|x|$. Тогда ее объем можно найти следующим образом.

```
def f(y,x):
    return 1 -np.abs(x)-np.abs(y)
g=lambda x: np.abs(x)-1
h=lambda x: 1-np.abs(x)
I = dblquad(f, -1,1,g,h)
print('I=',I[0])
I= 0.6666666666666667
```

Трехкратный интеграл $\int_a^b dx \int_{g(x)}^{h(x)} dy \int_{q(x,y)}^{r(x,y)} f(z,y,x)dz$ вычисляется с помощью функции tplquad, которая имеет следующий синтаксис:

```
tplquad(func,a,b,gfun,hfun,qfun,rfun[,args=(),
                        epsabs=1.49e-08, epsrel=1.49e-08])
```

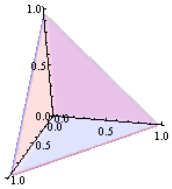
Аргумент func представляет подынтегральную функцию, по крайней мере, трех переменных, задаваемых в порядке z, y, x. Аргументы a,b определяют

пределы интегрирования по x; gfun,hfun задают нижний и верхний пределы интегрирования по y и являются функциями одной переменной x; qfun,rfun - определяют нижний и верхний пределы интегрирования по z и являются функциями двух переменных x,y (в указанном порядке). Смысл опций args, epsabs, epsrel такой же, как и для функции quad.

Вычислим трехкратный интеграл $\int_1^2 dx \int_x^{2x} dy \int_{x-y}^{x+y} (x+y+z)\,dz$.

```
from scipy.integrate import tplquad
def f(z, y, x):  # Обратите внимание на порядок задания аргументов
   return x+y+z
I=tplquad(f, 1, 2,
      lambda x: x,
      lambda x: 2*x,
      lambda x, y: x - y,
      lambda x, y: x + y)
print(I)
(40.0, 4.440892098500626e-13)
```

Вычислим интеграл $I = \iiint_T \frac{dx\,dy\,dz}{(1+x+y+z)^3}$, распространенный на тетраэдр T, (трехмерный симплекс) ограниченный плоскостями $x=0, y=0, z=0$ и $x+y+z=1$. Тетраэдр T показан на следующем рисунке.



Вначале преобразуем интеграл по области в трехкратный. Имеем.

$$I = \iiint_T \frac{dx\,dy\,dz}{(1+x+y+z)^3} = \int_0^1 dx \int_0^{1-x} dy \int_0^{1-x-y} \frac{dz}{(1+x+y+z)^3}.$$

Тогда код вычисления интеграла, стоящего в правой части может быть следующим.

```
def f(z, y, x):
   return 1/(1+x+y+z)**3
I=tplquad(f, 0, 1,
      lambda x: 0,
      lambda x: 1-x,
      lambda x, y: 0,
      lambda x, y: 1-x-y)
print(I)
(0.03407359027997266, 3.782928446046958e-16)
```

Для рассмотренного интеграла известно точное значение $\frac{1}{16}(8\cdot\ln 2 - 5)$. Сравним его с полученным приближенным значением.

```
Is=(8*np.log(2)-5)/16
```

```
print(Is)
0.03407359028
```

Вычислим *объем шара единичного радиуса*. Для этого вычислим объем части шара O, расположенной в первом октанте $(x \ge 0, y \ge 0, z \ge 0)$, а затем умножим результат на 8. При этом удобно использовать сферическую систему координат.

$$V = 8\iiint\limits_{O} dxdydz = 8\iiint\limits_{O} \rho^2 \sin\theta \, d\rho \, d\varphi \, d\theta = 8\int\limits_0^1 d\rho \int\limits_0^{\pi/2} d\varphi \int\limits_0^{\pi/2} \rho^2 \sin\theta \, d\theta$$

Тогда имеем.

```
def dv(theta,phi,ro):
    return ro**2*np.sin(theta)
V=8*tplquad(dv,0,1,
        lambda phi: 0, lambda phi: np.pi/2,
        lambda ro, phi: 0, lambda ro,phi: np.pi/2)[0]
print(V)
4.1887902047863905
```

Сравним с известным значением объема единичного шара $\frac{4}{3}\pi$.

```
print(4/3*np.pi)
4.1887902047863905
```

Для вычисления n – кратных интегралов в модуле `scipy.integrate` имеется функция `nquad`. Она имеет следующий синтаксис:

```
nquad(func, ranges, args=None, opts=None)
```

Здесь `func` имя функции, интеграл от которой вычисляется. Аргумент `ranges` является последовательностью (iterable object), каждый элемент которой должен быть списком (или кортежем) из двух выражений (или двух чисел). Вместо пары выражений можно использовать функцию, которая возвращает список из двух значений. Пара значений в `ranges[0]` задает пределы интегрирования по `x0`, `ranges[1]` – по `x1`, и так далее. Например для четырехкратного интеграла, порядок следования аргументов функций должен быть таким, как написано ниже.

$$\int\limits_{g_3}^{h_3} d\,x_3 \int\limits_{g_2(x_3)}^{h_2(x_3)} d\,x_2 \int\limits_{g_1(x_2,x_3)}^{h_1(x_2,x_3)} d\,x_1 \int\limits_{g_0(x_1,x_2,x_3)}^{h_0(x_1,x_2,x_3)} f\left(x_0,x_1,x_2,x_3\right) d\,x_0$$

Т.о., при создании подынтегральной функции `f` аргументы следует указывать в порядке от `x0` – переменной внутреннего интегрирования, до `xn` – переменной внешнего интегрирования.

```
def f(x0,x1,x2,x3):
    . . .
```

Аргументы функций, задающих пределы интегрирования, тоже нужно указывать в приведенном порядке. Например, порядок аргументов функции `g0` должен быть следующим:

```
def g0(x1,x2,x3):
    . . .
```

Границы интегрирования задаются списком пар `ranges`. Пары состоят из нижнего и верхнего пределов интегрирования, и перечисляются от внутреннего интеграла к внешнему. Каждая пара может быть списком из двух констант, списком из двух функций или функцией, возвращающей пару выражений. Например, вычисление указанного четырехкратного интеграла может быть выполнено следующей инструкцией:

```
nquad(f, [[g0,h0],[g1,h1],[g2,h2],[g3,h3]])
```

Функция `func`, кроме переменных интегрирования, может принимать дополнительные аргументы, которые передаются ей через опцию `args` функции `nquad`.

Вычислим интеграл $\int_0^{\infty}\int_1^{\infty}\frac{e^{-xy}}{y^5}\,dy\,dx$.

```
from scipy.integrate import nquad
import numpy as np
def f(y, x):
   return np.exp(-x*y) / y**5
I=nquad(f, [[1, np.inf],[0, np.inf]])
print(I)
(0.2000000000189363, 1.3682975855986121e-08)
```

Обратите внимание на то, что порядок аргументов функции `f` должен соответствовать порядку, в котором передаются пределы интегрирования: от внутренней переменной к внешней. Внутреннему интегралу по аргументу `y` здесь соответствуют пределы `[1,np.inf]`, а внешнему (по переменной `x`) – соответствуют пределы `[0,np.inf]`. Если хотя бы один из пределов интегрирования непостоянен, то оба элемента пары должны быть функциями.

Вычислим интеграл

$$\iint\limits_{\substack{x\ge 0,\ y\ge 0\\ x+y\le 1}} x^p y^q (1-x-y)^r\,dx\,dy = \int_0^1 dx \int_0^{1-x} x^p y^q (1-x-y)^r\,dy = \frac{p!\,q!\,r!}{(p+q+r+2)!}.$$

Имеем.

```
from scipy.integrate import nquad
import numpy as np
def f(x, y, p, q, r):
   return x**p*y**q*(1-x-y)**r
def ybounds(x):
    return [0, 1-x]
def xbounds():
    return [0, 1]
I=nquad(lambda y,x: f(x,y,1,2,3), [ybounds, xbounds])
print(I)
(0.0002976190476190476, 1.039774842432043e-17)
print(12/np.math.factorial(8))  # точное значение
0.0002976190476190476
```

Обратите внимание на порядок аргументов лямбда – функции. Он соответствует порядку передачи пределов интегрирования – от внутреннего интеграла к внешнему.

Рассмотрим еще один двойной интеграл $\int_0^{1/2}\int_0^{1-2x} x\,y\,d\,y\,d\,x = \frac{1}{96}$ (его значение известно). Имеем.

```
f=lambda y, x:  x*y
xbnd=lambda :  [0, 0.5]
ybnd=lambda x: [0, 1-2*x]]
I=nquad(f, [ybnd, xbnd])
print(I)
(0.010416666666666668, 4.101620128472366e-16)
```

Заметим, что в этом примере инструкцию `nquad` можно использовать в виде `I=nquad(f,[ybnd,[0,0.5]])`, и не создавать функцию `xbnd`.

С помощью функции `nquad` вычислим тройной интеграл.

$$\iiint\limits_{\substack{x\ge 0,\ y\ge 9,\ z\ge 0\\ x^2+y^2+z^2\le 1}} \frac{x\,y\,z}{\sqrt{9x^2+4y^2+z^2}}\,d\,x\,d\,y\,d\,z = \int_0^1 d\,x \int_0^{\sqrt{1-x^2}} d\,y \int_0^{\sqrt{1-x^2-y^2}} \frac{x\,y\,z}{\sqrt{9x^2+4y^2+z^2}}\,d\,z$$

Имеем.

```
def f(z, y, x):
   return x*y*z/np.sqrt(9*x**2+4*y**2+z**2)
def zbnd(y,x):
    return [0, np.sqrt(1-x**2-y**2)]
def ybnd(x):
    return [0,np.sqrt(1-x**2)]
def xbnd():
    return [0,1]
I=nquad(f,[zbnd,ybnd, xbnd])
print(I)
(0.012222222222280263, 9.273154430624787e-09)
```

Известно, что объем n – мерного симплекса $x_1 \ge 0, x_2 \ge 0, \ldots, x_n \ge 0,$ $x_1 + x_2 + \ldots + x_n \le 1$ равен $\frac{1}{n!}$ (изображения тетраэдра, являющегося трехмерным симплексом, показано ранее). Определим численно объем четырехмерного симплекса $x_1 \ge 0, x_2 \ge 0, x_3 \ge 0, x_4 \ge 0,\ x_1 + x_2 + x_3 + x_4 \le 1$. Для этого вычислим следующий интеграл:

$$V = \int_0^1 d\,x_1 \int_0^{1-x_1} d\,x_2 \int_0^{1-x_1-x_2} d\,x_3 \int_0^{1-x_1-x_2-x_3} d\,x_4$$

Имеем.

```
from scipy.integrate import nquad
import numpy as np
def f(x4, x3, x2,x1):
   return 1
```

```
def lim4(x3,x2,x1):
    return [0, 1-x1-x2-x3]
def lim3(x2,x1):
    return [0, 1-x1-x2]
def lim2(x1):
    return [0, 1-x1]
def lim1():
    return [0, 1]
V=nquad(f,[lim4,lim3,lim2,lim1])
print(V)
(0.041666666666666664, 1.1030064411555582e-14)
```

Сравним со значением $1/4!$

```
print(1/np.math.factorial(4))
0.041666666666666664
```

До сих пор мы вычисляли интеграл на фиксированном отрезке [a,b], но часто нам нужно вычислять интеграл $\int_a^x f(t)dt$ с переменным верхним пределом, который с точностью до константы совпадает с первообразной $\int f(x)dx$. В пакете scipy.integrate есть функция cumtrapz, которая вычисляет подобный интеграл. Она имеет следующий синтаксис:

```
scipy.integrate.cumtrapz(y, x=None, dx=1.0, axis=-1,initial=None)
```

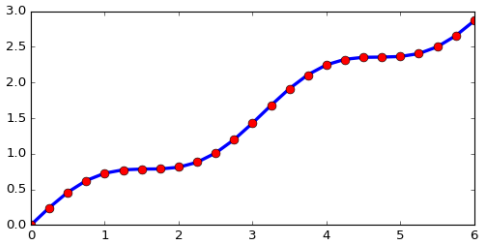
Здесь y – одномерный массив значений интегрируемой функции, x – одномерный массив абсцисс, dx – расстояние между абсциссами (используется только, когда x=None), axis – задает ось интегрирования (по умолчанию используется значение –1, т.е. последняя ось). Параметр initial задает число, используемое в качестве первого значения возвращаемого массива. Обычно оно должно быть равно нулю. По умолчанию ему присваивается значение None, что означает, что значение x[0] не используется, и возвращаемый массив на единицу короче, чем массив значений функции y.

Если массив x не передается (массив y вы должны передавать всегда), то аргумент dx определяет расстояние (по оси x) между элементами массива y. Обратите внимание, что результат не зависит от начальной абсциссы x[0], и она не требуется.

С помощью функции cumtrapz построим график функции $\int_0^x \cos^2 t\,dt$. Значение этого интеграла известно $\frac{x}{2}+\frac{\sin 2x}{4}$, и мы сравним его с численным решением. Имеем.

```
from scipy.integrate import cumtrapz
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 6, num=25)
```

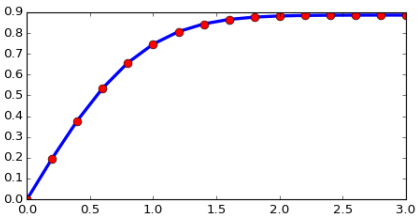
```
y=np.cos(x)**2
yint = cumtrapz(y, x, initial=0)                    # ∫_0^x cos^2 t dt
exact=lambda x: x/2+np.sin(2*x)/4                   # x/2+sin 2x/4
[p1,p2]=plt.plot( x, exact(x), 'b-', x, yint, 'ro' )
p1.set_linewidth(3)
p2.set_markersize(8)
fig=plt.gcf()
fig.set_facecolor('white')
plt.show()
```



■

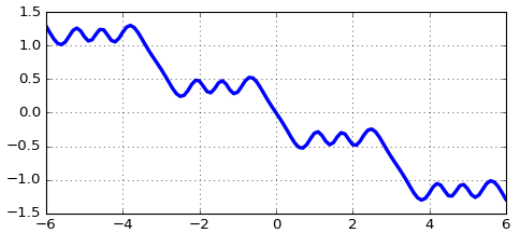
С помощью функции `cumtrapz` построим график функции $\int_0^x e^{-t^2}\,dt$. Этот интеграл не выражается через элементарные функции, и с точностью до множителя совпадает с интегралом ошибок $\operatorname{erf}(x)$. А именно, $\int_0^x e^{-t^2}\,dt = \frac{\sqrt{\pi}}{2}\operatorname{erf}(x)$.

```
from scipy.integrate import cumtrapz
import matplotlib.pyplot as plt
import numpy as np
import scipy.special as sp
x = np.linspace(0, 3, num=16)
y=np.exp(-x**2)
yint = cumtrapz(y, x, initial=0)
exact=lambda x: np.sqrt(np.pi)/2*sp.erf(x)
[p1,p2]=plt.plot( x, exact(x), 'b-', x, yint, 'ro' )
p1.set_linewidth(3)
p2.set_markersize(8)
fig=plt.gcf()
fig.set_facecolor('white')
plt.show()
```



Построим график функции $F(x)=\int_0^x \cos(10\cos t)\,dt$. Для этого интеграла не существует представления ни через элементарные, ни через специальные функции. Тем не менее, график этой функции можно построить. Имеем.

```
from scipy.integrate import cumtrapz
import matplotlib.pyplot as plt
import numpy as np
x1 = np.linspace(0, 6, num=61)
y1=np.cos(10*np.cos(x1))
yint1 = cumtrapz(y1, x1, initial=0)
x2 = np.linspace(0, -6, num=61)
y2=np.cos(10*np.cos(x2))
yint2 = cumtrapz(y2, x2, initial=0)
# объединяем массивы
X=np.append(x2[-1:0:-1],x1)
Y=np.append(yint2[-1:0:-1],yint1)
p1=plt.plot( X, Y, 'b-', linewidth=3 )
fig=plt.gcf()
fig.set_facecolor('white')
fig.gca().grid(True)
plt.show()
```



6.1.2 Вычисление длин, площадей и объемов.

Длина дуги плоской кривой, заданной явным уравнением $y = y(x)$ в декартовых координатах, вычисляется по формуле

$$L = \int_{x_0}^{x_1} \sqrt{1 + (y'(x))^2}\, d\,x, \qquad (1)$$

где x_0 и x_1 определяют начальную и конечную точки дуги.

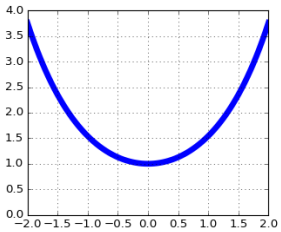
Пример. Длина цепной линии $y = ch\ x$ на отрезке $0 \le x \le 1$.

```
from scipy.integrate import quad
import numpy as np
import sympy as smp
import matplotlib.pyplot as plt
x=smp.symbols('x')
fn=smp.cosh(x)                  # символьное выражение уравнения
f=smp.sqrt(1+fn.diff(x)**2)     # подынтегральное символьное выражение
g=smp.lambdify(x, f, "numpy")   # преобразование в выражение numpy
L=quad(g,0,1)                   # вычисление интеграла
print(L)                        # значение интеграла и точность
(1.1752011936438014, 1.3047354237503154e-14)
```

Сравним с точным значением $\sinh(1)$.

```
print(np.sinh(1))
```

```
1.17520119364
# грфик цепной линии
fig = plt.figure(facecolor='white') #,xlim=(-2,2), ylim=(0,4))
ax = plt.axes(xlim=(-2, 2), ylim=(0, 4))
fun=smp.lambdify(x, fn, "numpy")
x=np.linspace(-2,2,81)
ax.plot(x,fun(x),lw=3)
ax.grid(True)
plt.show()
```



Обратите внимание на инструкцию `g=smp.lambdify(x,f,"numpy")`, которая из символьного выражения `f` создает функцию `g`. Она потребовалась потому, что символьные выражения нельзя использовать в функциях, выполняющих численное интегрирование и, в частности, в функции `quad`.

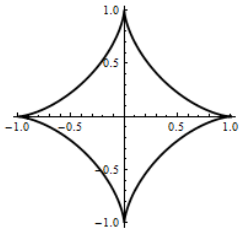
■

Длина дуги плоской кривой, заданной в параметрическом виде $x = x(t),\ y = y(t),$ вычисляется по формуле

$$L = \int_{t_0}^{t_1} \sqrt{(x'(t))^2 + (y'(t))^2}\, dt \qquad (2)$$

где t_0 и t_1 – значения параметра t в начальной и конечной точках дуги.

Пример. Длина астроиды $x = \cos^3 t,\ y = \sin^3 t,\quad 0 \le t \le 2\pi$.



```
from scipy.integrate import quad
import numpy as np
import sympy as smp
t=smp.symbols('t')
xt=smp.cos(t)**3
yt=smp.sin(t)**3
dt=smp.sqrt(xt.diff(t)**2+yt.diff(t)**2)  # подынтегральное выражение
g=smp.lambdify(t, dt, "numpy")
L=quad(g,0,2*np.pi)
print(L)
(6.0, 6.661338147750939e-14)
```

В этом коде также существенной является инструкция

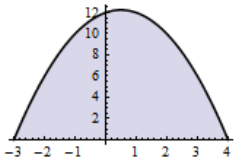
`g=smp.lambdify(t, dt, "numpy")`

Она из символьного выражения `dt` относительно символьной переменной `t` создает функцию `g`, которую можно использовать в численном интегрировании.

■

Геометрический смысл определенного интеграла $\int_a^b f(x)dx$ состоит в том, что он представляет *площадь под кривой* $y = f(x)$ $(a \le x \le b)$.

Пример. Найдем площадь фигуры, ограниченной сверху параболой $y = 9 - x^2$, а снизу осью абсцисс.



Для этого решим уравнение $12 + x - x^2 = 0$ и найдем точки, в которых парабола пересекается с осью абсцисс.

```
import sympy as smp
x=smp.symbols('x')
y=-x**2+x+12
root=smp.solve(y, x)
a=root[0]; b=root[1]; print(a,b)
-3 4
```

Теперь вычисляем площадь.

```
s=smp.integrate(y, (x,a,b))
print('s=',s)
s= 343/6
```

В этом примере интеграл удалось вычислить символьно, однако это бывает нечасто.

■

Площадь поверхности, заданной явным уравнением $z = f(x, y)$. Площадь куска поверхности, который проектируется на плоскость xOy в виде замкнутой области D, вычисляется по формуле

$$S = \iint_D \sqrt{1 + f_x^2 + f_y^2}\, dx\, dy. \qquad (3)$$

Пример. *Площадь поверхности сферы*. Уравнение $z = \sqrt{R^2 - x^2 - y^2}$ представляет полусферу радиуса `R`. Учитывая, что (3) дает площадь только верхней полусферы, мы приходим к следующей формуле для площади сферы:

$$S = 2 \cdot \iint_D \sqrt{1 + z_x^2 + z_y^2}\, dx\, dy = 2 \cdot \int_{-R}^{R} dx \int_{-\sqrt{R^2-x^2-y^2}}^{\sqrt{R^2-x^2-y^2}} \sqrt{1 + z_x^2 + z_y^2}\, dy$$

Имеем.

```
from scipy.integrate import dblquad
import numpy as np
import sympy as smp
x, y=smp.symbols('x y')
R=1
```

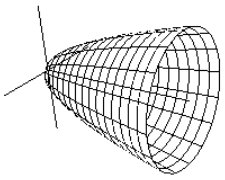
```
z=smp.sqrt(R**2-x**2-y**2)
ds=smp.sqrt(1+z.diff(x)**2+z.diff(y)**2)
DS=smp.lambdify((y,x), ds, "numpy")
g=lambda x: -np.sqrt(R**2-x**2)
h=lambda x: np.sqrt(R**2-x**2)
s=2*dblquad(DS,-R,R,g,h)[0]
print(s)
12.56637061435586
print(4*np.pi*R**2)          # точное значение площади 4πR²
12.566370614359172
```

■

Площадь поверхности, получаемой вращением кривой $y = f(x)\ (a \le x \le b)$ *вокруг оси* O*x*, определяется по формуле

$$S = 2\pi \int_a^b y\sqrt{1+y'^2_x}\,dx = 2\pi \int_a^b f(x)\sqrt{1+(f'(x))^2}\,dx \qquad (4)$$

Пример. Вычислим площадь поверхности, образованной вращением вокруг оси O*x* части кривой $y^2 = 4 + x$, отсеченной прямой *x*=2.



Уравнение $y^2 = 4 + x$ определяет параболу с вершиной в точке (-4, 0) и осью симметрии O*x*. Поэтому для вычисления площади поверхности вращения достаточно рассмотреть одну ветвь параболы $y = \sqrt{4+x}$ на отрезке [-4,2] (точное значение площади равно $62\pi/3$).

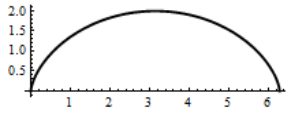
```
from scipy.integrate import quad
import numpy as np
import sympy as smp
x=smp.symbols('x')
y=smp.sqrt(4+x)
ds=y*smp.sqrt(1+y.diff(x)**2)
DS=smp.lambdify(x, ds, "numpy")
s=2*np.pi*quad(DS,-4,2)[0]
print(s)
64.92624817418037
```

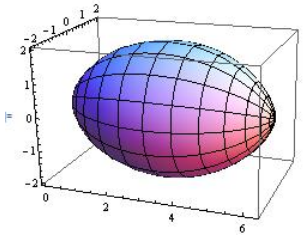
■

Плоская кривая задана параметрическими уравнениями $x = x(t),\ y = y(t)$ $(t_0 \le t \le T)$. *Площадь поверхности ее вращения вокруг оси* x вычисляется по формуле

$$S = 2\pi \int_{t_0}^{T} y(t)\sqrt{(x'(t))^2 + (y'(t))^2}\,dt \qquad (5)$$

Пример. Дана циклоида $x=a(t-\sin t),\quad y=a(1-\cos t)$. Найти площадь поверхности, образованной вращением одной дуги кривой вокруг оси Ox (точное значение площади равно $\frac{64a^2\pi}{3}$).





```
from scipy.integrate import quad
import numpy as np
import sympy as smp
t=smp.symbols('x')
a=1
x=a*(t-smp.sin(t))
y=a*(1-smp.cos(t))
ds=y*smp.sqrt(x.diff(t)**2+y.diff(t)**2)
DS=smp.lambdify(t, ds, "numpy")
s=2*np.pi*quad(DS,0,2*np.pi)[0]
print(s)
67.02064327658223
```

■

Объем тела U в декартовых координатах Oxyz вычисляется с помощью тройного интеграла

$$V=\iiint\limits_U d\,x\,d\,y\,d\,z$$

Пример. *Объем конуса высотой* H *и радиусом основания* R. Конус ограничен поверхностью $z=\frac{H}{R}\left(R-\sqrt{x^2+y^2}\right)$ и плоскостью z=0. В декартовых координатах его объем выражается следующим трехкратным интегралом

$$V=\iiint\limits_U d\,x\,d\,y\,d\,z=\int\limits_{-R}^{R} d\,x\int\limits_{-\sqrt{R^2-x^2}}^{\sqrt{R^2-x^2}} d\,y\int\limits_0^{\frac{H}{R}\left(R-\sqrt{x^2+y^2}\right)} d\,z\ .$$

Вычислим его.

```
from scipy.integrate import nquad
import numpy as np
f=lambda z,y,x: 1
zbnd=lambda y,x: [0, H/R*(R-np.sqrt(x**2+y**2))]
ybnd=lambda x: [-np.sqrt(R**2-x**2),np.sqrt(R**2-x**2)]
xbnd=lambda : [-R,R]
H=1; R=1
I=nquad(f,[zbnd,ybnd, xbnd])
print(I)
(1.047197551137596, 1.4896493976954173e-08)
```

Сравним полученный результат с известным точным значением $V = \frac{\pi R^2 H}{3}$.

```
print(np.pi*R**2*H/3)
1.0471975511965976
```

6.2 Обыкновенные дифференциальные уравнения и системы.

Задача Коши для одного дифференциального уравнения `n` – го порядка состоит в нахождении функции, удовлетворяющей равенству

$$y^{(n)} = f\left(t, y, y', ..., y^{(n-1)}\right)$$

и начальным условиям

$$y(t_0) = y_1^0,\ y'(t_0) = y_2^0, ..., y^{(n-1)}(t_0) = y_n^0$$

Перед решением эта задача (уравнение или система уравнений) должна быть переписана в виде системы ОДУ первого порядка

$$\begin{aligned} \frac{d\,y_1}{d\,t} &= f_1(y_1, y_2, ..., y_n, t) \\ \frac{d\,y_2}{d\,t} &= f_2(y_1, y_2, ..., y_n, t) \\ &\cdots\cdots\cdots\cdots \\ \frac{d\,y_n}{d\,t} &= f_n(y_1, y_2, ..., y_n, t) \end{aligned} \qquad (1)$$

с начальными условиями $y_1(t_0) = y_1^0, y_2(t_0) = y_2^0, ..., y_n(t_0) = y_n^0$.

Модуль `scipy.integrate` имеет две функции `ode()` и `odeint()`, предназначенные для решения систем обыкновенных дифференциальных уравнений (ОДУ) первого порядка с начальными условиями в одной точке (задача Коши). Функция `ode()` более универсальная, а функция `odeint()` (ODE integrator) имеет боле простой интерфейс и хорошо решает большинство задач.

Функция `odeint()`. Функция `odeint()`имеет три обязательных аргумента и много опций. Она имеет следующий формат

```
odeint(func, y0, t[,args=(), ...])
```

Аргумент `func` – это имя Python функции двух переменных, первой из которых является список `y=[y1,y2,...,yn]`, а второй – имя независимой переменной. Функция `func` должна возвращать список из `n` значений функций $f_i(y_1, ..., y_n, t)$ при заданном значении независимого аргумента `t`. Фактически функция `func(y,t)` реализует вычисление правых частей системы (1).

Второй аргумент `y0` функции `odeint()` является массивом (или списком) начальных значений $y_1^0, y_2^0, ..., y_n^0$ при `t=t0`. Третий аргумент является массивом моментов времени, в которые вы хотите получить решение задачи. При этом первый элемент этого массива рассматривается как `t0`.

Функция `odeint()` возвращает массив размера `len(t) x len(y0)`.

Функция `odeint()` имеет много опций, управляющих ее работой. Опции `rtol` (относительная погрешность) и `atol` (абсолютная погрешность) определяют погрешность вычислений e_i для каждого значения y_i по формуле $\| e_i \| \le r_{tol} \cdot | y_i | + a_{tol}$. Они могут быть векторами или скалярами. По умолчанию `rtol=atol=1.49012e-8`.

Договоримся, что в начале всех примеров этого параграфа выполняется следующее импортирование модулей и функций.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
```

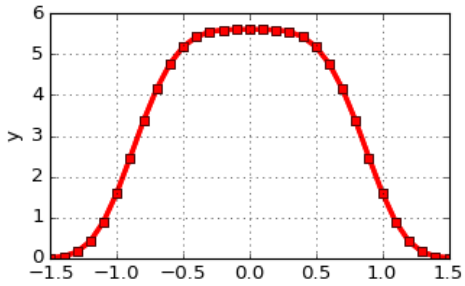
Пример. Рассмотрим задачу Коши $y' = -5t^3 \cdot y, \quad y(-1.5) = 0.01$.

Создадим функцию, реализующую правую часть уравнения.

```
def dydt(y, t):
    return -5*t**3*y
```

Вызываем солвер и строим график.

```
t = np.linspace(-1.5,1.5,31)      # вектор моментов времени
y0 = 0.01                         # начальное значение
y = odeint(dydt, y0, t)           # решение уравнения
y = np.array(y).flatten()         # преобразование массива
plt.plot(t, y,'-sr',linewidth=3)  # построение графика
```



Обратите внимание, что начальный момент времени определяется первым значением в массиве `t`, а не задается отдельно.

Пример. Решим дифференциальное уравнение $y'(x) + y = x$ с начальным условием $y(0) = 1$. Для этой задачи мы знаем «точное» решение $y(x) = x - 1 + 2e^{-x}$, которое сможем сравнить с приближенным.

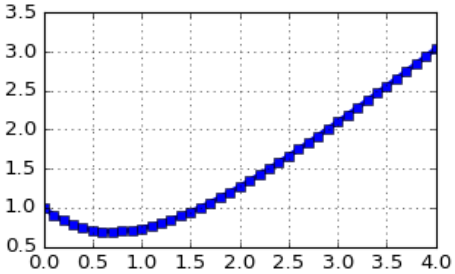
Создаем функцию, реализующую правую часть уравнения.

```
def dydx(y, x):                   # функция правой части y'(x) = x - y
    return x - y
```

Вызываем солвер и строим график.

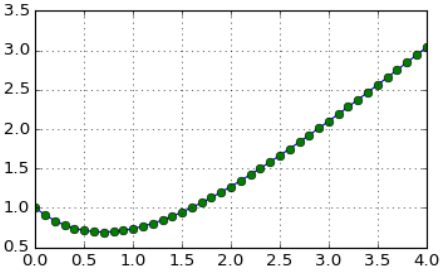
```
x = np.linspace(0,4,41)     # массив значений независимой переменной
y0 = 1.0                          # начальное значение
y = odeint(dydx, y0, x)           # решение уравнения
y = np.array(y).flatten()         # преобразование массива
fig = plt.figure(facecolor='white')  # построение графика
plt.plot(x, y,'-sb',linewidth=3)
```

```
ax = fig.gca()
ax.grid(True)
```



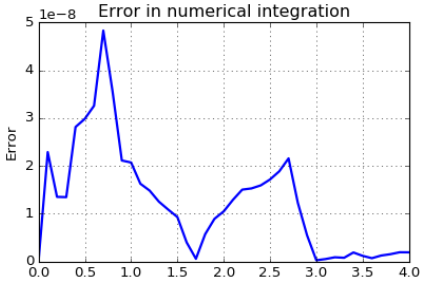
Сравним графики «точного» $y = x - 1 + 2e^{-x}$ и приближенного решений.

```
y_exact = x - 1 + 2*np.exp(-x)
fig = plt.figure(facecolor='white')
ax = fig.gca()
ax.plot(x, y, x, y_exact, 'o');
ax.grid(True)
```



Теперь построим график модуля разности точного и приближенного решений.

```
fig = plt.figure(facecolor='white')
y_diff =np.abs(y_exact - y)
plt.plot(x,y_diff, linewidth=2)
plt.ylabel("Error")
plt.xlabel("x")
plt.title("Error in numerical integration");
plt.grid(True)
```



Как видим, абсолютная погрешность не превосходит $5 \cdot 10^{-8}$.

Пример. Решим задачу Коши $z'' + \frac{1}{5} z' + z = 0,\ z(0) = 0,\ z'(0) = 1.$

Вначале приведем ее к задаче Коши для системы ОДУ 1-го порядка. Обозначим $y_1 = z, y_2 = z'$. Тогда приходим к следующей задаче

$$\frac{d\,y_1}{dt} = y_2,\ \frac{d\,y_2}{d\,t} = -\frac{1}{5} y_2 - y_1,\ y_1(0) = 0,\ y_2(0) = 1$$

Создаем функцию, реализующую вычисления правых частей полученной системы ОДУ. Ее первый аргумент должен быть списком (или массивом) имен

искомых функций, а второй – именем независимой переменной, даже если он не используется при формировании функции.

```
def f(y, t):
    y1, y2 = y      # вводим имена искомых функций
    return [y2,-0.2*y2-y1]
```

Решаем систему ОДУ.

```
t = np.linspace(0,20,41)
y0 = [0, 1]
w = odeint(f, y0, t)
w
array([[ 0.        ,  1.        ],
       [ 0.45623694,  0.79029901],
       [ 0.76275767,  0.41642039],
       [ 0.86239266, -0.01890162],
         . . . . . .
       [ 0.07506831,  0.1135842 ],
       [ 0.11799752,  0.05551654]])
```

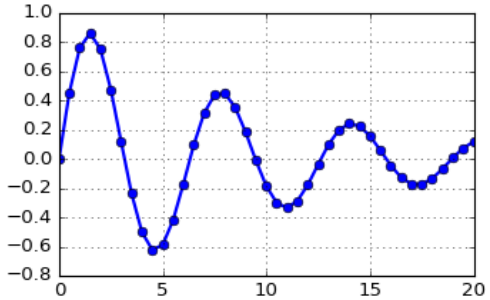
Первый столбец представляет значения функции $y_1(t)$, а второй – значения функции $y_2(t)$ в точках вектора `t`.

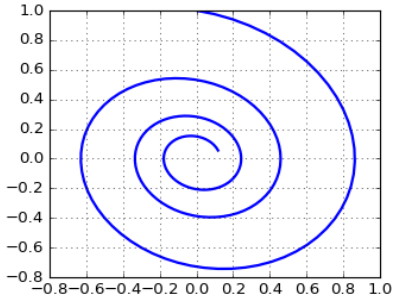
Выделяем вектора `y1` и `y2` из результирующей матрицы.

```
y1=w[:,0]        # вектор значений решения
y2=w[:,1]        # вектор значений производной
```

Строим график решения (значения искомой функции находятся в первом столбце матрицы `w`, т.е. в векторе `y1`)

```
fig = plt.figure(facecolor='white')  # следующий рисунок слева
plt.plot(t,y1, '-o',linewidth=2)
plt.ylabel("z")
plt.xlabel("t")
plt.grid(True)
```





Строим график фазовой траектории. Последовательность решения та же, но количество искомых точек возьмем больше.

```
t = np.linspace(0,20,201)
y0 = [0, 1]
[y1,y2]=odeint(f, y0, t, full_output=False).T
fig = plt.figure(facecolor='white')      # предыдущий рисунок справа
plt.plot(y1,y2,linewidth=2)
plt.grid(True)
```

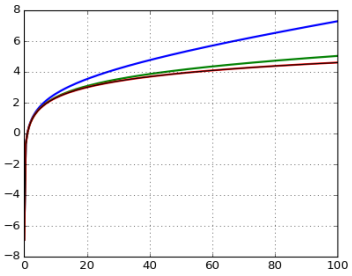
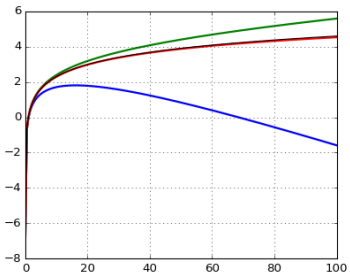
Чтобы сразу получит вектора решений `y1` и `y2` командой `[y1,y2]=odeint(...).T` , мы использовали опцию `full_output=False`. Если `full_output=True`, то функция `odeint()` возвращает дополнительную информацию в форме словаря (см. справочную систему), и тогда операция транспонирования не будет работать, точнее ее нужно применять к первому элементу результата.

Пример. Решим дифференциальное уравнение $y''=-\frac{1}{t^2}$ на отрезке $[a;100]$ с начальными условиями $y(a)=\ln(a)$, $y'(a)=1/a$ при a=0.001. Его точное решение $y=\ln t$.

Приведем задачу к системе ОДУ первого порядка и решим ее с относительной погрешностью `rtol`=$0.5\cdot10^{-4}$, 10^{-6}, 10^{-8} (абсолютная погрешность `atol` задана по умолчанию). Потом решим задачу с относительной погрешностью `rtol`, заданной по умолчанию, и различными значениями абсолютной погрешности `atol`=$10^{-2},10^{-3},10^{-4}$. Построим графики приближенных решений и график точного решения.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def f(y, x):
   y1, y2 = y
   return [y2,-1/x**2]

a=0.001
y0=[log(a),1/a]
x = np.linspace(a,100,200)
z=np.log(x)                          # точное решение
fig = plt.figure(facecolor='white')
plt.hold(True)

reltol=[0.5e-4,1e-6,1e-8]
for tol in reltol:
   [y1,y2]=odeint(f, y0, x, full_output=False, rtol=tol).T
   plt.plot(x,y1,linewidth=2)      # следующий рисунок слева
plt.plot(x,z,'k',linewidth=1)      # график точного решения.
plt.grid(True)

fig = plt.figure(facecolor='white')
plt.hold(True)
abstol=[1e-2,1e-3,1e-4]
for tol in abstol:
   [y1,y2]=odeint(f, y0, x, full_output=False, atol=tol).T
   plt.plot(x,y1,linewidth=2)      # следующий рисунок справа
plt.plot(x,z,'k',linewidth=1)      # график точного решения.
```

```
plt.grid(True)
```

На левом рисунке тонкой черной линией (нижняя кривая) показан график точного решения. Другими цветами показаны графики приближенного решения при различной относительной погрешности. Как видим, относительной погрешности $0.5\cdot10^{-4}$ и 10^{-6} для данной задачи явно недостаточно. Хорошее приближение к точному решению получилось только при `rtol` = 10^{-8} (график этого решения сливается с кривой точного решения). На правом рисунке средняя кривая представляет точное решение. С ней сливается график приближенного решения, соответствующего `atol`=10^{-4}.

Заметим, что названия опций абсолютная и относительная погрешность не полностью отвечают определениям этих терминов. Опции `rtol` и `atol` определяют погрешность вычислений e_i для каждого значения y_i по формуле $\| e_i \| \le r_{tol} \cdot | y_i | + a_{tol}$. Они будут соответствовать своим названиям, если в первой группе решений вы используете опцию `atol=0`, например, измените строку решения следующей

```
[y1,y2]=odeint(f,y0,x,full_output=False,rtol=tol,atol=0).T.
```

Аналогично, во второй группе строку решения можно заменить такой

```
[y1,y2]=odeint(f,y0,x,full_output=False,atol=tol,rtol=0).T
```

По умолчанию обе опции «точности» равны `rtol=atol=1.49012e-8`.

В этом примере мы использовали функцию `hold(True/False)`, которая включает (или выключает) режим добавления новых графиков на текущую графическую область. Если функция `hold()` вызывается без аргумента, то текущий режим переключается на противоположный. Заметим, что у некоторых функций типа `plot()` имеется аналогично действующая опция `hold=True/False`. Обычно режим добавления графиков включен по умолчанию.

Пример. Исследуем решение задачи Коши

$$x'' + x^3 = \sin t,\ x(0) = 0, x'(0) = 0$$

Преобразуем уравнение 2 – го порядка в систему уравнений 1 – го порядка. Сделав замену $x(t) = y_1(t), x'(t) = y_2(t)$, приходим к задаче

$$\frac{d\,y_1}{dt} = y_2,\ \frac{d\,y_2}{d\,t} = -y_l^3(t) - \sin t,\ y_1(0) = 0,\ y_2(0) = 0.$$

Создаем функцию правых частей системы.

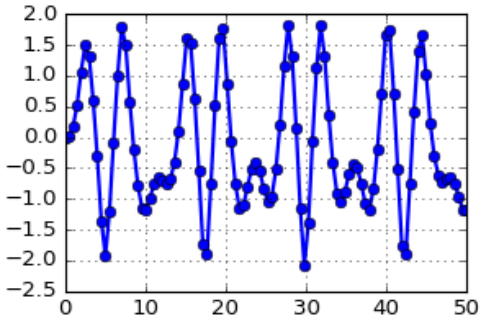
```
def f(y, t):
    y1, y2 = y                  # вводим имена искомых функций
    return [y2,-y1**3+sin(t)]
```

Решаем систему ОДУ.

```
t = np.linspace(0,50,201)
y0 = [0, 0]
[y1,y2]=odeint(f, y0, t, full_output=False).T
```

Строим график решения.

```
fig = plt.figure(facecolor='white')
plt.plot(t,y1, '-o',linewidth=2)     # график решения
plt.grid(True)
```

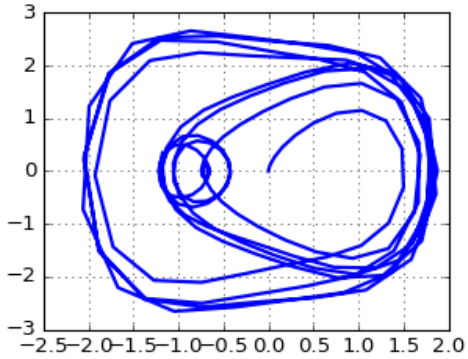


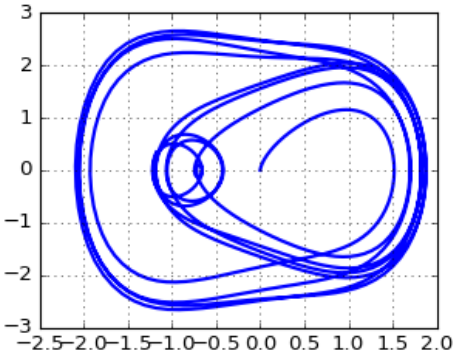
Строим фазовую траекторию.

```
fig = plt.figure(facecolor='white')
plt.plot(y1,y2,linewidth=2)                # следующий рисунок слева
plt.grid(True)
```

Как видите, количества точек, выбранных нами, недостаточно. Вы можете в команде `t = np.linspace(0,50,201)` увеличить это количество, например, до 1000. Можно поступить иначе – выполнить интерполяцию решения и использовать бо̀льшее количество точек только для построения графика фазовой траектории.

```
from scipy.interpolate import interp1d
F1=interp1d(t, y1,kind='cubic')
F2=interp1d(t, y2,kind='cubic')
tnew=np.linspace(0,50,1001)
fig = plt.figure(facecolor='white')
plt.plot(F1(tnew),F2(tnew),linewidth=2) # следующий рисунок справа
plt.grid(True)
```

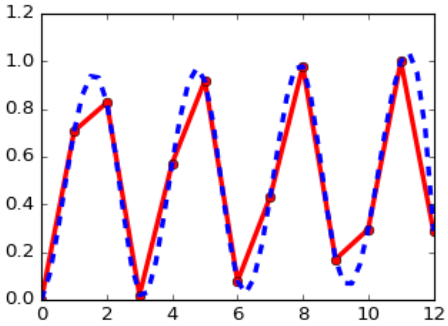




Поясним, как здесь выполнялась интерполяция. Функция `interp1d(xvec,yvec[,kind='linear',...])` принимает одномерные массивы `xvec`, `yvec` данных и возвращает ссылку на интерполяционную функцию. Тип интерполяции задается опцией `kind`, которая может принимать значения `'linear'`, `'nearest'`, `'zero'`, `'slinear'`, `'quadratic`,

'cubic', а также быть целым положительным числом (порядок интерполяционного сплайна). Вот простой пример.

```
from scipy.interpolate import interp1d
x = np.linspace(0, 12, num=13, endpoint=True)
y = np.sin(x)**2                          # массив значений
f = interp1d(x, y,kind='cubic')           # кубическая сплайн интерполяция
xx=np.linspace(0, 12, num=49, endpoint=True) # новый набор точек
fig = plt.figure(facecolor='white')
plt.plot(x, y, '-or', xx, f(xx),'--b',linewidth=3)      # ломаная и сплайн
```



Пример. Исследуем решение задачи Коши для системы уравнений

$$x' = y(t),\ y' = -0.01y(t) - \sin x(t),\ x(0) = 0,\ y(0) = 2.1$$

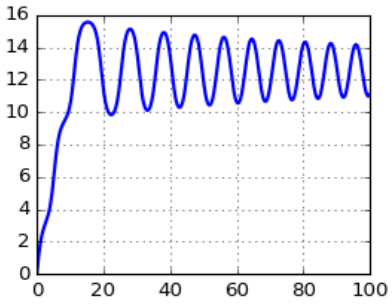
Весь код решения соберем в одну функцию `fun()` с подфункцией `f(y,t)`, вычисляющей правую часть системы уравнений.

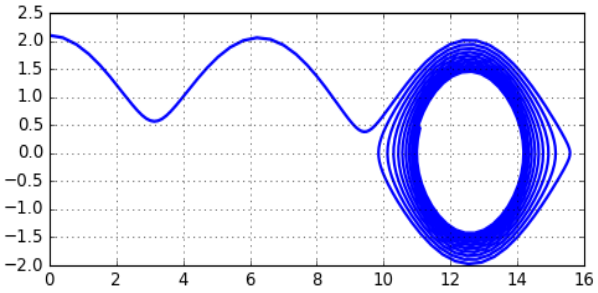
```
def fun():
    def f(y, t):
        y1, y2 = y
        return [y2,-0.01*y2-sin(y1)]

    t = np.linspace(0,100,501)
    y0 = [0, 2.1]
    [y1,y2]=odeint(f, y0, t, full_output=False).T

    fig = plt.figure(facecolor='white')
    plt.plot(t,y1, linewidth=2)           # график решения x(t) слева
    plt.grid(True)

    fig = plt.figure(facecolor='white')
    plt.plot(y1,y2,linewidth=2)           # фазовая траектория справа
    plt.grid(True)
```

Для решения задачи выполните следующие команды

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
fun()                                     # вызов основной функции
```

На левом рисунке показан график решения `x(t)`, а на правом – фазовая траектория.





■

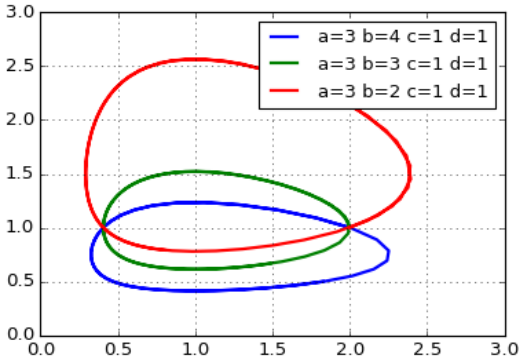
Пример. Исследуем двухвидовую модель «хищник – жертва», впервые построенную Вольтерра для объяснения колебаний рыбных уловов. Имеются два биологических вида, численностью в момент времени `t` соответственно `x(t)` и `y(t)`. Особи первого вида являются пищей для особей второго вида (хищников). Численности популяций в начальный момент времени известны. Требуется определить численность видов в произвольный момент времени. Математической моделью задачи является система дифференциальных уравнений Лотки – Вольтерра

$$\begin{cases} \dfrac{dx}{dt} = (a - b\,y)\cdot x \\ \dfrac{dy}{dt} = (-c + d\,x)\cdot y \end{cases}$$

где `a, b, c, d` – положительные константы. Проведем расчет численности популяций при $a = 3,\ c = 1, d = 1$ для трех значений параметра $b = 4, 3, 2$. Начальные значения положим $x(0) = 2,\ y(0) = 1$.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
def f(y, t,params):
    y1, y2 = y
    a,b,c,d=params
    return [y1*(a-b*y2),y2*(-c+d*y1)]

t = np.linspace(0,7,71)
y0 = [2, 1]
fig = plt.figure(facecolor='white')
plt.hold(True)

for b in range(4,1,-1):
    params=[3,b,1,1]
    st='a=%d b=%d c=%d d=%d' % tuple(params)
    [y1,y2]=odeint(f, y0, t,args=(params,), full_output=False).T
    plt.plot(y1,y2,linewidth=2, label=st)

plt.legend(fontsize=12)
plt.grid(True)
plt.xlim(0,3)
```

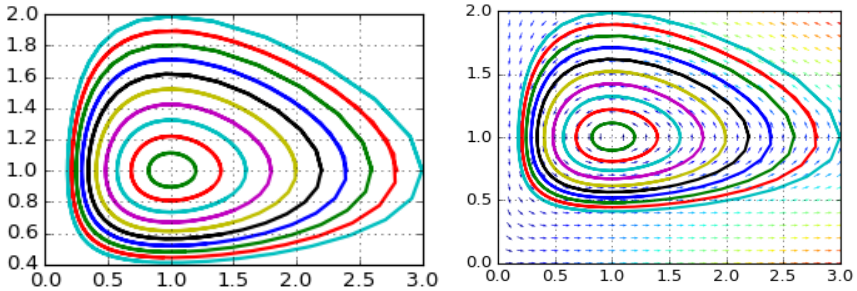
```
plt.ylim(0,3)
```



Из графиков фазовых траекторий видно, что численность популяций меняется периодически.

Теперь будем менять начальные условия, оставляя параметры `a=3,b=3,c=1,d=1` неизменными.

```
def f(y, t,params):
    y1, y2 = y
    a,b,c,d=params
    return [y1*(a-b*y2),y2*(-c+d*y1)]

t = np.linspace(0,7,71)
ic = np.linspace(1.0, 3.0, 11)     # начальные значения для 1-й функции
fig = plt.figure(facecolor='white')
for r in ic:
    y0 = [r, 1.0]
    y = odeint(f, y0, t,args=([3,3,1,1],))
    plt.plot(y[:,0],y[:,1],linewidth=2)
plt.grid(True)                     # следующий рисунок слева
```



Добавьте к предыдущему коду следующие инструкции.

```
x1 = np.linspace(0, 3, 31)
y1 = np.linspace(0, 2, 21)
X1,Y1  = np.meshgrid(x1, y1)    # создание сетки
DX1,DY1 = f([X1, Y1],t,[3,3,1,1])
LN=np.sqrt(DX1**2+DY1**2)
LN[LN == 0] = 1.                # исключение деления на 0
DX1 /= LN                       # нормировка
DY1 /= LN
plt.quiver(X1,Y1, DX1,DY1, LN, pivot='mid',cmap=plt.cm.jet)
```

Результатом работы примера будет график, показанный на предыдущем рисунке справа.

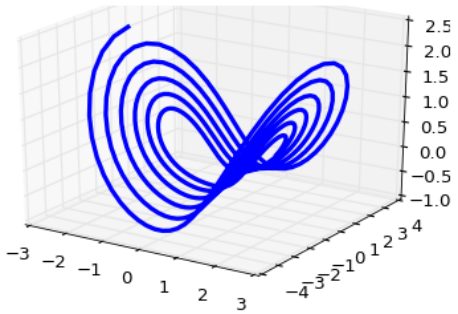
Пример. Решим систему дифференциальных уравнений

$$\frac{dx}{dt} = y, \frac{dy}{dt} = 0.1 \cdot y - x \cdot (z-1) - x^3, \frac{dz}{dt} = x \cdot y - 0.1 \cdot z$$

с начальными условиями $x(0)=1, y(0)=1, z(0)=0$ и построим ее фазовый портрет. Отличие от предыдущих задач состоит в том, что фазовая кривая расположена в трехмерном пространстве.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def f(y, t):
    y1, y2, y3 = y
    return [y2,0.1*y2-y1*(y3-1)-y1**3,y1*y2-0.1*y3]
y0=[1,1,0]
t = np.linspace(0,25,501)
fig = plt.figure(facecolor='white')
ax=Axes3D(fig)
[y1,y2,y3]=odeint(f, y0, t, full_output=False).T
ax.plot(y1,y2,y3,linewidth=3)
```

Фазовая траектория показана на следующем рисунке.



Пример. Исследуем поведения математического маятника. Пусть масса груза равна единице, а стержень, на котором подвешена масса, невесом. Тогда дифференциальное уравнение движения груза имеет вид

$$\varphi'' + k\varphi' + \omega^2 \sin\varphi = 0$$

где $\varphi(t)$ угол отклонения маятника от положения равновесия (нижнее положение), параметр k характеризует величину трения, $\omega^2 = g/l$ (g ускорение свободного падения, l – длина маятника). Для определения конкретного движения к уравнению надо добавить начальные условия $\varphi(0)=\varphi_0$, $\varphi'(0)=\varphi_0'$.

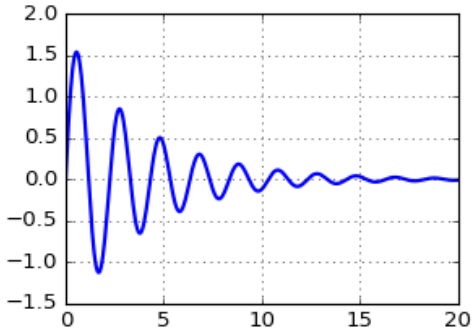
Преобразуем уравнение к системе ОДУ 1 – го порядка. Если обозначить $u \equiv \varphi$, $v \equiv \varphi'$, то получим следующую задачу

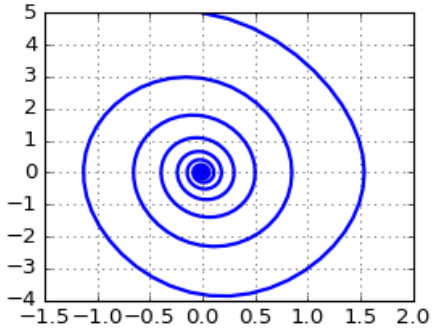
$$\begin{cases} u' = v \\ v' = -kv - \omega^2 \sin(u) \end{cases}, \quad u(0)=\varphi_0,\ v(0)=\varphi_0'.$$

Пусть $k=0.5$, $\omega^2 = 10$, и зададим следующие начальные значения $\varphi_0 = 0, \varphi_0' = v_0 = 5$.

```
import numpy as np
```

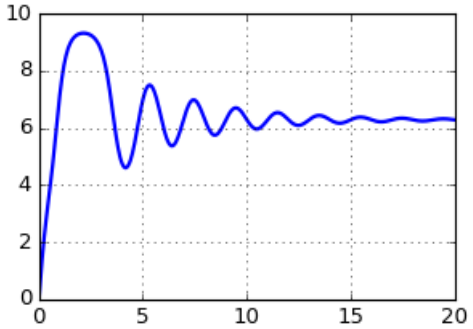
```
from scipy.integrate import odeint
import matplotlib.pyplot as plt
```
Создаем функцию
```
pend=lambda y,t: [y[1],-0.5*y[1]-10*np.sin(y[0])]
```
Решаем систему и строим график (следующий рисунок слева)
```
v0=5                # начальная скорость  φ'0 = v0
t = np.linspace(0,20,401)
[y1,y2]=odeint(pend, [0,v0], t, full_output=False).T
fig = plt.figure(facecolor='white')
plt.plot(t,y1,linewidth=2)
plt.grid(True)
```
Строим фазовую траекторию (рисунок справа)
```
fig = plt.figure(facecolor='white')
plt.plot(y1,y2,linewidth=2)
plt.grid(True)
```

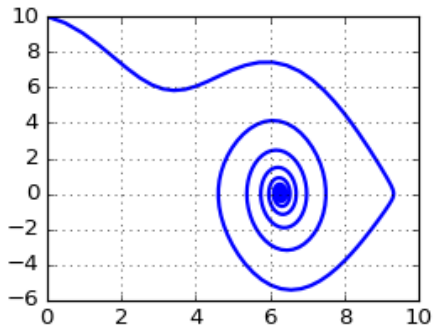




Как видно из левого графика максимальный угол отклонения маятника не превышают $\pi/2$ и колебания маятника затухают.

Увеличим начальную скорость до 10. Для этого в предыдущем коде заменим инструкцию `v0=5` на `v0=10`. Решаем задачу и строим график решения (следующий рисунок слева) и фазовую траекторию (следующий рисунок справа).

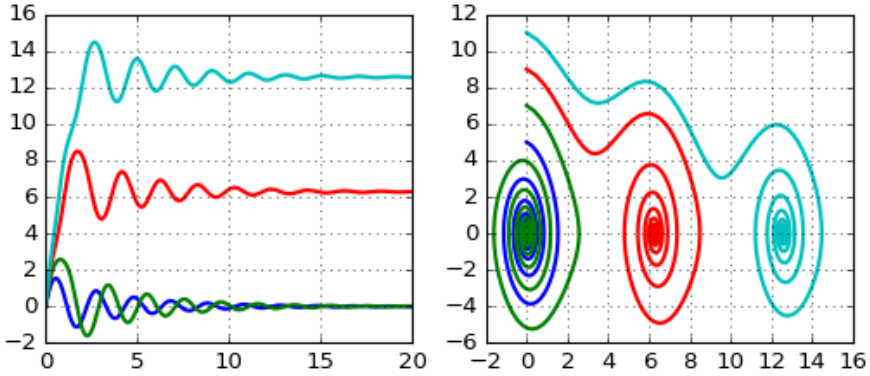




Максимальное значение угла составляет примерно 9 радиан. Маятник сделал один полный оборот вокруг точки закрепления (угол отклонения увеличился на 2π), а затем колебания затухают в окрестности значения 2π радиан (для маятника угол поворота 2π представляет то же, что и 0 радиан, т.е. положение равновесия).

Построим несколько графиков угла отклонения (следующий рисунок слева) и фазовых траекторий (следующий рисунок справа), задавая различную начальную скорость `v0=5,7,9,11`.

```
fig = plt.figure(facecolor='white')
ax1 = fig.add_subplot(1,2,1)
ax1.grid(True)
ax1.hold(True)
ax2 = fig.add_subplot(1,2,2)
ax2.grid(True)
ax2.hold(True)

t = np.linspace(0,20,401)
for v0 in range(5,12,2):
    [y1,y2]=odeint(pend, [0,v0], t, full_output=False).T
    ax1.plot(t,y1,linewidth=2)
    ax2.plot(y1,y2,linewidth=2)
```



Как видим, начальная скорость `v0=5` и `7` недостаточна, чтобы маятник прошел верхнюю точку и сделал хотя бы один полный оборот. При начальной скорости `v0=9` маятник совершает один полный оборот, а затем его колебания затухаю. При `v0=11` маятник выполнил два полных оборота и только после этого его колебания стали затухать вокруг положения равновесия.

Пример. Решим задачу движения планеты вокруг Солнца под действием тяготения. Она записывается в виде системы ОДУ 2-го порядка

$$\ddot{x} = -\frac{k\,x}{\left(x^2+y^2\right)^{3/2}}, \quad \ddot{y} = -\frac{k\,y}{\left(x^2+y^2\right)^{3/2}},$$

где $x(t), y(t)$ - координаты движущейся планеты. Преобразуем исходную систему к системе ОДУ 1-го порядка. Для этого введем обозначения $z_1 = x, z_2 = \dot{x}, z_3 = y, z_4 = \dot{y}$. Тогда

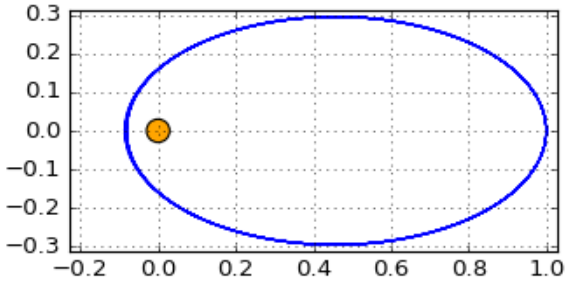
$$\begin{cases} z_1' = z_2 \\ z_2' = -\dfrac{k\,z_1}{\left(z_1^2+z_3^2\right)^{3/2}} \\ z_3' = z_4 \\ z_4' = -\dfrac{k\,z_3}{\left(z_1^2+z_3^2\right)^{3/2}} \end{cases}$$

Для модельной задачи выберем `k=1` и зададим начальные условия

$$z_1(0)=1,\ z_2(0)=0,\ z_3(0)=0,\ z_4(0)=1.$$

Код решения ОДУ и построения графика траектории планеты приведен ниже.

```
import numpy as np
from scipy.integrate import odeint, ode
import matplotlib.pyplot as plt
from matplotlib.patches import Circle

def f(y, t):
    y1, y2, y3, y4 = y
    return [y2,
            -y1/(y1**2+y3**2)**(3/2),
             y4,
            -y3/(y1**2+y3**2)**(3/2)]

t = np.linspace(0,20,1001)
y0 = [1, 0, 0, 0.4]
[y1,y2, y3, y4]=odeint(f, y0, t, full_output=False).T

fig, ax = plt.subplots()
fig.set_facecolor('white')
ax.plot(y1,y3,linewidth=1)
circle = Circle((0, 0), 0.03, facecolor='orange')    # круг
ax.add_patch(circle)
plt.axis('equal')
plt.grid(True)
```

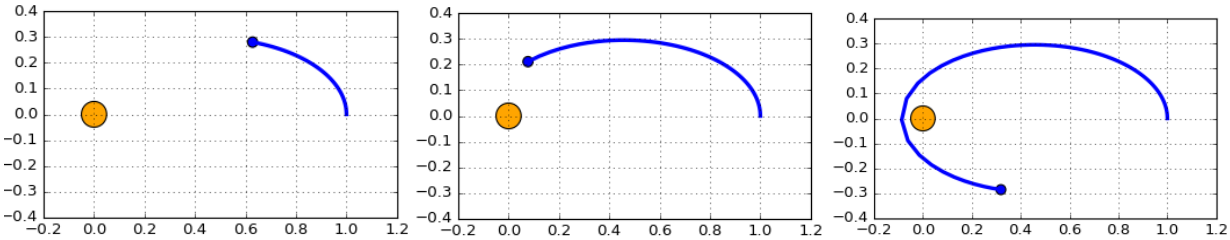


Теперь добавим код создания анимации движения планеты.

```
import matplotlib.animation as animation
fig2 = plt.figure(facecolor='white')
ax = plt.axes(xlim=(-0.2, 1.2), ylim=(-0.4, 0.4) )
line, = ax.plot([ ], [ ], lw=3)
circle = Circle((0, 0), 0.05, facecolor='orange')
ax.add_patch(circle)

pc = plt.Circle((y0[0], y0[2]), 0.02, fc='b')
ax.add_patch(pc)
ax.grid(True)

def redraw(i):
    x = y1[0:i+1]
    y = y3[0:i+1]
    line.set_data(x, y)
    pc.center=(x[-1],y[-1])
```

```
anim =animation.FuncAnimation(fig2,redraw,frames=126,interval=50)
plt.show()
```

При запуске программы открывается графическое окно со статическим рисунком, показанным выше. Закройте его мышкой. После этого откроется второе окно с анимацией. Несколько кадров приведено на следующем рисунке.



Анимация создается с помощью функции `FuncAnimation` в графическом окне `fig2` путем многократно вызыва функции `redraw`. Значение опции `frames=126` подбиралось экспериментально.

Пример. Решим систему уравнений Лоренца

$$\begin{cases} y_1' = s \cdot (y_2 - y_1) \\ y_2' = y_1 \cdot (r - y_3) - y_2 \\ y_3' = y_1 \cdot y_2 - b \cdot y_3 \end{cases},$$

где `s, r, b` некоторые параметры. Положим $s = 10, r = 25, b = 3$ и выберем следующие начальные условия:

$$x(0) = 1, \quad y(0) = -1, z(0) = 10.$$

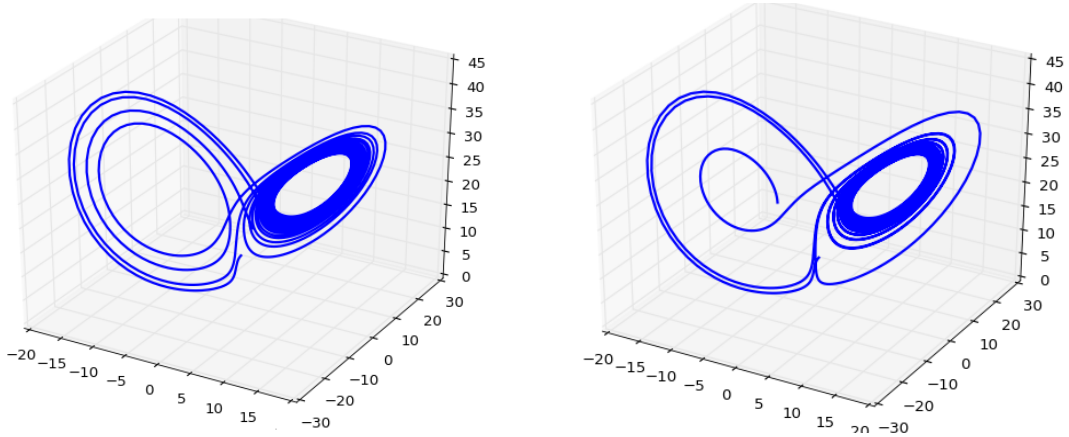
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Создаем функцию правой части системы уравнений.

```
s,r,b=10,25,3
def f(y, t):
    y1, y2, y3 = y
    return [s*(y2-y1),
            -y2+(r-y3)*y1,
            -b*y3+y1*y2]
```

Решаем систему ОДУ и строим ее фазовую траекторию, которая показана на следующем рисунке слева.

```
t = np.linspace(0,20,2001)
y0 = [1, -1, 10]
[y1,y2,y3]=odeint(f, y0, t, full_output=False).T
fig = plt.figure(facecolor='white')  # следующий  рисунок слева
ax=Axes3D(fig)
ax.plot(y1,y2,y3,linewidth=2)
```



Система описывает хаотический аттрактор, поэтому любые малые изменения в начальных условиях будут приводить к существенному изменению решения. Вы это сможете заметить по меняющемуся количеству петель фазовой траектории. Например, скорректируйте начальное значение первой функции на `0.0001`, т.е. вместо инструкции `y0=[1, -1, 10]` введите следующую строку `y0=[1.0001,-1,10]`. Выполните код и получите график фазовой траектории, показанный на предыдущем рисунке справа.

Функция `ode()`. Вторая функция модуля `scipy.integrate`, которая предназначена для решения дифференциальных уравнений и систем, называется `ode()`. Она создает объект ОДУ (тип `scipy.integrate._ode.ode`). Имея ссылку на такой объект, для решения дифференциальных уравнений следует использовать его методы. Аналогично функции `odeint()`, функция `ode(func)` предполагает приведение задачи к системе дифференциальных уравнений вида (1) и использовании ее функции правых частей. Отличие только в том, что функция правых частей `func(t,y)` первым аргументом принимает независимую переменную, а вторым – список значений искомых функций. Например, следующая последовательность инструкций создает объект ODE, представляющий задачу Коши.

```
from scipy.integrate import ode
f = lambda t, y: 2*t      # dy/dt=2*t
ODE=ode(f)                # ссылка на объект ОДУ
```

Используя методы объекта ODE, можно выбрать численный метод решения.

```
ODE.set_integrator('vode')
```

Здесь указано название метода `'vode'`. Он реализует неявный метод Адамса (для нежестких задач), а для жестких задач – метод, основанный на формулах обратного дифференцирования (backward differentiation formulas, BDF). Список доступных методов можно узнать на странице справки функции `ode()`.

С помощью метода `set_initial_value()` задают начальные значения. Формат его вызова следующий: `ODE.set_initial_value(y,t=0.0)`. Например,

```
OD.set_initial_value(1, 0)  # y(0)=1
```

Здесь сказано, что начальное значение t0 независимой переменной равно нулю (t0=0) , а начальное значение искомой функции y0=1. Если решается система, то в качестве начальных значений передается список.

Вычисления выполняются методом integrate(tnew). Например,

```
rez=ODE.integrate(0.1)
```

В данном случае возвращаемое значение rez представляет одноэлементный массив, содержащий значение искомой функции в момент t=tnew. При этом используются заданные ранее начальные условия. Результат вычисления в момент t=tnew также находится в атрибуте ODE.y.

Соберем описанные инструкции в единый пример, в котором решается задача $\frac{dy}{dt} = 2t$, $y(0) = 1$ (задача имеет решение $y = t^2 + 1$), и выполним код.

```
from scipy.integrate import ode
f = lambda t, y: 2*t                # dy/dt=2*t (функция правой части)
ODE=ode(f)                          # ссылка на объект ОДУ
ODE.set_integrator('vode')
ODE.set_initial_value(1, 0)         # y(0)=1
ODE.integrate(0.1)                  # вычисление решения в момент 0.1
print(ODE.y)
[ 1.0100005]
```

Вы можете задавать новые значения независимой переменной и получать значения искомой функции.

```
ODE.integrate(0.2)
print(OD.y)
[ 1.0400005]
print(ODE.integrate(0.3))
[ 1.0900005]
```

Атрибут ODE.t хранит момент времени, для которого вычислено значение решения.

```
ODE.t
0.5                 # точка последнего вычисления
```

Можно построить таблицу решений. Для примера рассмотрим следующую систему 3–х дифференциальных уравнений

$$\frac{dx}{dt} = y - z, \frac{dy}{dt} = z - x, \frac{dz}{dt} = x - 2 \cdot y,$$

с начальными условиями $x(0) = 1$, $y(0) = 0$, $z(0) = 2$.

```
f=lambda t,y: [y[1]-y[2],y[2]-y[0],y[0]-2*y[1]]
OD=ode(f)
y0,t0=[1,0,2], 0
r=OD.set_integrator('vode')
r=OD.set_initial_value(y0, t0)
t1=1.0
dt=0.1
```

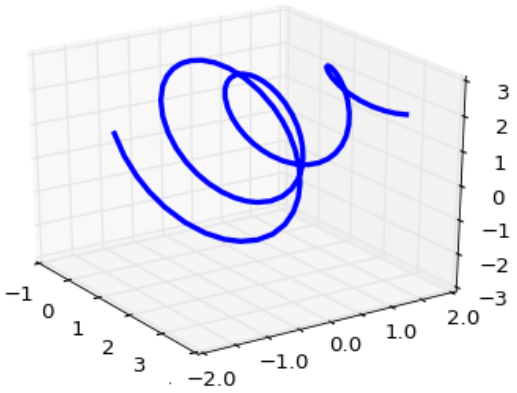
```
while OD.successful() and OD.t <=t1:
   print('%.2f \t%.3f \t%.3f \t%.3f' % (OD.t, OD.y[0],OD.y[1],OD.y[2]))
   OD.integrate(OD.t+dt)
0.00    1.000   0.000   2.000
0.10    0.801   0.114   2.079
0.20    0.609   0.254   2.113
0.30    0.432   0.413   2.099
0.40    0.274   0.585   2.034
0.50    0.144   0.762   1.920
0.60    0.044   0.938   1.759
0.70    -0.020  1.103   1.556
0.80    -0.046  1.250   1.316
0.90    -0.033  1.373   1.049
1.00    0.019   1.465   0.764
```

Здесь в условии цикла `while` мы использовали метод `ODE.successful()`, который возвращает `True`, если шаг интегрирования ДУ завершился успешно.

Многократно решать задачу Коши для того, чтобы получить таблицу значений решения, неэффективно. У объекта ODE имеется метод `ODE.set_solout(fout)`, который загружает в объект специальную функцию `fout()` (обработчик шага). Она вызывается после каждого шага интегрирования. В теле этой функции можно запоминать значение текущего момента времени и соответствующие ему значения решения. Если функция вернет `-1`, то процесс интегрирования в методе `integrate()` завершится на текущем шаге. Чтобы процесс интегрирования не завершился, функция `fout()` ничего не должна возвращать (None) или возвращать 0.

Построим график решения последней задачи.

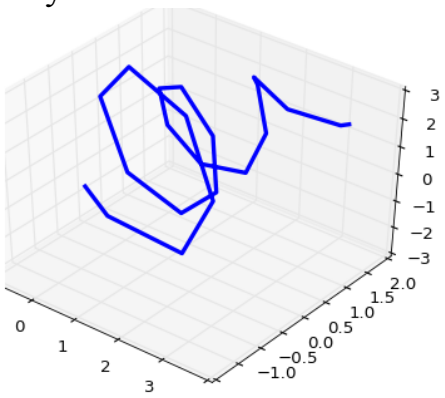
```
ts = [ ]
ys = [ ]
def fout(t, y):          # функция «обработчик шага» ничего не возвращает
   ts.append(t)                    # запоминание t
   ys.append(list(y.copy()))       # запоминание y

f=lambda t,y: [y[1]-y[2],y[2]-y[0],y[0]-2*y[1]]      # функция правой части
OD=ode(f)                          # создание объекта ОДУ
y0, t0=[3.5,1.5, 2], -4
r=OD.set_integrator('dopri5')      # метод Рунге – Кутты 4(5) порядка
                                   # с контролем шага
r.set_solout(fout)                 # загрузка обработчика шага
r=OD.set_initial_value(y0, t0)     # задание начальных значений
ret = r.integrate(6.0)             # решение ОДУ до момента t=6

Y=np.array(ys)
fig = plt.figure(facecolor='white')      # график фазовой траектории
ax=Axes3D(fig)
ax.plot(Y[:,0],Y[:,1],Y[:,2],linewidth=3)
```



В нашем примере функция обработчика шага `fout()` добавляет в глобальные списки `ts` и `ys` значение независимого аргумента `t` и список трех чисел – значений решения при этом `t`. Затем двойной список `ys` мы преобразуем в двумерный массив `Y`, компоненты которого представляют вектора значений решения задачи.

Замечание. Не каждый метод позволяет использование функции обработчика шага. Сейчас эту возможность поддерживают методы Рунге – Кутты `'dopri5'` и `'dop853'`.

Каждый из методов имеет свои собственные опции настройки. Например, измените в предыдущем коде в инструкции `r=OD.set_integrator('dopri5')` имя метода на `'dop853'` и выполните пример. Вы получите грубый график фазовой траектории, показанный на следующем рисунке. Чтобы построить гладкую кривую следует использовать опцию `max_step`, которая задает максимальное значение шага, используемое солвером. Замените инструкцию определения метода следующей командой `r=OD.set_integrator('dop853', max_step=0.1)`, и снова выполните пример. В результате вы получите гладкую фазовую кривую, такую же, какая показана на предыдущем рисунке.



■

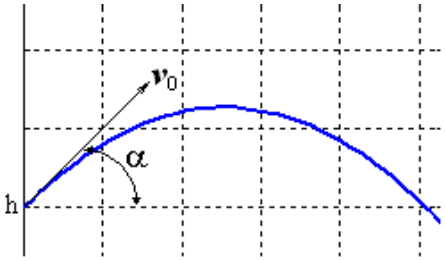
У функции `ode(f[,jac=None])` имеется опция явного задания якобиана $J_{ij} = \frac{\partial f_i}{\partial y_j}$ правой части системы (1). Обычно функция вызывается в формате `ode(f)`, а якобиан вычисляется численно. Вы можете уменьшить погрешность вычислений, если явно создадите эту функцию.

У методов функции `ode()`, кроме опции `max_step` , имеются другие опции, например, `first_step`, `min_step`, `nsteps` (максимальное количество внутренних шагов) и много других опций. Познакомьтесь с ними самостоятельно по справочной системе.

Пример. Решим задачу Коши, описывающую движение тела, брошенного с начальной скоростью v_0 под углом α к горизонту в предположении, что сопротивление воздуха пропорционально квадрату скорости. В векторной форме уравнение движения имеет вид

$$m\ddot{\mathbf{r}} = -\gamma \cdot \mathbf{v} \left| \mathbf{v} \right| - m\mathbf{g},$$

где $\mathbf{r}(t)$ радиус – вектор движущегося тела, $\mathbf{v} = \dot{\mathbf{r}}(t)$ – вектор скорости тела, γ – коэффициент сопротивления, $m\mathbf{g}$ вектор силы веса тела массы m, g – ускорение свободного падения.



Особенность этой задачи состоит в том, что движение заканчивается в заранее неизвестный момент времени, когда тело падает на землю.

Если обозначить $k = \gamma / m$, то в координатной форме мы имеем систему уравнений

$$\ddot{x} = -k\,\dot{x}\sqrt{\dot{x}^2 + \dot{y}^2}$$

$$\ddot{y} = -k\,\dot{y}\sqrt{\dot{x}^2 + \dot{y}^2} - g$$

к которой следует добавить начальные условия: $x(0) = 0$, $y(0) = h$ (h начальная высота), $\dot{x}(0) = v_0 \cos\alpha$, $\dot{y}(0) = v_0 \sin\alpha$.

Положим $y_1 = x, y_2 = \dot{x}, y_3 = y, y_4 = \dot{y}$. Тогда соответствующая система ОДУ 1 – го порядка примет вид

$$\begin{cases} y_1' = y_2 \\ y_2' = -k\,y_2\sqrt{y_2^{\,2} + y_4^{\,2}} \\ y_3' = y_4 \\ y_4' = -k\,y_4\sqrt{y_2^{\,2} + y_4^{\,2}} - g \end{cases}$$

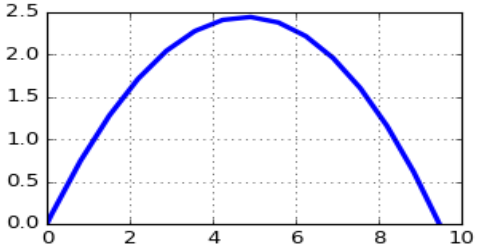
Для модельной задачи положим `h=0, k=0.01, g=9.81, v0=10, α=π/4`.
Создаем функции правой части системы и функцию «обработчик шага»

```
ts = [ ]
ys = [ ]
def fout(t, y):          # обработчик шага
    ts.append(t)
    ys.append(list(y.copy()))
```

```
def f(t, y):                # функция правой части системы ОДУ
    k=0.01
    g=9.81
    y1, y2, y3, y4 = y
    return [y2,
            -k*y2*sqrt(y2**2+y4**2),
            y4,
            -k*y4*sqrt(y2**2+y4**2)-g]
```

Решаем ОДУ и строим его график.

```
tmax=1.41               # время движения, подбирается экспериментально
alph=np.pi/4            # угол бросания тела
v0=10.0                 # начальная скорость

ODE=ode(f)
y0,t0=[0, v0*np.cos(alph), 0, v0*np.sin(alph)], 0  # начальные условия
r=ODE.set_integrator('dopri5', max_step=0.1)     # метод Рунге – Кутта
r.set_solout(fout)                  # загрузка обработчика шага
r=ODE.set_initial_value(y0, t0) # задание начальных значений
ret = r.integrate(tmax)             # решаем ОДУ
Y=np.array(ys)
fig, ax = plt.subplots()
fig.set_facecolor('white')
ax.plot(Y[:,0],Y[:,2],linewidth=3)        # график решения
ax.grid(True)
```



Длительность полета `tmax` нами подбиралась «на глазок» из условия, что в конце полета вертикальная координата равна нулю. Также приходится «на глазок» определять максимальную высоту и дальность полета. Конечно, эти значения следует вычислять. Это можно реализовать в теле функции «обработчика шага» `fout()`. В ней нужно проверять, что вертикальная координата больше нуля, а иначе останавливать вычисления и запоминать значение момента времени. Также в обработчике можно запоминать дальность и высоту полета.

В нашем примере функция `fout()`должна реагировать на обращение в ноль Y координаты тела (компоненты решения `y3`) при ее убывании (т.е. при $y' = y_4 < 0$), а также на обращения в ноль производной, соответствующей наивысшей точке траектории, т.е. на обращение компоненты решения `y4` в ноль. Из физических соображений ясно, что производная обращается в ноль только в одной точке – в самой верхней точке траектории тела. Поэтому, неважно убывает или возрастает производная в этой точке.

Перепишем начальный код примера и функцию обработчика шага.

```
ts = [ ]
ys = [ ]
FlightTime, Distance, Height =0,0,0
y4old=0
def fout(t, y):
    global FlightTime, Distance, Height,y4old
    ts.append(t)
    ys.append(list(y.copy()))
    y1, y2, y3, y4 = y
    if y4*y4old<=0:          # достигнута точка максимума
        Height=y3
    if y4<0 and y3<=0.0:     # тело достигло поверхности
        FlightTime=t
        Distance=y1
        return -1
    y4old=y4
```

Здесь мы создали 3 глобальные переменные (`FlightTime`, `Distance`, `Height`), предназначенные для хранения полетного времени, дальности и максимальной высоты полета. Также создали вспомогательную глобальную переменную `y4old`, которая содержит значение производной $y'(t)$ на предыдущем шаге вычислений. Поскольку производная $y'(t)$ вычисляется приближенно (вектор `y4`), то проверить равенство $y' == 0$ невозможно. Но можно проконтролировать изменение знака производной, проверив знак выражения `y4*y4old`, где `y4` значение производной на текущем шаге, а `y4old` – значение производной на предыдущем шаге. Условие `y4*y4old<=0` выполняется для наивысшей точки траектории. Поэтому, если `y4*y4old<=0`, то следует запомнить значение высоты `y3`. Второе условие `y4<0 and y3<=0.0` обнаруживает первое нулевое или отрицательное значение Y координаты тела при убывании этой координаты (т.е. $y' = y_4 < 0$). Если условие выполнилось, то мы запоминаем дальность и время полета и останавливаем вычисления в методе `integrate()`. Напомним, что для остановки вычислений функция `fout()` должна вернуть `-1`.

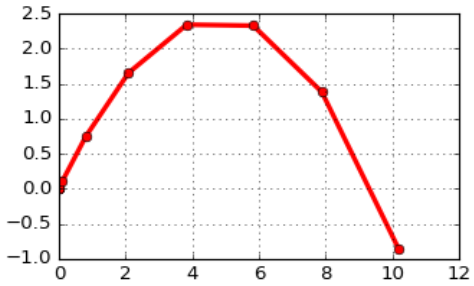
Функция правой части системы уравнений не меняется

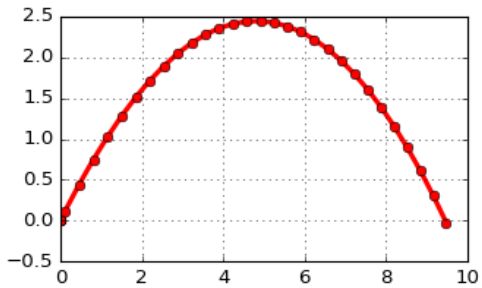
```
def f(t, y):
    k=0.01
    g=9.81
    y1, y2, y3, y4 = y
    return [y2,
            -k*y2*sqrt(y2**2+y4**2),
            y4,
            -k*y4*sqrt(y2**2+y4**2)-g]
```

Последующий код определяет максимально допустимый момент времени, решает систему ОДУ, печатает значение полетного времени, дальность и максимальную высоту. Затем строится траекторию движения тела. Обратите

внимание на инструкцию `ODE.set_integrator('dopri5', max_step=0.05)`, которая использует опцию `max_step`. Если ее не задавать, то метод использует слишком большой шаг, и вы получите грубую траекторию, показанную на следующем рисунке слева. При задании опции `max_step=0.05` количество точек будет больше и график, показанный на следующем рисунке справа, получается более гладким.

```
tmax=100        # максимально допустимый момент времени
alph=np.pi/4    # угол бросания тела
v0=10.0         # начальная скорость

ODE=ode(f)
y0,t0=[0, v0*np.cos(alph), 0, v0*np.sin(alph)], 0  # начальные условия
#r=ODE.set_integrator('dopri5')                    # след. график слева
r=ODE.set_integrator('dopri5', max_step=0.05)  # след. график справа
r.set_solout(fout)
r=ODE.set_initial_value(y0, t0)
ret = r.integrate(tmax)
print('Flight time = %.4f Distance = %.4f Height =%.4f '\
                              % (FlightTime,Distance,Height))
Y=np.array(ys)
fig, ax = plt.subplots()
fig.set_facecolor('white')
ax.plot(Y[:,0],Y[:,2],'-or',linewidth=3)
ax.grid(True)
```





В результате вычислений мы получаем следующие значения времени движения, дальности и максимальной высоты.

```
Flight time = 1.4157 Distance = 9.4739 Height =2.4443
```

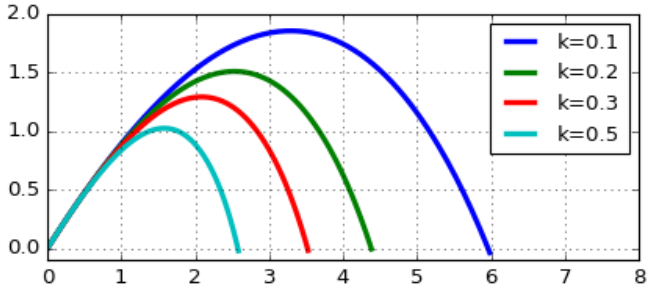
Проведем в этой задаче исследование зависимости дальности полета от коэффициента сопротивления среды. Для этого мы должны переписать код функции правой части системы так, чтобы она содержала дополнительный аргумент `k` – коэффициент сопротивления среды.

```
FlightTime, Distance, Height, y4old =0,0,0,0
def fout(t, y):             # обработчик шага
    global FlightTime, Distance, Height,y4old
    ts.append(t)
    ys.append(list(y.copy()))
    y1, y2, y3, y4 = y
    if y4*y4old<=0:                  # достигнута точка максимума
        Height=y3
```

```
    if y4<0 and y3<=0.0:          # тело достигло поверхности
        FlightTime=t
        Distance=y1
        return -1
    y4old=y4
# функция правых частей системы ОДУ
def f(t, y, k):           #  имеется дополнительный аргумент k
    g=9.81
    y1, y2, y3, y4 = y
    return [y2,
            -k*y2*sqrt(y2**2+y4**2),
            y4,
            -k*y4*sqrt(y2**2+y4**2)-g]

tmax=100                  # максимально допустимый момент аремени
alph=np.pi/4              # угол бросания тела
v0=10.0                   # начальная скорость
K=[0.1,0.2,0.3,0.5]       # анализируемые коэффициенты сопротивления
y0,t0=[0, v0*np.cos(alph), 0, v0*np.sin(alph)], 0   # начальные условия

ODE=ode(f)
ODE.set_integrator('dopri5', max_step=0.01)
ODE.set_solout(fout)
fig, ax = plt.subplots()
fig.set_facecolor('white')
for k in K:               # перебор значений коэффициента сопртивления
    ts, ys = [ ],[ ]
    ODE.set_initial_value(y0, t0) # задание начальных значений
    ODE.set_f_params(k)           # передача дополнительного аргумента k
                          # в функцию f(t,y,k) правых частей системы ОДУ
    ODE.integrate(tmax)           # решение ОДУ
    print('Flight time = %.4f Distance = %.4f Height =%.4f '\
          % (FlightTime,Distance,Height))
    Y=np.array(ys)
    ax.plot(Y[:,0],Y[:,2],linewidth=3,label='k=%.1f'% k)
ax.grid(True)
ax.set_xlim(0,8)
ax.set_ylim(-0.1,2)
```

Здесь мы использовали метод `ODE.set_f_params(k)`, который функции правой части `f(t, y, k)` передает дополнительный параметр `k`. Обратите внимание на то, что в цикле для построения новой траектории мы каждый раз обновляем начальные условия.

Графики траекторий показаны на следующем рисунке.



Параметры траекторий печатаются в окне исполнительной системы.

```
Flight time = 1.2316 Distance = 5.9829 Height =1.8542
Flight time = 1.1016 Distance = 4.3830 Height =1.5088
Flight time = 1.0197 Distance = 3.5265 Height =1.2912
Flight time = 0.9068 Distance = 2.5842 Height =1.0240
```

Пример. Упругий мяч имеет начальное положение и скорость. Сопротивление воздуха пренебрежимо мало, но энергия движения расходуется при отскоке мяча от земли. Пусть при отскоке от земли вертикальная скорость мяча составляет 90% от вертикальной скорости в момент падения. Смоделируем такое движение.

Уравнение движения (без учета сопротивления воздуха) имеет вид

$$m\ddot{\mathbf{r}} = -m\mathbf{g},$$

где $\mathbf{g}$ – вектор ускорения свободного падения, имеющий направление вертикально вниз. В покоординатной форме уравнение принимает вид $\ddot{x} = 0, \ddot{y} = -g$. Задачу следует дополнить начальными условиями $x(0) = x_0, y(0) = h, \dot{x}(0) = v_0^x, \dot{y}(0) = v_0^y$. Система распадается на два независимых уравнения, первое из которых имеет решение $x(t) = x_0 + v_0^x \cdot t$. Это указывает на то, что в горизонтальном направлении движение происходит с постоянной скоростью v_0^x. В нашем примере положим $x_0 = 0$. Второе уравнение также интегрируется, но мы хотим показать, как его можно решить численно, используя функцию `ode()`. В момент отскока t_k вертикальная скорость $\dot{y}$ меняет знак и абсолютное значение, т.е. $\dot{y}_{после\ отскока}(t_k) = -0.9 \cdot \dot{y}_{до\ отскока}(t_k)$.

Функция правой части системы ОДУ 1 – го порядка имеет вид:

```
def f(t, y):
    g=9.81
    y3, y4 = y
    return [y4,-g]
```

Функция обработчика шага имеет вид:

```
def fout(t, y):
    ts.append(t)                    # запоминаем моменты времени
    ys.append(list(y.copy()))       # и значения y и vy
    y3, y4 = y
    if y4<0 and y3<=0.0:            # при достижении поверхности
        return -1                   # остановить интегрирование
```

Создаем функцию горизонтальной координаты тела.

```
horX=lambda t: vox*t                # x(t) = x0 + vx · t, где x0 = 0
```

Вводим исходные данные.

```
ts,ys = [ ],[ ]
tmax=15          # максимально допустимый момент аремени
vox=1            # начальная скорость вдоль оси x
voy=10           # начальная скорость вдоль оси y
```

Создаем ODE объект.

```
ODE=ode(f)
ODE.set_integrator('dopri5', max_step=0.01, nsteps=500)
ODE.set_solout(fout)
Y0, t0 =[0,voy], 0      # начальные значения (начальная высота=0)
```

Осталось задать ODE объекту начальные условия и решить систему ОДУ. При достижении мячом земли функция `fout()` останавливает интегрирование, предварительно запомнив все моменты времени и координаты решения в глобальных списках `ts` и `ys`. Чтобы строить решение в последующие моменты времени, мы должны возобновить вычисления с новыми начальными условиями, соответствующими моменту отскока. Последние элементы в списках `ts` и `ys` (время и скорость) используются при задании новых начальных значений для следующего гладкого участка траектории мяча. При этом вертикальная скорость мяча в момент отскока становится положительной и составляет 90% от абсолютного значения вертикальной скорости в момент падения. На всех участках используются общие списки `ts` и `ys`. После завершения цикла все последовательные значения моментов времени находятся в списке `ts`, а соответствующие им значения вертикальной координаты и скорости – в списке `ys`.

```
for i in range(6):      # моделируем 6 отскоков
    ODE.set_initial_value(Y0, t0)
    ODE.integrate(tmax)
    t0=ts[-1]                       # новый начальный момент времени
    Y0=[0,0.9*abs(ys[-1][1])]       # новые начальные значения
```

Создаем массивы горизонтальных X и вертикальных Y[:,0] координат мяча, и используем их для построения траектории мяча.

```
t=np.array(ts)
Y=np.array(ys)
X=horX(t)
fig, ax = plt.subplots()
fig.set_facecolor('white')
ax.plot(X,Y[:,0],'-',linewidth=3)
ax.grid(True)
```

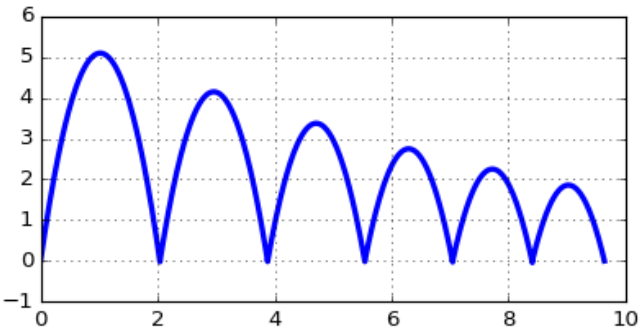


График траектории движения построен для вектора начальной скорости $\mathbf{v}(0)=(v_0^x, v_0^y)=(1,10)$.