

Оглавление

1	Основы языка питон	3
1.1	Числа	4
1.2	Строки	9
1.3	Списки	15
1.4	Кортежи	24
1.5	Множества	26
1.6	Словари	29
1.7	Функции	33
1.8	Итераторы	42
1.9	Объектно-ориентированное программирование	48
1.10	Исключения	63
1.11	Модули	66
1.12	Ввод-вывод, файлы, директории	79
2	Пакеты для научных вычислений	89
2.1	numpy	90
2.1.1	Одномерные массивы	90
2.1.2	Операции над одномерными массивами	96
2.1.3	2-мерные массивы	103
2.1.4	Линейная алгебра	108
2.1.5	Преобразование Фурье	110
2.1.6	Интегрирование	111
2.1.7	Дифференциальные уравнения	111
2.2	matplotlib	113
2.2.1	Логарифмический масштаб	125
2.2.2	Полярные координаты	127
2.2.3	Экспериментальные данные	129
2.2.4	Гистограмма	130
2.2.5	Контурные графики	131
2.2.6	Images (пиксельные картинки)	134
2.2.7	Трёхмерная линия	135
2.2.8	Поверхности	137
2.3	mpmath	141
2.3.1	Специальные функции	143
2.3.2	Решение уравнений	146
2.3.3	Производные	148
2.3.4	Интегралы	148

2.3.5	Сумма ряда	149
2.3.6	Дифференциальные уравнения	149
2.3.7	Матрицы	150
2.4	pandas	153
2.4.1	Series	153
2.4.2	DataFrame	158
2.5	SymPy	166
2.5.1	Многочлены и рациональные функции	167
2.5.2	Элементарные функции	169
2.5.3	Структура выражений	173
2.5.4	Решение уравнений	177
2.5.5	Ряды	178
2.5.6	Производные	181
2.5.7	Интегралы	182
2.5.8	Суммирование рядов	183
2.5.9	Пределы	184
2.5.10	Дифференциальные уравнения	184
2.5.11	Линейная алгебра	185
2.5.12	Графики	191
2.6	IMinuit	200
2.6.1	Простой пример	200
2.6.2	Дайте мне 3 параметра, и я профитирую слона. С 4 параметрами он будет махать хоботом.	203
2.6.3	Резонанс без фона	206
2.7	sh, или как писать shell-скрипты на питоне	211
2.8	plumbum	214
2.8.1	Работа на локальной машине	214
2.8.2	Работа на удалённой машине	221
2.9	RPyC	222
2.10	cython	227
2.10.1	Функции	227
2.10.2	Интерфейс к библиотеке на C	229
2.10.3	Структуры	230
2.10.4	cdef классы	231
2.10.5	Интерфейс к библиотеке на C	233

Глава 1

ОСНОВЫ ЯЗЫКА ПИТОН

1.1 Числа

Арифметические операции имеют ожидаемые приоритеты. При необходимости используются скобки.

```
In [1]: 1+2*3
```

```
Out[1]: 7
```

```
In [2]: (1+2)*3
```

```
Out[2]: 9
```

Возведение целого числа в целую степень даёт целое число, если показатель степени ≥ 0 , и число с плавающей точкой, если он < 0 . Так что тип результата невозможно определить статически, если значение переменной `n` неизвестно.

```
In [3]: n=3  
        2**n
```

```
Out[3]: 8
```

```
In [4]: n=-3  
        2**n
```

```
Out[4]: 0.125
```

Арифметические операции можно применять к целым и числам с плавающей точкой в любых сочетаниях.

```
In [5]: n+1.0
```

```
Out[5]: -2.0
```

Деление целых чисел всегда даёт результат с плавающей точкой, даже если они делятся нацело. Операторы `//` и `%` дают целое частное и остаток.

```
In [6]: 7/4
```

```
Out[6]: 1.75
```

```
In [7]: 7//4
```

```
Out[7]: 1
```

```
In [8]: 7%4
```

```
Out[8]: 3
```

```
In [9]: 4/2
```

```
Out[9]: 2.0
```

Если Вы попытаетесь использовать переменную, которой не присвоено никакого значения, то получите сообщение об ошибке.

```
In [10]: x+1
```

```
-----  
  
NameError                                Traceback (most recent call last)  
  
  <ipython-input-10-d9a77b2c0933> in <module>()  
----> 1 x+1  
  
NameError: name 'x' is not defined
```

`x+=1` означает `x=x+1`, аналогично для других операций. В питоне строго различаются операторы (например, присваивание) и выражения, так что таких операций, как `++` в C, нет. Хотя вызов функции в выражении может приводить к побочным эффектам.

```
In [11]: x=1  
        x+=1  
        print(x)
```

```
2
```

```
In [12]: x*=2  
        print(x)
```

```
4
```

Оператор `del` уничтожает переменную.

```
In [13]: del x  
        x
```

```
-----  
  
NameError                                Traceback (most recent call last)  
  
  <ipython-input-13-726510e32795> in <module>()  
    1 del x  
----> 2 x  
  
NameError: name 'x' is not defined
```

Любопытная особенность питона: можно использовать привычные из математики сравнения вроде $x < y < z$, которые в других языках пришлось бы записывать как $x < y$ and $y < z$.

```
In [14]: 1<2<=2
```

```
Out[14]: True
```

```
In [15]: 1<2<2
```

```
Out[15]: False
```

Логические выражения можно комбинировать с помощью **and** и **or** (эти операции имеют более низкий приоритет, чем сравнения). Если результат уже ясен из первого операнда, второй операнд не вычисляется. А вот так выглядит оператор **if**.

```
In [16]: n=4
         # Кстати, это комментарий
         if 1<2 and n<3:
             print('T')
         else:
             print('F')
```

```
F
```

```
In [17]: if 1<2 or n<3:
           print('T')
         else:
           print('F')
```

```
T
```

После строчки, заканчивающейся **:**, можно писать последовательность операторов с одинаковым отступом (больше, чем у строчки **if**). Никакого признака конца такой группы операторов не нужно. Первая строчка после **else:**, имеющая тот же уровень отступа, что и **if** и **else:** — это следующий оператор после **if**.

Оператора, аналогичного **case** или **switch**, в питоне нет. Используйте длинную последовательность **if ... elif ... elif ... else**.

```
In [18]: if n==1:
           print('один')
         elif n==2:
           print('два')
         elif n==3:
           print('три')
         else:
           print('много')
```

```
много
```

Есть и условные выражения:

```
In [20]: (0 if n<0 else 1)+1
```

```
Out[20]: 2
```

Обычно в начале пишется *основное* выражение, оно защищается условием в `if`, а после `else` пишется *исключительный случай*.

В питоне немного встроенных функций. Большинство надо импортировать. Элементарные функции импортируют из модуля `math`. Заниматься импортозамещением (писать свою реализацию синуса) не нужно.

```
In [21]: from math import sin,pi
```

```
In [22]: pi
```

```
Out[22]: 3.141592653589793
```

```
In [23]: sin(pi/6)
```

```
Out[23]: 0.49999999999999994
```

Любой объект имеет тип.

```
In [24]: type(2)
```

```
Out[24]: int
```

```
In [25]: type(int)
```

```
Out[25]: type
```

```
In [26]: type(2.1)
```

```
Out[26]: float
```

```
In [27]: type(True)
```

```
Out[27]: bool
```

Имена типов по совместительству являются функциями, преобразующими в этот тип объекты других типов (если такое преобразование имеет смысл).

```
In [28]: float(2)
```

```
Out[28]: 2.0
```

```
In [29]: int(2.0)
```

```
Out[29]: 2
```

```
In [30]: int(2.9)
```

```
Out [30]: 2
```

```
In [31]: int(-2.9)
```

```
Out [31]: -2
```

Преобразование числа с плавающей точкой в целое производится путём отбрасывания дробной части, а не округления. Для округления используется функция `round`.

```
In [32]: round(2.9)
```

```
Out [32]: 3
```


1.2 Строки

Питон хорошо приспособлен для работы с текстовой информацией. В нём есть много операций для работы со строками, несколько способов записи строк (удобных в разных случаях). В современных версиях питона (3.x) строки юникодные, т.е. они могут содержать одновременно русские и греческие буквы, немецкие умляуты и китайские иероглифы.

```
In [1]: s='Какая-нибудь строка \u00F6 \u03B1'
        print(s)
```

Какая-нибудь строка ö α

```
In [2]: 'Эта строка может содержать " внутри'
```

```
Out[2]: 'Эта строка может содержать " внутри'
```

```
In [3]: "Эта строка может содержать ' внутри"
```

```
Out[3]: "Эта строка может содержать ' внутри"
```

```
In [4]: s='Эта содержит и \', и \"'
        print(s)
```

Эта содержит и ', и "

```
In [5]: s="""Строка,
           занимающая
           несколько
           строчек"""
        print(s)
```

Строка,
занимающая
несколько
строчек

```
In [6]: s=="Строка,\nзанимающая\nнесколько\nстрочек"
```

```
Out[6]: True
```

Несколько строковых литералов, разделённых лишь пробелами, слипаются в одну строку. Подчеркнём ещё раз: это должны быть литералы, а не переменные со строковыми значениями. Такой способ записи особенно удобен, когда нужно передать длинную строку при вызове функции.

```
In [7]: s='Такие ' 'строки ' 'слипаются'
        print(s)
```

Такие строки слипаются

```
In [8]: print('Такие\n'  
             'строки\n'  
             'слипаются')
```

Такие
строки
слипаются

В питоне нет специального типа `char`, его роль играют строки длины 1. Функция `ord` возвращает (юникодный) номер символа, а обратная ей функция `chr` возвращает символ (строку длины 1).

```
In [9]: n=ord('a')  
        n
```

```
Out[9]: 1072
```

```
In [10]: chr(n)
```

```
Out[10]: 'a'
```

Функция `len` возвращает длину строки. Она применима не только к строкам, но и к спискам, словарям и многим другим типам, про объекты которых разумно спрашивать, какая у них длина.

```
In [11]: s='0123456789'  
         len(s)
```

```
Out[11]: 10
```

Символы в строке индексируются с 0. Отрицательные индексы используются для счёта с конца: `s[-1]` — последний символ в строке, и т.д.

```
In [12]: s[0]
```

```
Out[12]: '0'
```

```
In [13]: s[3]
```

```
Out[13]: '3'
```

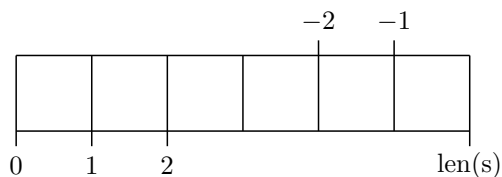
```
In [14]: s[-1]
```

```
Out[14]: '9'
```

```
In [15]: s[-2]
```

```
Out[15]: '8'
```

Можно выделить подстроку, указав диапазон индексов. Подстрока включает символ, соответствующий началу диапазона, но не включает соответствующий концу. Удобно представлять себе, что индексы соответствуют положениям между символами строки. Тогда подстрока `s[n:m]` будет расположена между индексами `n` и `m`.



```
In [16]: s[1:3]
```

```
Out[16]: '12'
```

```
In [17]: s[:3]
```

```
Out[17]: '012'
```

```
In [18]: s[3:]
```

```
Out[18]: '3456789'
```

```
In [19]: s[:-1]
```

```
Out[19]: '012345678'
```

```
In [20]: s[3:-2]
```

```
Out[20]: '34567'
```

Если не указано начало диапазона, подразумевается от начала строки; если не указан его конец — до конца строки.

Строки являются неизменяемым типом данных. Построив строку, нельзя изменить в ней один или несколько символов. Операции над строками строят новые строки — результаты, не меняя своих операндов. Сложение строк означает конкатенацию, а умножение на целое число (с любой стороны) — повторение строки несколько раз.

```
In [21]: s='abc'; t='def'
         s+t
```

```
Out[21]: 'abcdef'
```

```
In [22]: s*3
```

```
Out[22]: 'abcabcabc'
```

Операция `in` проверяет, содержится ли символ (или подстрока) в строке.

```
In [23]: 'a' in s
```

```
Out[23]: True
```

```
In [24]: 'd' in s
```

```
Out[24]: False
```

```
In [25]: 'ab' in s
```

```
Out[25]: True
```

```
In [26]: 'b' not in s
```

```
Out[26]: False
```

У объектов типа строка есть большое количество методов. Метод `lstrip` удаляет все whitespace-символы (пробел, `tab`, `newline`) в начале строки; `rstrip` — в конце; а `strip` — с обеих сторон. Им можно передать необязательный аргумент — символы, которые нужно удалять.

```
In [27]: s='   строка   '
         s.lstrip()
```

```
Out[27]: 'строка   '
```

```
In [28]: s.rstrip()
```

```
Out[28]: '   строка'
```

```
In [29]: s.strip()
```

```
Out[29]: 'строка'
```

`lower` и `upper` переводят все буквы в маленькие и заглавные.

```
In [30]: s='СтРоКа'
         s.lower()
```

```
Out[30]: 'строка'
```

```
In [31]: s.upper()
```

```
Out[31]: 'СТРОКА'
```

Проверки: буквы (маленькие и заглавные), цифры, пробелы.

```
In [32]: 'A6Br'.isalpha()
```

```
Out[32]: True
```

```
In [33]: 'a6Br'.islower()
```

```
Out[33]: True
```

```
In [34]: 'АВВГ'.isupper()
```

```
Out[34]: True
```

```
In [35]: '0123'.isdigit()
```

```
Out[35]: True
```

```
In [36]: ' \t\n'.isspace()
```

```
Out[36]: True
```

Строки имеют тип `str`.

```
In [37]: type(s)
```

```
Out[37]: str
```

```
In [38]: s=str(123)
          s
```

```
Out[38]: '123'
```

```
In [39]: n=int(s)
          n
```

```
Out[39]: 123
```

```
In [40]: int('123x')
```

```
-----
ValueError                                Traceback (most recent call last)

<ipython-input-40-528edaa9c06f> in <module>()
----> 1 int('123x')
```

```
ValueError: invalid literal for int() with base 10: '123x'
```

```
In [41]: x=float('123.456E-7')
          x
```

```
Out[41]: 1.23456e-05
```

Часто требуется вставить в строку значения каких-нибудь переменных (или выражений). Такие строки особенно полезны для печати сообщений. Для этого используются форматные строки: в них в фигурных скобках можно писать выражения, они вычисляются, и их значения подставляются в строку.

```
In [42]: f's = {s}, n = {n}, x = {x}'
Out[42]: 's = 123, n = 123, x = 1.23456e-05'
```

После выражения можно поставить знак `:` и указать некоторые детали того, как это значение должно печататься. В частности, можно задать ширину поля (т.е. число символов). Если значение не влезает в эту ширину поля, для его печати будет использовано больше символов — столько, сколько надо, чтобы напечатать это значение полностью.

```
In [43]: print(f'"{s:5}" "{n:5}" "{x:5}"')
"123  " " 123" "1.23456e-05"
```

Целые числа можно печатать в десятичном, шестнадцатичном или двоичном виде.

```
In [44]: print(f'десятичное "{n:5d}", 16-ричное "{n:5x}", двоичное "{n:5b}"')
десятичное " 123", 16-ричное " 7b", двоичное "1111011"
```

Для чисел с плавающей точкой можно задать число цифр после точки и формат с фиксированной точкой или экспоненциальный.

```
In [45]: print(f'{x:10.5f} {x:10.5e} {1/x:10.5f} {1/x:10.5e}')
0.00001 1.23456e-05 81000.51840 8.10005e+04
```

1.3 Списки

Списки могут содержать объекты любых типов (в одном списке могут быть объекты разных типов). Списки индексируются так же, как строки.

```
In [1]: l=[0,1,2,3,4,5,6,7,8,9]
        1
```

```
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [2]: len(l)
```

```
Out[2]: 10
```

```
In [3]: l[0]
```

```
Out[3]: 0
```

```
In [4]: l[3]
```

```
Out[4]: 3
```

```
In [5]: l[10]
```

```
-----
IndexError                                Traceback (most recent call last)
```

```
<ipython-input-5-e4a648ff0fa9> in <module>()
----> 1 l[10]
```

```
IndexError: list index out of range
```

```
In [6]: l[-2]
```

```
Out[6]: 8
```

```
In [7]: l[1:3]
```

```
Out[7]: [1, 2]
```

Обратите внимание, что `l[:3]+l[3:]`==`l`.

```
In [8]: l[:3]
```

```
Out[8]: [0, 1, 2]
```

```
In [9]: l[3:]
```

```
Out[9]: [3, 4, 5, 6, 7, 8, 9]
```

```
In [10]: l[3:3]
```

```
Out[10]: []
```

```
In [11]: l[3:-2]
```

```
Out[11]: [3, 4, 5, 6, 7]
```

```
In [12]: l[:-2]
```

```
Out[12]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Списки являются изменяемыми объектами. Это сделано для эффективности. В списке может быть 1000000 элементов. Создавать его копию каждый раз, когда мы изменили один элемент, слишком дорого.

```
In [13]: l[3]='три'
1
```

```
Out[13]: [0, 1, 2, 'три', 4, 5, 6, 7, 8, 9]
```

Можно заменить какой-нибудь подсписок на новый список (в том числе другой длины).

```
In [14]: l[3:3]=[0]
1
```

```
Out[14]: [0, 1, 2, 0, 'три', 4, 5, 6, 7, 8, 9]
```

```
In [15]: l[3:3]=[10,11,12]
1
```

```
Out[15]: [0, 1, 2, 10, 11, 12, 0, 'три', 4, 5, 6, 7, 8, 9]
```

```
In [16]: l[5:7]=[0,0,0,0]
1
```

```
Out[16]: [0, 1, 2, 10, 11, 0, 0, 0, 0, 'три', 4, 5, 6, 7, 8, 9]
```

```
In [17]: l[3:]=[]
1
```

```
Out[17]: [0, 1, 2]
```

```
In [18]: l[len(l):]=[3,4]
1
```

```
Out[18]: [0, 1, 2, 3, 4]
```

Некоторые из этих операций могут быть записаны в другой форме.


```
In [19]: l=[0,1,2,3,4,5,6,7]
         del l[3]
         l
```

```
Out[19]: [0, 1, 2, 4, 5, 6, 7]
```

```
In [20]: del l[3:5]
         l
```

```
Out[20]: [0, 1, 2, 6, 7]
```

```
In [21]: l.insert(3,0)
         l
```

```
Out[21]: [0, 1, 2, 0, 6, 7]
```

```
In [22]: l.append(8)
         l
```

```
Out[22]: [0, 1, 2, 0, 6, 7, 8]
```

```
In [23]: l.extend([9,10,11])
         l
```

```
Out[23]: [0, 1, 2, 0, 6, 7, 8, 9, 10, 11]
```

Элементы списка могут быть разных типов.

```
In [24]: l=[0,[1,2,3], 'abc']
         l[1][1]='x'
         l
```

```
Out[24]: [0, [1, 'x', 3], 'abc']
```

Когда мы пишем `m=l`, мы присваиваем переменной `m` ссылку на тот же объект, на который ссылается `l`. Поэтому, изменив этот объект (список) через `l`, мы увидим эти изменения и через `m` — ведь список всего один.

```
In [25]: l=[0,1,2,3,4,5]
         m=l
         l[3]='три'
         m
```

```
Out[25]: [0, 1, 2, 'три', 4, 5]
```

Операция `is` проверяет, являются ли `m` и `l` **одним и тем же объектом**.

```
In [26]: m is l
```

```
Out[26]: True
```

Если мы хотим видоизменять `m` и `l` независимо, нужно присвоить переменной `m` не список `l`, а его копию. Тогда это будут два различных списка, просто в начальный момент они состоят из одних и тех же элементов. Для этого в питоне есть идиома: `l[:]` — это подсписок списка `l` от начала до конца, а подсписок всегда копируется.

```
In [27]: m=l[:]
```

Теперь `m` и `l` — два независимых объекта, имеющих равные значения.

```
In [28]: m is l
```

```
Out[28]: False
```

```
In [29]: m==l
```

```
Out[29]: True
```

Их можно менять независимо.

```
In [30]: l[3]=0
         1
```

```
Out[30]: [0, 1, 2, 0, 4, 5]
```

```
In [31]: m
```

```
Out[31]: [0, 1, 2, 'три', 4, 5]
```

Как и для строк, сложение списков означает конкатенацию, а умножение на целое число — повторение списка несколько раз. Операция `in` проверяет, содержится ли элемент в списке.

```
In [32]: [0,1,2]+[3,4,5]
```

```
Out[32]: [0, 1, 2, 3, 4, 5]
```

```
In [33]: 2*[0,1,2]
```

```
Out[33]: [0, 1, 2, 0, 1, 2]
```

```
In [34]: l=[0,1,2]
         l+=[3,4,5]
         1
```

```
Out[34]: [0, 1, 2, 3, 4, 5]
```

```
In [35]: 2 in l
```

```
Out[35]: True
```

Простейший вид цикла в питоне — это цикл по элементам списка.

```
In [36]: for x in l:
          print(x)
```

```
0
1
2
3
4
5
```

Можно использовать цикл `while`. В этом примере он выполняется, пока список `l` не пуст. Этот цикл гораздо менее эффективен, чем предыдущий — в нём на каждом шаге меняется список `l`. Он тут приведён не для того, чтобы ему подражали, а просто чтобы показать синтаксис цикла `while`.

```
In [37]: while l:
          print(l[0])
          l=l[1:]
```

```
0
1
2
3
4
5
```

```
In [38]: l
```

```
Out[38]: []
```

Очень часто используются циклы по диапазонам целых чисел.

```
In [39]: for i in range(4):
          print(i)
```

```
0
1
2
3
```

Функция `range(n)` возвращает диапазон целых чисел от 0 до $n - 1$ (всего n штук) в виде специального объекта `range`, который можно использовать в `for` цикле. Можно превратить этот объект в список функцией `list`. Но этого делать не нужно, если только такой список не нужен для проведения каких-нибудь списковых операций. Число `n` может быть равно 1000000. Зачем занимать память под длинный список, если он не нужен? Для написания цикла достаточно короткого объекта `range`, который хранит только пределы.

```
In [40]: r=range(4)
         r
```

```
Out[40]: range(0, 4)
```

```
In [41]: list(r)
```

```
Out[41]: [0, 1, 2, 3]
```

Функции `range` можно передать первый параметр — нижний предел.

```
In [42]: for i in range(2,4):
         print(i)
```

```
2
3
```

```
In [43]: r=range(2,4)
         r
```

```
Out[43]: range(2, 4)
```

```
In [44]: list(r)
```

```
Out[44]: [2, 3]
```

Функция `list` превращает строку в список символов.

```
In [45]: l=list('абвгд')
         l
```

```
Out[45]: ['a', 'б', 'в', 'г', 'д']
```

Как написать цикл, если в его теле нужно использовать и номера элементов списка, и сами эти элементы? Первая идея, которая приходит в голову по аналогии с C — это использовать `range`.

```
In [46]: for i in range(len(l)):
         print(i, ' ', l[i])
```

```
0    а
1    б
2    в
3    г
4    д
```

Можно поступить наоборот — устроить цикл по элементам списка, а индексы вычислять.

```
In [47]: i=0
         for x in l:
             print(i, ' ', x)
             i+=1
```

```
0    а
1    б
2    в
3    г
4    д
```

Оба этих способа не есть идиоматический питон. Более изящно использовать функцию `enumerate`, которая на каждом шаге возвращает пару из индекса `i` и `i`-го элемента списка.

```
In [48]: for i,x in enumerate(l):
         print(i, ' ', x)
```

```
0    а
1    б
2    в
3    г
4    д
```

Про такие пары мы поговорим в следующем параграфе.

Довольно часто удобно использовать цикл `while True:`, то есть пока рак на горе не свистнет, а выход (или несколько выходов) из него устраивать в нужном месте (или местах) при помощи `break`.

```
In [49]: while True:
         print(l[-1])
         l=l[:-1]
         if l==[]:
             break
```

```
д
г
в
б
а
```

Этот конкретный цикл — отнюдь не пример для подражания, он просто показывает синтаксис.

Можно строить список поэлементно.

```
In [50]: l=[]
         for i in range(10):
             l.append(i**2)
         l
```

```
Out [50]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Но более компактно и элегантно такой список можно создать при помощи генератора списка (list comprehension). К тому же это эффективнее — размер списка известен заранее, и не нужно много раз увеличивать его. Опытные питон-программисты используют генераторы списков везде, где это возможно (и разумно).

```
In [51]: [i**2 for i in range(10)]
```

```
Out [51]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [52]: [[i,j] for i in range(3) for j in range(2)]
```

```
Out [52]: [[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
```

В генераторе списков могут присутствовать некоторые дополнительные элементы, хотя они используются реже. Например, в список-результат можно включить не все элементы.

```
In [53]: [i**2 for i in range(10) if i!=5]
```

```
Out [53]: [0, 1, 4, 9, 16, 36, 49, 64, 81]
```

Создадим список случайных целых чисел.

```
In [54]: from random import randint
         l=[randint(0,9) for i in range(10)]
         l
```

```
Out [54]: [6, 0, 5, 8, 9, 0, 9, 4, 0, 6]
```

Функция `sorted` возвращает отсортированную копию списка. Метод `sort` сортирует список на месте. Им можно передать дополнительный параметр — функцию, указывающую, как сравнивать элементы.

```
In [55]: sorted(l)
```

```
Out [55]: [0, 0, 0, 4, 5, 6, 6, 8, 9, 9]
```

```
In [56]: l
```

```
Out [56]: [6, 0, 5, 8, 9, 0, 9, 4, 0, 6]
```

```
In [57]: l.sort()
         l
```

```
Out [57]: [0, 0, 0, 4, 5, 6, 6, 8, 9, 9]
```

Аналогично, функция `reversed` возвращает обращённый список (точнее говоря, некий объект, который можно использовать в `for` цикле или превратить в список функцией `list`). Метод `reverse` обращает список на месте.

```
In [58]: list(reversed(l))
```

```
Out[58]: [9, 9, 8, 6, 6, 5, 4, 0, 0, 0]
```

```
In [59]: l
```

```
Out[59]: [0, 0, 0, 4, 5, 6, 6, 8, 9, 9]
```

```
In [60]: l.reverse()  
l
```

```
Out[60]: [9, 9, 8, 6, 6, 5, 4, 0, 0, 0]
```

Метод `split` расщепляет строку в список подстрок. По умолчанию расщепление производится по пустым промежуткам (последовательностям пробелов и символов `tab` и `newline`). Но можно передать ему дополнительный аргумент — подстроку-разделитель.

```
In [61]: s='abc \t def \n ghi'  
l=s.split()  
l
```

```
Out[61]: ['abc', 'def', 'ghi']
```

Чтобы напечатать элементы списка через запятую или какой-нибудь другой символ (или строку), очень полезен метод `join`. Он создаёт строку из всех элементов списка, разделяя их строкой-разделителем. Запрограммировать это в виде цикла было бы существенно длиннее, и такую программу было бы сложнее читать.

```
In [62]: s=', '.join(l)  
s
```

```
Out[62]: 'abc, def, ghi'
```

```
In [63]: s.split(', ')
```

```
Out[63]: ['abc', 'def', 'ghi']
```

1.4 Кортежи

Кортежи (tuples) очень похожи на списки, но являются неизменяемыми. Как мы видели, использование изменяемых объектов может приводить к неприятным сюрпризам.

Кортежи пишутся в круглых скобках. Если элементов > 1 или 0 , это не вызывает проблем. Но как записать кортеж с одним элементом? Конструкция `(x)` абсолютно легальна в любом месте любого выражения, и означает просто `x`. Чтобы избежать неоднозначности, кортеж с одним элементом `x` записывается в виде `(x,)`.

```
In [1]: (1,2,3)
```

```
Out[1]: (1, 2, 3)
```

```
In [2]: ()
```

```
Out[2]: ()
```

```
In [3]: (1,)
```

```
Out[3]: (1,)
```

Скобки ставить не обязательно, если кортеж — единственная вещь в правой части присваивания.

```
In [4]: t=1,2,3
        t
```

```
Out[4]: (1, 2, 3)
```

Работать с кортежами можно так же, как со списками. Нельзя только изменять их.

```
In [5]: len(t)
```

```
Out[5]: 3
```

```
In [6]: t[1]
```

```
Out[6]: 2
```

```
In [7]: u=4,5
        t+u
```

```
Out[7]: (1, 2, 3, 4, 5)
```

```
In [8]: 2*u
```

```
Out[8]: (4, 5, 4, 5)
```

В левой части присваивания можно написать несколько переменных через запятую, а в правой кортеж. Это одновременное присваивание значений нескольким переменным.

```
In [9]: x,y=1,2
```



```
In [10]: x
```

```
Out[10]: 1
```

```
In [11]: y
```

```
Out[11]: 2
```

Сначала вычисляется кортеж в правой части, исходя из *старых* значений переменных (до этого присваивания). Потом одновременно всем переменным присваиваются новые значения из этого кортежа. Поэтому так можно обменять значения двух переменных.

```
In [12]: x,y=y,x
```

```
In [13]: x
```

```
Out[13]: 2
```

```
In [14]: y
```

```
Out[14]: 1
```

Это проще, чем в других языках, где приходится использовать третью переменную.

В стандартной библиотеке есть полезный тип `namedtuple`:

```
In [15]: from collections import namedtuple
         point=namedtuple('point',('x','y','z'))
         p=point(0,1,2)
         print(p)
```

```
point(x=0, y=1, z=2)
```

К его полям можно обращаться как по имени, так и по номеру (как для обычного кортежа).

```
In [16]: p.y,p[1]
```

```
Out[16]: (1, 1)
```

При создании объекта типа `namedtuple` аргументы можно задавать в любом порядке, если указывать их имена.

```
In [17]: p=point(y=1,z=2,x=0)
         p
```

```
Out[17]: point(x=0, y=1, z=2)
```

В этих объектах нет накладных расходов по памяти: только значения полей (как в `structure` в C или `record` в Pascal). Соответствие между именами полей и их номерами хранится в памяти один раз для всего типа (в нашем примере `point`). В этом состоит важное отличие от словарей с ключами-строками. Кроме того, невозможно добавлять или удалять поля налету.

1.5 Множества

В соответствии с математическими обозначениями, множества пишутся в фигурных скобках. Элемент может содержаться в множестве только один раз. Порядок элементов в множестве не имеет значения, поэтому питон их сортирует. Элементы множества могут быть любых типов. Множества используются существенно реже, чем списки. Но иногда они бывают весьма полезны. Например, когда я собирался делать апгрейд системы на сервере, я написал на питоне программу, которая строила множество пакетов, установленных в системе до апгрейда; множество пакетов, имеющихся на инсталляционных CD; имеющихся на основных сайтах с дополнительными пакетами, и т.д. И она мне помогла восстановить функциональность после апгрейда.

```
In [1]: s={0,1,0,5,5,1,0}
        s
```

```
Out[1]: {0, 1, 5}
```

Принадлежит ли элемент множеству?

```
In [2]: 1 in s, 2 in s, 1 not in s
```

```
Out[2]: (True, False, False)
```

Множество можно получить из списка, или строки, или любого объекта, который можно использовать в `for` цикле (итерабельного).

```
In [3]: l=[0,1,0,5,5,1,0]
        set(l)
```

```
Out[3]: {0, 1, 5}
```

```
In [4]: set('a66a')
```

```
Out[4]: {'a', '6'}
```

Как записать пустое множество? Только так.

```
In [5]: set()
```

```
Out[5]: set()
```

Дело в том, что в фигурных скобках в питоне пишутся также словари (мы будем их обсуждать в следующем параграфе). Когда в них есть хоть один элемент, можно отличить словарь от множества. Но пустые фигурные скобки означают пустой словарь.

Работать с множествами можно как со списками.

```
In [6]: len(s)
```

```
Out[6]: 3
```

```
In [7]: for x in s:
        print(x)
```

```
0
1
5
```

Это генератор множества (set comprehension).

```
In [8]: {i for i in range(5)}
```

```
Out[8]: {0, 1, 2, 3, 4}
```

Объединение множеств.

```
In [9]: s2=s|{2,5}
        s2
```

```
Out[9]: {0, 1, 2, 5}
```

Проверка того, является ли одно множество подмножеством другого.

```
In [10]: s<s2, s>s2, s<=s2, s>=s2
```

```
Out[10]: (True, False, True, False)
```

Пересечение.

```
In [11]: s2&{1,2,3}
```

```
Out[11]: {1, 2}
```

Разность и симметричная разность.

```
In [12]: s2-{1,3,5}
```

```
Out[12]: {0, 2}
```

```
In [13]: s2^ {1,3,5}
```

```
Out[13]: {0, 2, 3}
```

Множества (как и списки) являются изменяемыми объектами. Добавление элемента в множество и исключение из него.

```
In [14]: s2.add(4)
        s2
```

```
Out[14]: {0, 1, 2, 4, 5}
```

```
In [15]: s2.remove(1)
        s2
```

```
Out[15]: {0, 2, 4, 5}
```

Как и в случае `+=`, можно скомбинировать теоретико-множественную операцию с присваиванием.

```
In [16]: s2|={1,2}
         s2
```

```
Out[16]: {0, 1, 2, 4, 5}
```

Существуют также неизменяемые множества. Этот тип данных называется **frozenset**. Операции над такими множествами подобны обычным, только невозможно изменять их (добавлять и исключать элементы).

1.6 Словари

Словарь содержит пары ключ — значение (их порядок несущественен). Это один из наиболее полезных и часто используемых типов данных в питоне.

```
In [1]: d={'one':1,'two':2,'three':3}
        d
```

```
Out[1]: {'one': 1, 'three': 3, 'two': 2}
```

Можно узнать значение, соответствующее некоторому ключу. Словари реализованы как хэш-таблицы, так что поиск даже в больших словарях очень эффективен. В языках низкого уровня (например, C) для построения хэш-таблиц требуется использовать внешние библиотеки и писать заметное количество кода. В скриптовых языках (perl, python, php) они уже встроены в язык, и использовать их очень легко.

```
In [2]: d['two']
```

```
Out[2]: 2
```

```
In [3]: d['four']
```

```
-----
KeyError                                Traceback (most recent call last)

<ipython-input-3-a0944cd0c15b> in <module>()
----> 1 d['four']

KeyError: 'four'
```

Можно проверить, есть ли в словаре данный ключ.

```
In [4]: 'one' in d, 'four' in d
```

```
Out[4]: (True, False)
```

Можно присваивать значения как имеющимся ключам, так и отсутствующим (они добавятся к словарю).

```
In [5]: d['one']=-1
        d
```

```
Out[5]: {'one': -1, 'three': 3, 'two': 2}
```

```
In [6]: d['four']=4
        d
```

```
Out[6]: {'four': 4, 'one': -1, 'three': 3, 'two': 2}
```

Длина — число ключей в словаре.

```
In [7]: len(d)
```

```
Out[7]: 4
```

Можно удалит ключ из словаря.

```
In [8]: del d['two']  
d
```

```
Out[8]: {'four': 4, 'one': -1, 'three': 3}
```

Метод `get`, если он будет вызван с отсутствующим ключом, не приводит к ошибке, а возвращает специальный объект `None`. Он используется всегда, когда необходимо указать, что объект отсутствует (в какой-то мере он аналогичен `null` в C). Если передать методу `get` второй аргумент — значение по умолчанию, то будет возвращаться это значение, а не `None`.

```
In [9]: d.get('one'),d.get('five')
```

```
Out[9]: (-1, None)
```

```
In [10]: type(None)
```

```
Out[10]: NoneType
```

```
In [11]: d.get('one',0),d.get('five',0)
```

```
Out[11]: (-1, 0)
```

Словари обычно строят последовательно: начинают с пустого словаря, а затем добавляют ключи со значениями.

```
In [12]: d={}  
d
```

```
Out[12]: {}
```

```
In [13]: d['zero']=0  
d
```

```
Out[13]: {'zero': 0}
```

```
In [14]: d['one']=1  
d
```

```
Out[14]: {'one': 1, 'zero': 0}
```

А это генератор словаря (dictionary comprehension).

```
In [15]: d={i:i**2 for i in range(5)}
         d
```

```
Out[15]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Ключами могут быть любые неизменяемые объекты, например, целые числа, строки, кортежи.

```
In [16]: d={}
         d[0,0]=1
         d[0,1]=0
         d[1,0]=0
         d[1,1]=-1
         d
```

```
Out[16]: {(0, 0): 1, (0, 1): 0, (1, 0): 0, (1, 1): -1}
```

```
In [17]: d[0,0]+d[1,1]
```

```
Out[17]: 0
```

Словари, подобно спискам, можно использовать в `for` циклах. Перебираются имеющиеся в словаре ключи (в каком-то непредсказуемом порядке).

```
In [18]: d={'one':1,'two':2,'three':3}
         for x in d:
             print(x, ' ',d[x])
```

```
one      1
two      2
three    3
```

Метод `keys` возвращает список ключей, метод `values` — список соответствующих значений (в том же порядке), а метод `items` — список пар (ключ,значение). Точнее говоря, это не списки, а некоторые объекты, которые можно использовать в `for` циклах или превратить в списки функцией `list`. Если хочется написать цикл по упорядоченному списку ключей, то можно использовать `sorted(d.keys())`.

```
In [19]: d.keys(),d.values(),d.items()
```

```
Out[19]: (dict_keys(['one', 'two', 'three']),
         dict_values([1, 2, 3]),
         dict_items([('one', 1), ('two', 2), ('three', 3)]))
```

```
In [20]: for x in sorted(d.keys()):
         print(x, ' ',d[x])
```

```
one      1
three    3
two      2
```

```
In [21]: for x,y in d.items():  
         print(x, ' ',y)
```

```
one      1  
two      2  
three    3
```

```
In [22]: del x,y
```

Что есть истина? И что есть ложь? Подойдём к этому философскому вопросу экспериментально.

```
In [23]: bool(False),bool(True)
```

```
Out[23]: (False, True)
```

```
In [24]: bool(None)
```

```
Out[24]: False
```

```
In [25]: bool(0),bool(123)
```

```
Out[25]: (False, True)
```

```
In [26]: bool(''),bool(' ')
```

```
Out[26]: (False, True)
```

```
In [27]: bool([]),bool([0])
```

```
Out[27]: (False, True)
```

```
In [28]: bool(set()),bool({0})
```

```
Out[28]: (False, True)
```

```
In [29]: bool({}),bool({0:0})
```

```
Out[29]: (False, True)
```

На выражения, стоящие в булевых позициях (после `if`, `elif` и `while`), неявно напускается функция `bool`. Некоторые объекты интерпретируются как `False`: число 0, пустая строка, пустой список, пустое множество, пустой словарь, `None` и некоторые другие. Все остальные объекты интерпретируются как `True`. В операторах `if` или `while` очень часто используется список, словарь или что-нибудь подобное, что означает делай что-то если этот список (словарь и т.д.) не пуст.

Заметим, что число с плавающей точкой 0.0 тоже интерпретируется как `False`. Это использовать категорически не рекомендуется: вычисления с плавающей точкой всегда приближённые, и неизвестно, получите Вы 0.0 или 1.234E-12. Лучше напишите `if abs(x)<epsilon:`.

1.7 Функции

Это простейшая в мире функция. Она не имеет параметров, ничего не делает и ничего не возвращает. Оператор `pass` означает “ничего не делай”; он используется там, где синтаксически необходим оператор, а делать ничено не нужно (после `if` или `elif`, после `def` и т.д.).

```
In [1]: def f():
        pass

In [2]: f

Out[2]: <function __main__.f>

In [3]: pass

In [4]: type(f)

Out[4]: function

In [5]: r=f()
        print(r)

None
```

Эта функция более полезна: она имеет параметр и что-то возвращает.

```
In [6]: def f(x):
        return x+1

In [7]: f(1),f(1.0)

Out[7]: (2, 2.0)

In [8]: f('abc')
```

```
-----

TypeError                                Traceback (most recent call last)

<ipython-input-8-410386031a44> in <module>()
----> 1 f('abc')
```

```

<ipython-input-6-e9c32f618734> in f(x)
      1 def f(x):
----> 2     return x+1

TypeError: must be str, not int
```

Если у функции много параметров, то возникает желание вызывать её попроще в наиболее часто встречающихся случаях. Для этого в операторе `def` можно задать значения некоторых параметров по умолчанию (они должны размещаться в конце списка параметров). При вызове необходимо указать все обязательные параметры (у которых нет значений по умолчанию), а необязательные можно и не указывать. Если при вызове указывать параметры в виде `имя=значение`, то это можно делать в любом порядке. Это гораздо удобнее, чем вспоминать, является данный параметр восьмым или девятым при вызове какой-нибудь сложной функции.

```
In [9]: def f(x,a=0,b='b'):  
        print(x, ' ',a, ' ',b)
```

```
In [10]: f(1.0)
```

```
1.0    0    b
```

```
In [11]: f(1.0,1)
```

```
1.0    1    b
```

```
In [12]: f(1.0,b='a')
```

```
1.0    0    a
```

```
In [13]: f(1.0,b='a',a=2)
```

```
1.0    2    a
```

```
In [14]: f(a=2,x=2.0)
```

```
2.0    2    b
```

Переменные, используемые в функции, являются локальными. Присваивание им не меняет значений глобальных переменных с такими же именами.

```
In [15]: a=1
```

```
In [16]: def f():  
        a=2  
        return a
```

```
In [17]: f()
```

```
Out[17]: 2
```

```
In [18]: a
```

```
Out[18]: 1
```

Если в функции нужно использовать какие-нибудь глобальные переменные, их нужно описать как `global`.

```
In [19]: def f():  
         global a  
         a=2  
         return a
```

```
In [20]: f()
```

```
Out[20]: 2
```

```
In [21]: a
```

```
Out[21]: 2
```

Пространство имён устанавливает соответствие между именами переменных и объектами — их значениями. Есть пространство имён локальных переменных функции, пространство имён глобальных переменных программы и пространство имён встроенных функций языка питон. Для реализации пространств имён используются словари.

Если функции передаётся в качестве аргумента какой-нибудь изменяемый объект, и функция его изменяет, то это изменение будет видно снаружи после этого вызова. Мы уже обсуждали эту ситуацию, когда две переменные (в данном случае глобальная переменная и параметр функции) указывают на один и тот же изменяемый объект объект.

```
In [22]: def f(x,l):  
         l.append(x)  
         return l
```

```
In [23]: l=[1,2,3]  
         f(0,l)
```

```
Out[23]: [1, 2, 3, 0]
```

```
In [24]: l
```

```
Out[24]: [1, 2, 3, 0]
```

Если в качестве значения какого-нибудь параметра по умолчанию используется изменяемый объект, то это может приводить к неожиданным последствиям. В данном случае исполнение определения функции приводит к созданию двух объектов: собственно функции и объекта-списка, первоначально пустого, который используется для инициализации параметра функции при вызове. Функция изменяет этот объект. При следующем вызове он опять используется для инициализации параметра, но его значение уже изменилось.

```
In [25]: def f(x,l=[]):  
         l.append(x)  
         return l
```

```
In [26]: f(0)
```

```
Out[26]: [0]
```

```
In [27]: f(1)
```

```
Out[27]: [0, 1]
```

```
In [28]: f(2)
```

```
Out[28]: [0, 1, 2]
```

Чтобы избежать таких сюрпризов, в качестве значений по умолчанию лучше использовать только неизменяемые объекты.

```
In [29]: def f(x,l=None):  
         if l is None:  
             l=[]  
         l.append(x)  
         return l
```

```
In [30]: f(0)
```

```
Out[30]: [0]
```

```
In [31]: f(1)
```

```
Out[31]: [1]
```

```
In [32]: f(2,[0,1])
```

```
Out[32]: [0, 1, 2]
```

Эта функция имеет один обязательный параметр плюс произвольное число необязательных. При вызове все такие дополнительные аргументы объединяются в кортеж, который функция может использовать по своему усмотрению.

```
In [33]: def f(x,*l):  
         print(x, ' ',l)
```

```
In [34]: f(0)
```

```
0    ()
```

```
In [35]: f(0,1)
```

```
0    (1,)
```

```
In [36]: f(0,1,2)
```

```
0    (1, 2)
```

```
In [37]: f(0,1,2,3)
```

```
0    (1, 2, 3)
```

Звёздочку можно использовать и при вызове функции. Можно заранее построить список (или кортеж) аргументов, а потом вызвать функцию с этими аргументами.

```
In [38]: l=[1,2]
         f(*l)
```

```
1    (2,)
```

```
In [39]: c=('a','b')
         f(*l,0,*c)
```

```
1    (2, 0, 'a', 'b')
```

Такую распаковку из списков и кортежей можно использовать не только при вызове функции, но и при построении списка или кортежа.

```
In [40]: (*l,0,*c)
```

```
Out[40]: (1, 2, 0, 'a', 'b')
```

```
In [41]: [*l,0,*c]
```

```
Out[41]: [1, 2, 0, 'a', 'b']
```

```
In [42]: [*l,3]
```

```
Out[42]: [1, 2, 3]
```

```
In [43]: [3,*l]
```

```
Out[43]: [3, 1, 2]
```

Эта функция имеет два обязательных параметра плюс произвольное число необязательных ключевых параметров. При вызове они должны задаваться в виде `имя=значение`. Они собираются в словарь, который функция может использовать по своему усмотрению.

```
In [44]: def f(x,y,**d):
         print(x, ' ', y, ' ', d)
```

```
In [45]: f(0,1,foo=2,bar=3)
```

```
0    1    {'foo': 2, 'bar': 3}
```

Двойную звёздочку можно использовать и при вызове функции. Можно заранее построить словарь аргументов, сопоставляющий значения именам параметров, а потом вызвать функцию с этими ключевыми аргументами.

```
In [46]: d={'foo':2, 'bar':3}
         f(0,1,**d)
```

```
0    1    {'foo': 2, 'bar': 3}
```

```
In [47]: d['x']=0
         d['y']=1
         f(**d)
```

```
0    1    {'foo': 2, 'bar': 3}
```

Вот любопытный способ построить словарь с ключами-строками.

```
In [48]: def f(**d):
         return d
```

```
In [49]: f(x=0,y=1,z=2)
```

```
Out[49]: {'x': 0, 'y': 1, 'z': 2}
```

Двойную звёздочку можно использовать не только при вызове функции, но и при построении словаря.

```
In [50]: d={0:'a',1:'b'}
         {**d,2:'c'}
```

```
Out[50]: {0: 'a', 1: 'b', 2: 'c'}
```

Вот простой способ объединить два словаря.

```
In [51]: d1={0:'a',1:'b'}
         d2={2:'c',3:'d'}
         {**d1,**d2}
```

```
Out[51]: {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
```

Если один и тот же ключ встречается несколько раз, следующее значение затирает предыдущее.

```
In [52]: d2={1:'B',2:'C'}
         {**d1,3:'D',**d2,3:'d'}
```

```
Out [52]: {0: 'a', 1: 'B', 2: 'C', 3: 'd'}
```

Это наиболее общий вид списка параметров функции. Сначала идут обязательные параметры (в данном случае два), затем произвольное число необязательных (при вызове они будут объединены в кортеж), а затем произвольное число ключевых параметров (при вызове они будут объединены в словарь).

```
In [53]: def f(x,y,*l,**d):
          print(x, ' ', y, ' ', l, ' ', d)
```

```
In [54]: f(0,1,2,3,foo=4,bar=5)
```

```
0      1      (2, 3)      {'foo': 4, 'bar': 5}
```

Функции можно передать функцию в качестве аргумента. Например, эта функция реализует численное интегрирование по формуле Симпсона. Её первый параметр — функция, которую надо проинтегрировать; далее задаются пределы интегрирования и число интервалов, на которое нужно разбить область интегрирования.

```
In [55]: def simpson(f,a,b,n):
          h=(b-a)/(2*n)
          s=0.5*(f(a)+f(b))+2*f(a+h)
          x=a+2*h
          for i in range(n-1):
              s+=f(x)+2*f(x+h)
              x+=2*h
          return 2/3*h*s
```

```
In [56]: from math import sin,pi
          [simpson(sin,0,pi,n) for n in [1,10,100,1000]]
```

```
Out [56]: [2.0943951023931953, 2.0000067844418012, 2.000000000676474, 2.000000000000091]
```

В питоне функции являются гражданами первого сорта. Они могут присутствовать везде, где допустимы объекты других типов — среди элементов списков, значений в словарях и т.д.

```
In [57]: def f0(x):
          return x+2
```

```
In [58]: def f1(x):
          return 2*x
```

```
In [59]: l=[f0,f1]
          l
```

```
Out [59]: [<function __main__.f0>, <function __main__.f1>]
```

```
In [60]: x=2.0
          n=1
          l[n](x)
```

Out [60]: 4.0

Если Вы пишете функцию не для того, чтобы один раз её вызвать и навсегда забыть, то нужна документация, объясняющая, что эта функция делает. Для этого сразу после строчки `def` пишется строка. Она называется док-строкой, и сохраняется при трансляции исходного текста на питоне в байт-код (в отличие от комментариев, которые при этом отбрасываются). Обычно эта строка заключается в тройные кавычки и занимает несколько строчек. Док-строка доступна как атрибут `__doc__` функции, и используется функцией `help`. Вот пример культурно написанной функции, вычисляющей n -е число Фибоначчи.

Для проверки типов аргументов, переданных функции, удобно использовать оператор `assert`. Если условие в нём истинно, всё в порядке, и он ничего не делает; если же оно ложно, выдаётся сообщение об ошибке.

```
In [61]: def fib(n):
        "вычисляет n-е число Фибоначчи"
        assert type(n) is int and n>0
        if n<=2:
            return 1
        x,y=1,1
        for i in range(n-2):
            x,y=y,x+y
        return y
```

```
In [62]: fib.__doc__
```

Out [62]: 'вычисляет n-е число Фибоначчи'

```
In [63]: help(fib)
```

Help on function fib in module __main__:

```
fib(n)
    вычисляет n-е число Фибоначчи
```

```
In [64]: [fib(n) for n in range(1,10)]
```

Out [64]: [1, 1, 2, 3, 5, 8, 13, 21, 34]

```
In [65]: fib(-1)
```

```
-----
AssertionError                                Traceback (most recent call last)

<ipython-input-66-b876e14fb318> in <module>()
----> 1 fib(-1)
```



```
<ipython-input-62-800ccd3a2d90> in fib(n)
      1 def fib(n):
      2     "вычисляет n-е число Фибоначчи"
----> 3     assert type(n) is int and n>0
      4     if n<=2:
      5         return 1
```

AssertionError:

In [66]: fib(2.0)

AssertionError Traceback (most recent call last)

```
<ipython-input-67-363564e722ae> in <module>()
----> 1 fib(2.0)
```

```
<ipython-input-62-800ccd3a2d90> in fib(n)
      1 def fib(n):
      2     "вычисляет n-е число Фибоначчи"
----> 3     assert type(n) is int and n>0
      4     if n<=2:
      5         return 1
```

AssertionError:

1.8 Итераторы

На выражение, стоящее после `for x in`, питон автоматически напускает функцию `iter`. Она возвращает объект — итератор. Существуют и выражения-итераторы. Они выглядят как генераторы списков, но пишутся в круглых скобках, а не в квадратных. Сравним следующие 2 примера:

```
In [1]: s=0
        for n in [i**2 for i in range(1000)]:
            s+=n
        s
```

```
Out[1]: 332833500
```

```
In [2]: s=0
        for n in (i**2 for i in range(1000)):
            s+=n
        s
```

```
Out[2]: 332833500
```

В первом случае в памяти создаётся список из 1000 элементов. Во втором в памяти хранится только короткое выражение — итератор. Оно выдаёт очередные члены последовательности по одному, по мере надобности.

Посмотрим, как работает такое выражение.

```
In [3]: it=(i**2 for i in range(4) if i!=2)
        it
```

```
Out[3]: <generator object <genexpr> at 0x7fceb40e6e60>
```

```
In [4]: next(it)
```

```
Out[4]: 0
```

```
In [5]: next(it)
```

```
Out[5]: 1
```

```
In [6]: next(it)
```

```
Out[6]: 9
```

```
In [7]: next(it)
```

```
-----
StopIteration
```

```
Traceback (most recent call last)
```

```
<ipython-input-7-2cdb14c0d4d6> in <module>()
```

```
----> 1 next(it)
```

```
StopIteration:
```

Итераторы могут использоваться не только в циклах. Есть много функций с аргументами - итераторами.

```
In [8]: max((10*x-x**2 for x in range(10)))
```

```
Out[8]: 25
```

Функция `min` аналогична. В таких случаях, когда выражение - итератор является единственным аргументом функции, заключать его в скобки не обязательно.

```
In [9]: sum(10*x-x**2 for x in range(10))
```

```
Out[9]: 165
```

Часто хочется применить какую-нибудь функцию к каждому элементу последовательности. Это делает функция `map`, она возвращает объект `map`, который тоже является итератором.

```
In [10]: def f(x):  
         return x**2
```

```
In [11]: m=map(f, [0,1,2])  
         m
```

```
Out[11]: <map at 0x7fceb40a0cc0>
```

```
In [12]: list(m)
```

```
Out[12]: [0, 1, 4]
```

Часто бывает нужна какая-нибудь очень простая функция. Не хочется придумывать для неё имя, которое будет использовано всего 1 раз, и засорять пространство имён. В таких случаях лучше использовать анонимную функцию:

```
In [13]: list(map(lambda x:2*x, [0,1,2]))
```

```
Out[13]: [0, 2, 4]
```

Анонимные функции записываются так:

```
In [14]: f=lambda x,y:x+2*y  
         f
```

```
Out[14]: <function __main__.<lambda>>
```

Их, естественно, можно вызывать:

```
In [15]: f(1,2)
```

```
Out[15]: 5
```

К сожалению, только очень простые функции можно записать в виде анонимных — они должны состоять из одного единственного выражения. Для многострочных функций это невозможно.

Ещё одна полезная функция — `filter`, она позволяет отфильтровать последовательность, оставив в ней только те элементы, которые удовлетворяют некоторому условию.

```
In [16]: list(filter(lambda x:x>0,[0,1,-2,3,-4]))
```

```
Out[16]: [1, 3]
```

Выражения-итераторы позволяют задавать только довольно простые последовательности. Значительно более широкие возможности предоставляют функции-генераторы. Они выглядят как функции, в которых вместо `return` используется `yield`.

```
In [17]: def gen():
        yield 0
        yield -1
        yield 4
```

Вызвав такую функцию, мы получим некоторый объект, являющийся итератором.

```
In [18]: it=gen()
        it
```

```
Out[18]: <generator object gen at 0x7fceb407ddb0>
```

Его можно использовать любым обычным образом.

```
In [19]: for x in it:
        print(x)
```

```
0
-1
4
```

Вызвав функцию `gen` снова, мы получим новый итератор, который опять можно использовать.

```
In [20]: it=gen()
        list(it)
```

```
Out[20]: [0, -1, 4]
```

При первом вызове `next` операторы функции выполняются до первого `yield`. Возвращается указанное в нём значение; текущее состояние функции (точка выполнения, значения локальных переменных) запоминается. При следующем вызове `next` выполнение продолжается с того же места до тех пор, пока опять не встретится `yield`. Когда выполнение дойдёт до конца (или до `return`), выдаётся исключение `StopIteration`.

```
In [21]: it=gen()
         next(it)
```

```
Out[21]: 0
```

```
In [22]: next(it)
```

```
Out[22]: -1
```

```
In [23]: next(it)
```

```
Out[23]: 4
```

```
In [24]: next(it)
```

```
-----

StopIteration                                Traceback (most recent call last)

<ipython-input-24-2cdb14c0d4d6> in <module>()
----> 1 next(it)

StopIteration:
```

Много интересных функций для работы с итераторами имеется в модуле `itertools` стандартной библиотеки.

```
In [25]: from itertools import repeat,count,isllice,cycle,chain,accumulate
```

Вызов `repeat(x)` возвращает итератор, повторяющий значение `x` до бесконечности; `repeat(x,n)` повторяет его `n` раз.

```
In [26]: list(repeat('abc',3))
```

```
Out[26]: ['abc', 'abc', 'abc']
```

Бесконечные итераторы могут использоваться для написания циклов, выход из которых производится по `break`; они также полезны как аргументы различных операций над итераторами. Одна из таких операций - `isllice`: `isllice(it,n)` — это итератор, возвращающий первые `n` элементов итератора `it`, а `isllice(it,n,m)` возвращает элементы с `n`-ного (включительно) до `m`-го (не включая его).

```
In [27]: list(isllice([0,1,4,9],2))
```

```
Out[27]: [0, 1]
```

```
In [28]: list(isllice([0,1,4,9],1,3))
```

```
Out[28]: [1, 4]
```

Вызов `count()` возвращает итератор, выдающий бесконечную последовательность 0, 1, 2, ...; `count(n)` - начиная с `n`; `count(n,h)` - с шагом `h`.

```
In [29]: list(islice(count(),10))
```

```
Out[29]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [30]: list(islice(count(4),10))
```

```
Out[30]: [4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

```
In [31]: list(islice(count(4,2),10))
```

```
Out[31]: [4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

Вызов `cycle(it)` возвращает итератор, выдающий элементы `it` по циклу до бесконечности (для этого, разумеется, итератор `it` должен быть конечным; в противном случае мы никогда не доберёмся до конца первого цикла).

```
In [32]: list(islice(cycle('кy'),10))
```

```
Out[32]: ['к', 'y', 'к', 'y', 'к', 'y', 'к', 'y', 'к', 'y']
```

Вызов `chain(i1,i2)` возвращает итератор, выдающий сначала все элементы `i1`, а затем все элементы `i2`. Аргументов может быть и > 2 . Разумеется, если среди аргументов встретится бесконечный итератор, то до его конца мы никогда не доберёмся.

```
In [33]: for i in chain([0,1],[4,9]):  
         print(i)
```

```
0  
1  
4  
9
```

Есть и несколько встроенных функций для работы с итераторами, их не нужно импортировать из `itertools`. Так, `zip` работает следующим образом:

```
In [34]: list(zip([0,1],[4,9]))
```

```
Out[34]: [(0, 4), (1, 9)]
```

Он прекращает работу, когда закончится более короткая последовательность. Аргументов может быть и > 2 .

```
In [35]: list(zip(count(),[1,2,4],'abcdefgh'))
```

```
Out[35]: [(0, 1, 'a'), (1, 2, 'b'), (2, 4, 'c')]
```

Отсюда видно, что `enumerate(x)`, который мы уже обсуждали, эквивалентен `zip(count(),x)`.

Из числовой последовательности x_0, x_1, x_2, \dots можно построить последовательность кумулятивных сумм $s_0 = x_0, s_1 = s_0 + x_1, s_2 = s_1 + x_2, \dots$

```
In [36]: list(islice(accumulate(count(1)),10))
```

```
Out[36]: [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

Вместо сложения можно использовать любую функцию 2 переменных.

```
In [37]: list(islice(accumulate(count(1),lambda x,y:x*y),10))
```

```
Out[37]: [1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Кстати, вместо этого `lambda` выражения мы могли бы использовать функцию `mul`, которую надо импортировать из модуля `operator`. Там есть и `add`, и другие инфиксные операции в виде функций, так что их можно использовать как фактические параметры в вызовах.

Функция `reduce` из модуля `functools` стандартной библиотеки фактически возвращает последний элемент последовательности s_i , то есть $((x_0 + x_1) + x_2) + x_3) \dots$. Вместо сложения может использоваться любая функция 2 переменных.

```
In [38]: from functools import reduce
         from operator import add
         reduce(add,[1,4,9,16])
```

```
Out[38]: 30
```

С помощью итераторов можно делать поразительные вещи. Вот, например, бесконечная последовательность простых чисел (методом решета Эратосфена).

```
In [39]: def primes():
         yield 2
         d={}
         for q in count(3,2):
             p=d.pop(q,None)
             if p is None: # q простое
                 d[q**2]=q
                 yield q
             else:         # q составное
                 x=q+2*p
                 while x in d:
                     x+=2*p
                 d[x]=p
```

```
In [40]: list(islice(primes(),10))
```

```
Out[40]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

1.9 Объектно-ориентированное программирование

Питон является развитым объектно-ориентированным языком. Всё, с чем он работает, является объектами — целые числа, строки, словари, функции и т.д. Каждый объект принадлежит определённому типу (или классу, что одно и то же). Класс тоже является объектом. Классы наследуют друг от друга. Класс `object` является корнем дерева классов — каждый класс наследует от него прямо или через какие-то промежуточные классы.

```
In [1]: object, type(object)
```

```
Out[1]: (object, type)
```

Функция `dir` возвращает список атрибутов класса.

```
In [2]: dir(object)
```

```
Out[2]: ['__class__',
         '__delattr__',
         '__dir__',
         '__doc__',
         '__eq__',
         '__format__',
         '__ge__',
         '__getattr__',
         '__gt__',
         '__hash__',
         '__init__',
         '__init_subclass__',
         '__le__',
         '__lt__',
         '__ne__',
         '__new__',
         '__reduce__',
         '__reduce_ex__',
         '__repr__',
         '__setattr__',
         '__sizeof__',
         '__str__',
         '__subclasshook__']
```

Атрибуты, имена которых начинаются и кончаются двойным подчеркиванием, используются интерпретатором для особых целей. Например, атрибут `__doc__` содержит док-строку.

```
In [3]: object.__doc__
```

```
Out[3]: 'The most base type'
```

```
In [4]: help(object)
```


Help on class object in module builtins:

```
class object
|   The most base type
```

Ниже мы рассмотрим цели некоторых других специальных атрибутов.

Вот простейший класс. Поскольку не указано, от чего он наследует, он наследует от `object`.

```
In [5]: class A:
        pass
```

```
In [6]: A,type(A)
```

```
Out[6]: (__main__.A, type)
```

Создать объект какого-то класса можно, вызвав имя класса как функцию (возможно, с какими-нибудь аргументами). Мы уже это видели: имена классов `int`, `str`, `list` и т.д. создают объекты этих классов.

```
In [7]: o=A()
        o,type(o)
```

```
Out[7]: (<__main__.A at 0x7f5cb30bcbe0>, __main__.A)
```

Узнать, какому классу принадлежит объект, можно при помощи функции `type` или атрибута `__class__`.

```
In [8]: type(o),o.__class__
```

```
Out[8]: (__main__.A, __main__.A)
```

У только что созданного объекта `o` нет атрибутов. Их можно создавать (и удалять) налету.

```
In [9]: o.x=1
        o.y=2
        o.x,o.y
```

```
Out[9]: (1, 2)
```

```
In [10]: o.z
```

```
-----
AttributeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-10-c8c0d478b237> in <module>()
```

```
----> 1 o.z
```

```
AttributeError: 'A' object has no attribute 'z'
```

```
In [11]: del o.y
         o.y
```

```
-----

AttributeError                                Traceback (most recent call last)

<ipython-input-11-68acd6859c06> in <module>()
      1 del o.y
----> 2 o.y
```

```
AttributeError: 'A' object has no attribute 'y'
```

Такой объект похож на словарь, ключами которого являются имена атрибутов: можно узнать значение атрибута, изменить его, добавить новый или удалить старый. Это и неудивительно: для реализации атрибутов объекта используется именно словарь.

```
In [12]: o.__dict__
```

```
Out[12]: {'x': 1}
```

Класс вводит пространство имён. В описании класса мы определяем его атрибуты (атрибуты, являющиеся функциями, называются методами). Потом эти атрибуты можно использовать как `Class.attribute`. Принято, чтобы имена классов начинались с заглавной буквы.

Вот более полный пример класса. В нём есть док-строка, метод `f`, статический атрибут `x` (атрибут класса, а не конкретного объекта) и статический метод `getx` (опять же принадлежащий классу, а не конкретному объекту).

```
In [13]: class S:
         'Простой класс'

         x=1

         def f(self):
             print(self)

         @staticmethod
         def getx():
             return S.x
```

Заклинание тёмной магии, начинающееся с `@`, называется декоратором. Запись

```
@dec
def fun(x):
    ...
```

эквивалентна

```
def fun(x):
    ...
fun=dec(fun)
```

То есть `dec` — это функция, параметр которой — функция, и он возвращает эту функцию, преобразованную некоторым образом. Мы не будем обсуждать, как самим сочинять такие заклинания — за этим обращайтесь в Дурмстранг.

Функция `dir` возвращает список атрибутов класса. Чтобы не смотреть снова на атрибуты, унаследованные от `object`, мы их вычтем.

```
In [14]: set(dir(S))-set(dir(object))
```

```
Out[14]: {'__dict__', '__module__', '__weakref__', 'f', 'getx', 'x'}
```

```
In [15]: dict(S.__dict__)
```

```
Out[15]: {'__dict__': <attribute '__dict__' of 'S' objects>,
          '__doc__': 'Простой класс',
          '__module__': '__main__',
          '__weakref__': <attribute '__weakref__' of 'S' objects>,
          'f': <function __main__.S.f>,
          'getx': <staticmethod at 0x7f5cb28553c8>,
          'x': 1}
```

```
In [16]: S.x
```

```
Out[16]: 1
```

```
In [17]: S.x=2
         S.x
```

```
Out[17]: 2
```

```
In [18]: S.f,S.getx
```

```
Out[18]: (<function __main__.S.f>, <function __main__.S.getx>)
```

```
In [19]: S.getx()
```

```
Out[19]: 2
```

Теперь создадим объект этого класса.

```
In [20]: o=S()
         o,type(o)
```

```
Out[20]: (<__main__.S at 0x7f5cb2855c88>, __main__.S)
```

Метод класса можно вызвать и через объект.

```
In [21]: o.getx()
```

Out[21]: 2

Следующее присваивание создаёт атрибут объекта `o` с именем `x`. Когда мы запрашиваем `o.x`, атрибут `x` ищется сначала в объекте `o`, а если он там не найден — в его классе. В данном случае он найдётся в объекте `o`. На атрибут класса `S.x` это присваивание не влияет.

```
In [22]: o.x=5
         o.x,S.x,o.getx()
```

Out[22]: (5, 2, 2)

Как мы уже обсуждали, можно вызвать метод класса `S.f` с каким-нибудь аргументом, например, `o`.

```
In [23]: S.f(o)

<__main__.S object at 0x7f5cb2855c88>
```

Следующий вызов означает в точности то же самое. Интерпретатор питон фактически преобразует его в предыдущий.

```
In [24]: o.f()

<__main__.S object at 0x7f5cb2855c88>
```

То есть текущий объект передаётся методу в качестве первого аргумента. Этот первый аргумент любого метода принято называть `self`. В принципе, Вы можете назвать его как угодно, но это затруднит понимание Вашего класса читателями, воспитанными в этой традиции.

Отличие метода класса (`@staticmethod`) от метода объекта состоит в том, что такое автоматическое вставление первого аргумента не производится.

`o.f` — это связанный метод; `S.f` связанный с объектом `o`.

```
In [25]: o.f

Out[25]: <bound method S.f of <__main__.S object at 0x7f5cb2855c88>>
```

```
In [26]: g=o.f
         g()

<__main__.S object at 0x7f5cb2855c88>
```

Док-строка доступна как атрибут `__doc__` и используется функцией `help`.

```
In [27]: S.__doc__

Out[27]: 'Простой класс'

In [28]: help(S)
```

Help on class S in module __main__:

```
class S(builtins.object)
|   Простой класс
|
|   Methods defined here:
|
|   f(self)
|
|   -----
|   Static methods defined here:
|
|   getx()
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
|
|   -----
|   Data and other attributes defined here:
|
|   x = 2
```

Классу можно добавить новый атрибут налету (равно как и удалить имеющийся).

```
In [29]: S.y=2
         S.y
```

```
Out[29]: 2
```

Можно добавить и атрибут, являющийся функцией, т.е. метод. Сначала опишем (вне тела класса!) какую-нибудь функцию, а потом добавим её к классу в качестве нового метода.

```
In [30]: def g(self):
         print(self.y)
         S.g=g
         o.g()
```

2

Менять класс налету таким образом — плохая идея. Когда в каком-то месте программы Вы видите, что используется какой-то объект некоторого класса, первое, что Вы сделаете

— это посмотрите определение этого класса. И если текущее его состояние отлично от его определения, это сильно затрудняет понимание программы.

Класс `S`, который мы рассмотрели в качестве примера — отнюдь не пример для подражания. В нормальном объектно-ориентированном подходе объект класса должен создаваться в допустимом (пригодном к использованию) состоянии, со всеми необходимыми атрибутами. В других языках за это отвечает конструктор. В питоне аналогичную роль играет метод инициализации `__init__`. Вот пример такого класса.

```
In [31]: class C:
```

```
    def __init__(self,x):
        self.x=x

    def getx(self):
        return self.x

    def setx(self,x):
        self.x=x
```

Теперь для создания объекта мы должны вызвать `C` с одним аргументом `x` (первый аргумент метода `__init__`, `self`, это свежесозданный объект, в котором ещё ничего нет и который надо инициализировать).

```
In [32]: o=C(1)
         o.getx()
```

```
Out[32]: 1
```

```
In [33]: o.setx(2)
         o.getx()
```

```
Out[33]: 2
```

Этот класс — тоже не пример для подражания. В некоторых объектно-ориентированных языках считается некошерным напрямую читать и писать атрибуты; считается, что вся работа должна производиться через вызов методов. В питоне этот предрассудок не разделяют. Так что писать методы типа `getx` и `setx` абсолютно излишне. Они не добавляют никакой полезной функциональности — всё можно сделать, просто используя атрибут `x`.

```
In [34]: o.x
```

```
Out[34]: 2
```

Любой объектно-ориентированный язык, заслуживающий такого названия, поддерживает наследование. Класс `C2` наследует от `C`. Его объекты являются вполне законными для класса `C` (имеют атрибут `x`), но в добавок к этому имеют ещё и атрибут `y`. Метод `__init__` теперь должен иметь 2 параметра `x` и `y` (не считая обязательного `self`). К методам `getx` и `setx`, унаследованным от `C`, добавляются методы `gety` и `sety`.

Чтобы инициализировать атрибут `x`, который был в родительском классе, мы могли бы, конечно, скопировать код из метода `__init__` класса `C`. В данном случае он столь прост, что

это не преступление. Но, вообще говоря, копировать куски кода из одного места в другое категорически не рекомендуется. Допустим, в скопированном куске найден и исправлен баг. А в копии он остался. Поэтому для инициализации нового объекта, рассматриваемого как объект родительского класса `C`, нам следует вызвать метод `__init__` класса `C`, а после этого доавить инициализацию атрибута `y`, специфичного для дочернего класса `C2`. Первую часть задачи можно выполнить, вызвав `C.__init__(self,x)` (мы ведь только что написали строчку `class`, в которой указали, что класс-предок называется `C`). Но есть более универсальный метод, не требующий второй раз писать имя родительского класса. Функция `super()` возвращает текущий объект `self`, рассматриваемый как объект родительского класса `C`. Поэтому мы можем написать `super().__init__(x)`.

Конечно, не только `__init__`, но и другие методы дочернего класса могут захотеть вызвать методы родительского класса. Для этого используется либо вызов через имя родительского класса, либо `super()`.

```
In [35]: class C2(C):
```

```
    def __init__(self,x,y):
        super().__init__(x)
        self.y=y

    def gety(self):
        return self.y

    def sety(self,y):
        self.y=y
```

```
In [36]: o=C2(1,2)
         o.getx(),o.gety()
```

```
Out[36]: (1, 2)
```

`o` является объектом класса `C2`, а также его родительского класса `C` (и, конечно, класса `object`), но не является объектом класса `S`.

```
In [37]: isinstance(o,C2),isinstance(o,C),isinstance(o,object),isinstance(o,S)
```

```
Out[37]: (True, True, True, False)
```

`C2` является подклассом (потомком) себя, класса `C` и `object`, но не является подклассом `S`.

```
In [38]: issubclass(C2,C2),issubclass(C2,C),issubclass(C2,object),issubclass(C2,S)
```

```
Out[38]: (True, True, True, False)
```

Эти функции используются редко. В питоне придерживаются принципа утиной типизации: *если объект ходит, как утка, плавает, как утка, и крикает, как утка, значит, он утка*. Пусть у нас есть класс `Утка` с методами `иди`, `плыви` и `крякни`. Конечно, можно создать подкласс `Кряква`, наследующий эти методы и что-то в них переопределяющий. Но можно написать класс `Кряква` с нуля, без всякой генетической связи с классом `Утка`, и реализовать эти методы. Тогда

в любую программу, ожидающую получить объект класса `Утка` (и общающуюся с ним при помощи методов `иди`, `плыви` и `крякни`), можно вместо этого подставить объект класса `Кряква`, и программа будет по-прежнему работать. А функции `isinstance` и `issubclass` нарушают принцип утиной типизации.

Класс может наследовать от нескольких классов. Мы не будем обсуждать множественное наследование, оно используется редко. Атрибут `__bases__` даёт кортеж родительских классов.

```
In [39]: C2.__bases__
```

```
Out[39]: (__main__.C,)
```

```
In [40]: C.__bases__
```

```
Out[40]: (object,)
```

```
In [41]: object.__bases__
```

```
Out[41]: ()
```

```
In [42]: set(dir(C))-set(dir(object))
```

```
Out[42]: {'__dict__', '__module__', '__weakref__', 'getx', 'setx'}
```

```
In [43]: set(dir(C2))-set(dir(object))
```

```
Out[43]: {'__dict__', '__module__', '__weakref__', 'getx', 'gety', 'setx', 'sety'}
```

```
In [44]: set(dir(C2))-set(dir(C))
```

```
Out[44]: {'gety', 'sety'}
```

```
In [45]: help(C2)
```

```
Help on class C2 in module __main__:
```

```
class C2(C)
|   Method resolution order:
|       C2
|       C
|       builtins.object
|
|   Methods defined here:
|
|   __init__(self, x, y)
|       Initialize self.  See help(type(self)) for accurate signature.
|
|   gety(self)
|
|   sety(self, y)
|
```



```

| -----
| Methods inherited from C:
|
|   getx(self)
|
|   setx(self, x)
|
| -----
| Data descriptors inherited from C:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)

```

В питоне все методы являются, в терминах других языков, виртуальными. Пусть у нас есть класс `A`; метод `get` вызывает метод `str`.

```
In [46]: class A:
```

```

    def __init__(self,x):
        self.x=x

    def str(self):
        return str(self.x)

    def get(self):
        print(self.str())
        return self.x

```

Класс `B` наследует от него и переопределяет метод `str`.

```
In [47]: class B(A):
```

```

    def str(self):
        return 'The value of x is '+super().str()

```

Создадим объект класса `A` и вызовем метод `get`. Он вызывает `self.str()`; `str` ищется (и находится) в классе `A`.

```
In [48]: oa=A(1)
         oa.get()
```

```
1
```

```
Out[48]: 1
```

Теперь создадим объект класса В и вызовем метод `get`. Он ищется в В, не находится, потом ищется и находится в А. Этот метод `A.get(ob)` вызывает `self.str()`, где `self` — это `ob`. Поэтому метод `str` ищется в классе В, находится и вызывается. То есть метод родительского класса вызывает переопределённый метод дочернего класса.

```
In [49]: ob=B(1)
         ob.get()
```

The value of x is 1

```
Out[49]: 1
```

Напишем класс 2-мерных векторов, определяющий некоторые специальные методы для того, чтобы к его объектам можно было применять встроенные операции и функции языка питон (в тех случаях, когда это имеет смысл).

```
In [50]: from math import sqrt
```

```
In [51]: class Vec2:
         '2-dimensional vectors'

         def __init__(self,x=0,y=0):
             self.x=x
             self.y=y

         def __repr__(self):
             return 'Vec2({}, {})'.format(self.x,self.y)

         def __str__(self):
             return '({},{})'.format(self.x,self.y)

         def __bool__(self):
             return self.x!=0 or self.y!=0

         def __eq__(self,other):
             return self.x==other.x and self.y==other.y

         def __abs__(self):
             return sqrt(self.x**2+self.y**2)

         def __neg__(self):
             return Vec2(-self.x,-self.y)

         def __add__(self,other):
             return Vec2(self.x+other.x,self.y+other.y)

         def __sub__(self,other):
             return Vec2(self.x-other.x,self.y-other.y)
```

```

def __iadd__(self, other):
    self.x+=other.x
    self.y+=other.y
    return self

def __isub__(self, other):
    self.x-=other.x
    self.y-=other.y
    return self

def __mul__(self, other):
    return Vec2(self.x*other, self.y*other)

def __rmul__(self, other):
    return Vec2(self.x*other, self.y*other)

def __imul__(self, other):
    self.x*=other
    self.y*=other
    return self

def __truediv__(self, other):
    return Vec2(self.x/other, self.y/other)

def __itruediv__(self, other):
    self.x/=other
    self.y/=other
    return self

```

Создадим вектор. Когда в командной строке питона написано выражение, его значение печатается при помощи метода `__repr__`. Он старается напечатать объект в таком виде, чтобы эту строку можно было вставить в исходный текст программы и воссоздать этот объект. (Для объектов некоторых классов это невозможно, тогда `__repr__` печатает некоторую информацию в угловых скобках `<...>`).

```
In [52]: u=Vec2(1,2)
         u
```

```
Out[52]: Vec2(1,2)
```

Метод `__str__` печатает объект в виде, наиболее простом для восприятия человека (не обязательно машинно-читаемом). Функция `print` использует этот метод.

```
In [53]: print(u)
```

```
(1,2)
```

Это выражение автоматически преобразуется в следующий вызов.

```
In [54]: u*2
```

```
Out[54]: Vec2(2,4)
```

```
In [55]: u.__mul__(2)
```

```
Out[55]: Vec2(2,4)
```

А это выражение — в следующий.

```
In [56]: 3*u, u.__rmul__(3)
```

```
Out[56]: (Vec2(3,6), Vec2(3,6))
```

Такой оператор преобразуется в вызов `u.__imul__(2)`.

```
In [57]: u*=2
         u
```

```
Out[57]: Vec2(2,4)
```

Другие арифметические операторы работают аналогично.

```
In [58]: v=Vec2(-1,2)
         2*u+3*v
```

```
Out[58]: Vec2(1,14)
```

Унарный минус преобразуется в `__neg__`.

```
In [59]: -v, v.__neg__()
```

```
Out[59]: (Vec2(1,-2), Vec2(1,-2))
```

Вызов встроенной функции `abs` — в метод `__abs__`.

```
In [60]: abs(u), u.__abs__()
```

```
Out[60]: (4.47213595499958, 4.47213595499958)
```

```
In [61]: u+=v
         u
```

```
Out[61]: Vec2(1,6)
```

Питон позволяет переопределять то, что происходит при чтении и записи атрибута (а также при его удалении). Эту тёмную магию мы изучать не будем, за одним исключением. Можно определить пару методов, один из которых будет вызываться при чтении некоторого “атрибута”, а другой при его записи. Такой “атрибут”, которого на самом деле нет, называется свойством. Пользователь класса будет спокойно читать и писать этот “атрибут”, не подозревая, что на самом деле для этого вызываются какие-то методы.

В питоне нет приватных атрибутов (в том числе приватных методов). По традиции, атрибуты (включая методы), имена которых начинаются с `_`, считаются приватными. Технически ничто не мешает пользователю класса обращаться к таким “приватным” атрибутам. Но автор класса может в любой момент изменить детали реализации, включая “приватные” атрибуты. Используя их код пользователя при этом сломается. Сам дурак.

В этом классе есть свойство `x`. Его чтение и запись приводят к вызову пары методов, которые читают и пишут “приватный” атрибут `_x`, а также выполняют некоторый код. Свойство создаётся при помощи декораторов. В принципе свойство может быть и чисто синтетическим (без соответствующего “приватного” атрибута) — его “чтение” возвращает результат некоторого вычисления, исходящего из реальных атрибутов, а “запись” меняет значения таких реальных атрибутов.

```
In [62]: class D:
```

```
    def __init__(self,x):
        self._x=x
```

```
    @property
    def x(self):
        print('getting x')
        return self._x
```

```
    @x.setter
    def x(self,x):
        print('setting x')
        self._x=x
```

```
In [63]: o=D('a')
         o.x
```

```
getting x
```

```
Out[63]: 'a'
```

```
In [64]: o.x='b'
```

```
setting x
```

```
In [65]: o.x
```

```
getting x
```

```
Out[65]: 'b'
```

Я использовал свойство, когда писал Монте-Карловское моделирование модели Изинга. У изинговской решётки было свойство — температура, которую можно было читать и писать. Но соответствующего атрибута не было. Был атрибут $x = \exp(-J/T)$, где J — энергия взаимодействия.

Свойства полезны также для обёртки GUI библиотек. Например, окно имеет свойство — заголовок. Чтение или изменение заголовка требует вызова соответствующих функций из низкоуровневой библиотеки (на C или C++). Но на питоне гораздо приятнее написать

```
w.title='Моё окно'
```

1.10 Исключения

Всякие недопустимые операции типа деления на 0 или открытия несуществующего файла приводят к возбуждению исключений. Интерпретатор питон печатает подробную и понятную информацию об исключении. Если это интерактивный интерпретатор, то сессия продолжается; если это программа, то её выполнение прекращается. В питоне отладчик приходится использовать гораздо реже, чем в более низкоуровневых языках, потому что эти сообщения интерпретатора позволяют сразу понять, где и что неверно. Впрочем, иногда приходится использовать и отладчик. Допустим, из сообщения об ошибке Вы поняли, что некоторая функция вызвана со строковым аргументом, а Вы про него думали, что он число. Тогда приходится искать — какая сволочь испортила мою переменную?

In [1]: 1/0

```
-----

ZeroDivisionError                                Traceback (most recent call last)

<ipython-input-1-05c9758a9c21> in <module>()
----> 1 1/0

ZeroDivisionError: division by zero
```

Исключения можно отлавливать, и в случае, если они произошли, выполнять какой-нибудь корректирующий код.

```
In [2]: try:
        x=0
        x=1/x
    except ZeroDivisionError:
        x=5
```

In [3]: x

Out[3]: 5

```
In [4]: try:
        s='xyzzzy'
        f=open(s)
    except IOError:
        print('cannot open '+s)
```

cannot open xyzzzy

Исключения — это объекты. Класс `Exception` является корнем дерева классов исключений. Этот объект можно поймать и исследовать.

```
In [5]: try:
        x=1/0
    except Exception as err:
        print(type(err))
        print(err)
        print(repr(err))
        print(err.args)

<class 'ZeroDivisionError'>
division by zero
ZeroDivisionError('division by zero',)
('division by zero',)
```

Если в Вашем коде возникла недопустимая ситуация, нужно возбудить исключение оператором `raise`.

```
In [6]: raise NameError('Hi there')
```

```
-----

NameError                                Traceback (most recent call last)

<ipython-input-6-d36a3cf2a944> in <module>()
----> 1 raise NameError('Hi there')

NameError: Hi there
```

Вот более полезный пример.

```
In [7]: def f(x):
        if x==0:
            raise ValueError('x should not be 0')
        return x
```

```
In [8]: try:
        x=f(1)
        x=f(0)
    except ValueError as err:
        print(repr(err))
```

```
ValueError('x should not be 0',)
```

```
In [9]: x
```

```
Out[9]: 1
```


Естественно, можно определять свои классы исключений, наследуя от `Exception` или от какого-нибудь его потомка, подходящего по смыслу. Именно так и нужно делать, чтобы Ваши исключения не путались с системными.

```
In [10]: class MyError(Exception):

        def __init__(self,value):
            self.value=value

        def __str__(self):
            return str(self.value)
```

```
In [11]: def f(x):
        if x<0:
            raise MyError(x)
        else:
            return x
```

```
In [12]: try:
        x=f(2)
        x=f(-2)
    except MyError as err:
        print(err)
```

-2

```
In [13]: x
```

```
Out[13]: 2
```

1.11 Модули

Модуль — это просто файл типа `.py`, содержащий последовательность операторов питона. Его можно использовать двумя способами: либо запустить как программу, либо импортировать в другой модуль, чтобы сделать доступными определённые там функции и переменные. При импортировании все операторы модуля выполняются от начала до конца, включая определения функций и классов и присваивания переменным. Впрочем, при повторном импортировании модуль не выполняется. Если Вы его изменили и хотите импортировать изменённую версию, нужно приложить специальные усилия.

```
In [1]: import math
        math,type(math)
```

```
Out[1]: (<module 'math' from '/usr/lib64/python3.6/lib-dynload/math.cpython-36m-x86_64-linux-gnu
        module>)
```

Модуль имеет своё пространство имён. Оператор `import math` вводит *объект типа модуль* `math` в текущее пространство имён. Имена, определённые в модуле, при этом в текущем пространстве имён не появляются — их нужно использовать как `math.что_то`. Функция `dir` возвращает список имён в модуле (как и в классе или объекте).

```
In [2]: dir(math)
```

```
Out[2]: ['__doc__',
        '__file__',
        '__loader__',
        '__name__',
        '__package__',
        '__spec__',
        'acos',
        'acosh',
        'asin',
        'asinh',
        'atan',
        'atan2',
        'atanh',
        'ceil',
        'copysign',
        'cos',
        'cosh',
        'degrees',
        'e',
        'erf',
        'erfc',
        'exp',
        'expm1',
        'fabs',
        'factorial',
        'floor',
```

```
'fmod',
'frexp',
'fsum',
'gamma',
'gcd',
'hypot',
'inf',
'isclose',
'isfinite',
'isinf',
'isnan',
'ldexp',
'lgamma',
'log',
'log10',
'log1p',
'log2',
'modf',
'nan',
'pi',
'pow',
'radians',
'sin',
'sinh',
'sqrt',
'tan',
'tanh',
'tau',
'trunc']
```

```
In [3]: math.__doc__
```

```
Out[3]: 'This module is always available. It provides access to the\nmathematical functions def
```

```
In [4]: math.pi, math.exp
```

```
Out[4]: (3.141592653589793, <function math.exp>)
```

```
In [5]: math.exp(math.pi)
```

```
Out[5]: 23.140692632779267
```

Встроенные функции, классы и т.д. языка питон живут в модуле **builtins**.

```
In [6]: import builtins
dir(builtins)
```

```
Out[6]: ['ArithmeticError',
'AssertionError',
```

```
'AttributeError',
'BaseException',
'BlockingIOError',
'BrokenPipeError',
'BufferError',
'BytesWarning',
'ChildProcessError',
'ConnectionAbortedError',
'ConnectionError',
'ConnectionRefusedError',
'ConnectionResetError',
'DeprecationWarning',
'EOFError',
'Ellipsis',
'EnvironmentError',
'Exception',
'False',
'FileExistsError',
'FileNotFoundError',
'FloatingPointError',
'FutureWarning',
'GeneratorExit',
'IOError',
'ImportError',
'ImportWarning',
'IndentationError',
'IndexError',
'InterruptedError',
'IsADirectoryError',
'KeyError',
'KeyboardInterrupt',
'LookupError',
'MemoryError',
'ModuleNotFoundError',
'NameError',
'None',
'NotADirectoryError',
'NotImplemented',
'NotImplementedError',
'OSError',
'OverflowError',
'PendingDeprecationWarning',
'PermissionError',
'ProcessLookupError',
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
```

```
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
```

```
'divmod',
'dreload',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'range',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
```

```
'sum',  
'super',  
'tuple',  
'type',  
'vars',  
'zip']
```

Если Вам лень полностью писать имя модуля перед каждым использованием функции из него, можно использовать `as` и задать ему краткое имя.

```
In [7]: import random as r  
r
```

```
Out[7]: <module 'random' from '/usr/lib64/python3.6/random.py'>
```

```
In [8]: dir(r)
```

```
Out[8]: ['BPF',  
         'LOG4',  
         'NV_MAGICCONST',  
         'RECIP_BPF',  
         'Random',  
         'SG_MAGICCONST',  
         'SystemRandom',  
         'TWOPI',  
         '_BuiltinMethodType',  
         '_MethodType',  
         '_Sequence',  
         '_Set',  
         '__all__',  
         '__builtins__',  
         '__cached__',  
         '__doc__',  
         '__file__',  
         '__loader__',  
         '__name__',  
         '__package__',  
         '__spec__',  
         '_acos',  
         '_bisect',  
         '_ceil',  
         '_cos',  
         '_e',  
         '_exp',  
         '_inst',  
         '_itertools',  
         '_log',  
         '_pi',  
         '_random',
```

```

'_sha512',
'_sin',
'_sqrt',
'_test',
'_test_generator',
'_urandom',
'_warn',
'betavariate',
'choice',
'choices',
'expovariate',
'gammavariate',
'gauss',
'getrandbits',
'getstate',
'lognormvariate',
'normalvariate',
'paretovariate',
'randint',
'random',
'randrange',
'sample',
'seed',
'setstate',
'shuffle',
'triangular',
'uniform',
'vonmisesvariate',
'weibullvariate']

```

```
In [9]: [r.random() for i in range(10)]
```

```

Out[9]: [0.09637772172557402,
0.89551664168298,
0.9426823013506337,
0.4094328234976712,
0.7232184697800689,
0.8586383884595843,
0.2778551445357017,
0.4321903116808209,
0.30183017632492515,
0.3057854562362986]

```

Такая форма оператора `import` вводит перечисленные имена (функции, переменные, классы) из модуля в текущее пространство имён. Мне она нравится — использовать импортированные таким образом объекты удобно, не надо писать перед каждым имя модуля.

```
In [10]: from sys import path
```


Переменная `path` — это список имён директорий, в которых оператор `import` ищет модули. В начале в него входит `'` — директория, в которой находится текущая программа (или текущая директория в случае интерактивной сессии); директории, перечисленные в переменной окружения `PYTHONPATH` (если такая переменная есть); и стандартные директории для данной версии питона. Но это обычный список, его можно менять стандартными языковыми средствами. Например, ревнители безопасности считают, что опасно включать текущую директорию в `path` — если пользователю в его директорию кто-нибудь подсунет зловредную версию `math.py`, а программа пользователя выполнит `import math`, то этот модуль выполнится, и может, скажем, удалить все файлы этого пользователя. Такие ревнители могут сделать `path=path[1:]`.

```
In [11]: path
```

```
Out[11]: ['',
          '/usr/lib64/python36.zip',
          '/usr/lib64/python3.6',
          '/usr/lib64/python3.6/lib-dynload',
          '/usr/lib64/python3.6/site-packages',
          '/usr/lib64/python3.6/site-packages/IPython/extensions',
          '/home/grozin/.ipython']
```

```
In [12]: path.append('/home/grozin/python')
         path
```

```
Out[12]: ['',
          '/usr/lib64/python36.zip',
          '/usr/lib64/python3.6',
          '/usr/lib64/python3.6/lib-dynload',
          '/usr/lib64/python3.6/site-packages',
          '/usr/lib64/python3.6/site-packages/IPython/extensions',
          '/home/grozin/.ipython',
          '/home/grozin/python']
```

Если Вам лень писать каждый раз длинное имя функции из модуля, можно дать ему короткий псевдоним.

```
In [13]: from math import factorial as f
```

```
In [14]: f(100)
```

```
Out[14]: 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414
```

Для самых ленивых есть оператор `from ... import *`, который импортирует в текущее пространство имён все имена, определённые в модуле. Обычно это плохая идея — Вы засоряете текущее пространство имён, и даже не знаете, чем. Такую форму импорта разумно использовать, когда Вы импортируете свой модуль, про который Вы всё знаете. Ну или в интерактивной сессии, когда Вы хотите попробовать всякие функции из какого-нибудь модуля. Но не в программе, которая пишется всерьёз и надолго.

Например, в текущей директории есть файл `fac.py`. Мы работаем в `ipython`, который предоставляет всякие удобства для интерактивной работы. Например, можно выполнить `shell` команду, если в начале строки поставить `!` (только не пробуйте этого делать в обычном интерпретаторе питон). Так что легко распечатать этот файл. В нём определена одна функция `fac`.

```

In [15]: !cat fac.py

#!/usr/bin/env python3
'В этом модуле определена функция fac'

def fac(n):
    'calculate factorial of n'
    assert type(n) is int and n >= 0
    r = 1
    for i in range(2, n + 1):
        r *= i
    return r

if __name__ == '__main__':
    from sys import argv, exit
    if len(argv) != 2:
        print('usage: ./fac.py n')
        exit(1)
    print(fac(int(argv[1])))

In [16]: from fac import *
         fac(10)

```

```
Out[16]: 3628800
```

Файл `fac.py` показывает типичное устройство любого файла на питоне. Первая строка позволяет запустить такой файл, если у него установлен бит, позволяющий исполнять его текущему пользователю. Почему не просто `#!/usr/bin/python3`? Потому что на некоторых машинах питон может быть в `/usr/local/bin` или ещё где-то; стандартная `unix`-утилита `env` (предположительно) всегда живёт в `/usr/bin`. Она позволяет установить какие-нибудь переменные окружения, а затем, если есть аргумент — имя программы, запускает эту программу в этом модифицированном окружении; если такого аргумента нет, просто печатает это окружение. Так что, вызвав просто `env`, Вы получите список всех текущих переменных окружения с их значениями. В данном случае вызывается `/usr/bin/env python3`, то есть никакие изменения окружения не производятся, и `env` вызывает `python3`, расположенный где угодно в `$PATH`. Почему `python3`? `python` может быть симлинком либо на `python2`, либо на `python3`; в свою очередь, `python3` может быть симлинком, скажем, на `python3.6`. Если наша программа предназначена для питона 3, то в первой строке лучше указывать `python3`, иначе на некоторых машинах могут возникнуть неприятные сюрпризы.

Дальше следует док-строка модуля. Потом определения всех функций, классов и т.д. Заключительная часть файла выполняется, если он запущен как программа, а не импортируется куда-то. В этой части обычно пишут какие-нибудь простые тесты определённых в файле функций. В данном случае используется `sys.argv` — список строк-аргументов командной строки. `argv[0]` — это имя программы, нас интересует переданный ей параметр, `argv[1]`.

```

In [17]: import fac
         fac.__doc__

```

```
Out[17]: 'В этом модуле определена функция fac'
```

```
In [18]: help(fac)
```

```
Help on module fac:
```

```
NAME
```

```
    fac - В этом модуле определена функция fac
```

```
FUNCTIONS
```

```
    fac(n)
        calculate factorial of n
```

```
FILE
```

```
    /home/grozin/python/book/fac.py
```

Функция `dir` без аргумента возвращает список имён в текущем пространстве имён. Многие имена в этом списке определены `ipython`-ом; в сессии с обычным интерпретатором питон их бы не было.

```
In [19]: dir()
```

```
Out[19]: ['In',
          'Out',
          '_',
          '_1',
          '_11',
          '_12',
          '_14',
          '_16',
          '_17',
          '_2',
          '_3',
          '_4',
          '_5',
          '_6',
          '_7',
          '_8',
          '_9',
          '__',
          '___',
          '__builtin__',
          '__builtins__',
          '__doc__',
          '__loader__',
          '__name__']
```

```

'__package__',
'__spec__',
'_dh',
'_exit_code',
'_i',
'_i1',
'_i10',
'_i11',
'_i12',
'_i13',
'_i14',
'_i15',
'_i16',
'_i17',
'_i18',
'_i19',
'_i2',
'_i3',
'_i4',
'_i5',
'_i6',
'_i7',
'_i8',
'_i9',
'_ih',
'_ii',
'_iii',
'_oh',
'_sh',
'builtins',
'exit',
'f',
'fac',
'get_ipython',
'math',
'path',
'quit',
'r']

```

В локальном пространстве имён этой функции два имени.

```

In [20]: def f(x):
          y=0
          print(dir())

In [21]: f(0)

['x', 'y']

```

В каждом модуле есть строковая переменная `__name__`, она содержит имя модуля. Главная программа (или интерактивная сессия) тоже является модулем, его имя `__main__`. Этим и объясняется вид оператора `if`, который стоит в конце файла `fac.py`.

```
In [22]: __name__
```

```
Out[22]: '__main__'
```

```
In [23]: r.__name__
```

```
Out[23]: 'random'
```

Модули не обязательно должны размещаться непосредственно в какой-нибудь директории из `sys.path`; они могут находиться в поддиректории. Например, в текущей директории (включённой в `path`) есть поддиректория `d1`, в ней поддиректория `d2`.

```
In [24]: !ls d1
```

```
__pycache__  d2  m1.py
```

```
In [25]: !ls d1/d2
```

```
__pycache__  m2.py
```

Мы можем импортировать модули `m1` и `m2` так.

```
In [26]: import d1.m1
         d1.m1.f1()
```

```
Out[26]: 1
```

```
In [27]: import d1.d2.m2
         d1.d2.m2.f2()
```

```
Out[27]: 2
```

Такое поддерево директорий с модулями можно превратить в пакет, который с точки зрения пользователя выглядит как единый модуль. Для этого нужно добавить файл `__init__.py`. Вот другое поддерево с теми же файлами `m1.py` и `m2.py`.

```
In [28]: !ls p1
```

```
__init__.py  __pycache__  m1.py  p2
```

```
In [29]: !ls p1/p2
```

```
__pycache__  m2.py
```

Только добавлен файл `__init__.py`.

```
In [30]: !cat p1/__init__.py
```

```
'Пакет, экспортирующий f1 из модуля m1 и f2 из модуля m2'  
from p1.m1 import f1  
from p1.p2.m2 import f2
```

Теперь мы можем импортировать этот пакет.

```
In [31]: import p1
```

Питон находит в `sys.path` директорию `p1`, содержащую `__init__.py`, и интерпретирует её как пакет. При импорте выполняется этот файл `__init__.py`, инициализирующий пакет. Все функции, переменные и т.д., определённые в этом файле (непосредственно или через импорт), становятся символами этого пакета. `__init__.py` может включать не все функции из модулей этого дерева директорий (и даже не все модули); символы, не определённые в `__init__.py`, недоступны после импорта пакета (конечно, пользователь всегда может импортировать любой модуль напрямую и получить доступ ко всем его символам).

```
In [32]: p1.__doc__
```

```
Out[32]: 'Пакет, экспортирующий f1 из модуля m1 и f2 из модуля m2'
```

```
In [33]: p1.f1(),p1.f2()
```

```
Out[33]: (1, 2)
```

1.12 Ввод-вывод, файлы, директории

Откроем текстовый файл на чтение (когда второй аргумент не указан, файл открывается именно на чтение).

```
In [1]: f=open('text.txt')
        f,type(f)
```

```
Out[1]: (<_io.TextIOWrapper name='text.txt' mode='r' encoding='UTF-8'>,
        _io.TextIOWrapper)
```

Получился объект `f` одного из файловых типов. Что с ним можно делать? Можно его использовать в `for` цикле, каждый раз будет возвращаться очередная строка файла (включая `'\n'` в конце; в конце последней строки текстового файла `'\n'` может и не быть).

```
In [2]: for s in f:
        print(s)
```

```
abcd
```

```
efgh
```

```
ijkl
```

Теперь файл нужно закрыть.

```
In [3]: f.close()
```

Такой стиль работы с файлом (`f=open(...)`; работа с `f`; `f.close()`) на самом деле не рекомендуется. Гораздо правильнее использовать оператор `with`. Он гарантирует, что файл будет закрыт как в том случае, когда исполнение тела `with` нормально дошло до конца, так и тогда, когда при этом произошло исключение, и мы покинули тело `with` аварийно.

В операторе `with` может использоваться любой объект класса, реализующего методы `__enter__` и `__exit__`. Обычно это объект-файл, возвращаемый функцией `open`.

```
In [4]: with open('text.txt') as f:
        for s in f:
            print(s[:-1])
```

```
abcd
```

```
efgh
```

```
ijkl
```

Метод `f.read(n)` читает `n` символов (когда файл близится к концу и прочитать именно `n` символов уже невозможно, читает меньше; в самый последний раз он читает 0 символов и возвращает `''`). Прочитаем файл по 1 символу.

```
In [5]: with open('text.txt') as f:
        while True:
            c=f.read(1)
            if c=='':
                break
            else:
                print(c)
```

```
a
b
c
d
```

```
e
f
g
h
```

```
i
j
k
l
```

Вызов `f.read()` без аргумента читает файл целиком (что не очень разумно, если в нём много гигабайт).

```
In [6]: with open('text.txt') as f:
        s=f.read()
        s
```

```
Out[6]: 'abcd\nefgh\nijkl\n'
```

`f.readline()` читает очередную строку (хотя проще использовать `for s in f:`).

```
In [7]: with open('text.txt') as f:
        while True:
            s=f.readline()
            if s=='':
                break
            else:
                print(s)
```

```
abcd
```



```
efgh
```

```
ijkl
```

Метод `f.readlines()` возвращает список строк (опять же его лучше не применять для очень больших файлов).

```
In [8]: with open('text.txt') as f:
        l=f.readlines()
        1
```

```
Out[8]: ['abcd\n', 'efgh\n', 'ijkl\n']
```

Теперь посмотрим, чем же оператор `with` лучше, чем пара `open — close`.

```
In [9]: def a(name):
        global f
        f=open(name)
        s=f.readline()
        n=1/0
        f.close()
        return s
```

```
In [10]: a('text.txt')
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)

<ipython-input-10-d62372657d26> in <module>()
----> 1 a('text.txt')

<ipython-input-9-7f445757684d> in a(name)
      3     f=open(name)
      4     s=f.readline()
----> 5     n=1/0
      6     f.close()
      7     return s
```

```
ZeroDivisionError: division by zero
```

```
In [11]: f.closed
```

```
Out[11]: False
```

```
In [12]: f.close()
```

Произошло исключение, мы покинули функцию до строчки `close`, и файл не закрылся.

```
In [13]: def a(name):
          global f
          with open(name) as f:
              s=f.readline()
              n=1/0
          return s
```

```
In [14]: a('text.txt')
```

```
-----

ZeroDivisionError                                Traceback (most recent call last)

<ipython-input-14-d62372657d26> in <module>()
----> 1 a('text.txt')

<ipython-input-13-cabd1416e96c> in a(name)
      3     with open(name) as f:
      4         s=f.readline()
----> 5         n=1/0
      6     return s

ZeroDivisionError: division by zero
```

```
In [15]: f.closed
```

```
Out[15]: True
```

Теперь всё в порядке.

Чтобы открыть файл на запись, нужно включить второй аргумент `'w'`.

```
In [16]: f=open('newtext.txt','w')
```

```
In [17]: f.write('aaa\n')
```

```
Out[17]: 4
```

```
In [18]: f.write('bbb\n')
```

```
Out[18]: 4
```

```
In [19]: f.write('ccc\n')
```

```
Out[19]: 4
```

```
In [20]: f.close()
```

Метод `write` возвращает число записанных символов.
Опять же, лучше использовать `with`.

```
In [21]: with open('newtext.txt', 'w') as f:
          f.write('aaa\n')
          f.write('bbb\n')
          f.write('ccc\n')
```

```
In [22]: !cat newtext.txt
```

```
aaa
bbb
ccc
```

Эта функция копирует старый текстовый файл в новый. Если строки нужно как-нибудь обработать, в последней строчке вместо `line` будет стоять что-нибудь вроде `f(line)`.

```
In [23]: def copy(old_name, new_name):
          with open(old_name) as old, open(new_name, 'w') as new:
              for line in old:
                  new.write(line)
```

```
In [24]: copy('text.txt', 'newtext.txt')
```

```
In [25]: !cat newtext.txt
```

```
abcd
efgh
ijkl
```

В интерактивной сессии (или в программе, запущенной с командной строки) можно попросить пользователя что-нибудь ввести. Аргумент функции `input` — это приглашение для ввода (prompt). Можно использовать просто `input()`, тогда приглашения не будет. Но это неудобно, т.к. в этом случае трудно заметить, что программа чего-то ждёт.

```
In [26]: s=input('Введите целое число ')
```

```
Введите целое число 123
```

```
In [27]: s
```

```
Out[27]: '123'
```

```
In [28]: n=int(s)
         n
```

```
Out[28]: 123
```

Питон — интерпретатор, поэтому он может во время выполнения программы интерпретировать строки как куски исходного текста на языке питон. Так, функция `eval` интерпретирует строку как выражение и вычисляет его (в текущем контексте — подставляя текущие значения переменных).

```
In [29]: s=input('Введите выражение ')
```

Введите выражение n+1

```
In [30]: s
```

```
Out[30]: 'n+1'
```

```
In [31]: eval(s)
```

```
Out[31]: 124
```

А функция `exec` интерпретирует строку как оператор и выполняет его. Оператор может менять значения переменных в текущем пространстве имён.

```
In [32]: s=input('Введите оператор ')
```

Введите оператор x=0

```
In [33]: s
```

```
Out[33]: 'x=0'
```

```
In [34]: exec(s)
         x
```

```
Out[34]: 0
```

Строка `s` может быть результатом длинного и сложного вычисления. Но лучше таких фокусов не делать, так как программа фактически становится саомодифицирующей. Такие программы очень сложно отлаживать.

Для работы с путями к файлам и директориям в стандартной библиотеке существует модуль `pathlib`. Объект класса `Path` представляет собой путь к файлу или директории.

```
In [35]: from pathlib import Path
```

`Path()` возвращает текущую директорию.

```
In [36]: p=Path()
         p
```

```
Out [36]: PosixPath('.')
```

Очень полезный метод `resolve` приводит путь к каноническому виду.

```
In [37]: p.resolve()
```

```
Out [37]: PosixPath('/home/grozin/python/book')
```

Путь может быть записан в совершенно идиотском виде; `resolve` его исправит.

```
In [38]: p=Path('.././book')
         p=p.resolve()
         p
```

```
Out [38]: PosixPath('/home/grozin/python/book')
```

Статический метод `cwd` возвращает текущую директорию (current working directory).

```
In [39]: Path.cwd()
```

```
Out [39]: PosixPath('/home/grozin/python/book')
```

Если `p` — путь к директории, то можно посмотреть все файлы в ней.

```
In [40]: for f in p.iterdir():
         print(f)

/home/grozin/python/book/b102_strings.ipynb
/home/grozin/python/book/tex
/home/grozin/python/book/.ipynb_checkpoints
/home/grozin/python/book/b103_lists.ipynb
/home/grozin/python/book/b109_exceptions.ipynb
/home/grozin/python/book/fac.py
/home/grozin/python/book/d1
/home/grozin/python/book/newtext.txt
/home/grozin/python/book/b108_oop.ipynb
/home/grozin/python/book/b106_dictionaries.ipynb
/home/grozin/python/book/b101_numbers.ipynb
/home/grozin/python/book/text.txt
/home/grozin/python/book/b104_tuples.ipynb
/home/grozin/python/book/__pycache__
/home/grozin/python/book/p1
/home/grozin/python/book/b107_functions.ipynb
/home/grozin/python/book/b110_modules.ipynb
/home/grozin/python/book/b105_sets.ipynb
/home/grozin/python/book/b111_input_output.ipynb
```

Если `p` — путь к директории, то `p/'fname'` — путь к файлу `fname` в ней (он, конечно, тоже может быть директорией).

```
In [41]: p2=p/'b101_numbers.ipynb'  
p2
```

```
Out[41]: PosixPath('/home/grozin/python/book/b101_numbers.ipynb')
```

Существует ли такой файл?

```
In [42]: p2.exists()
```

```
Out[42]: True
```

Является ли он симлинком, директорией, файлом?

```
In [43]: p2.is_symlink(),p2.is_dir(),p2.is_file()
```

```
Out[43]: (False, False, True)
```

Части пути p2.

```
In [44]: p2.parts
```

```
Out[44]: ('/', 'home', 'grozin', 'python', 'book', 'b101_numbers.ipynb')
```

Родитель — директория, в которой находится этот файл.

```
In [45]: p2.parent,p2.parent.parent
```

```
Out[45]: (PosixPath('/home/grozin/python/book'), PosixPath('/home/grozin/python'))
```

Имя файла, его основа и суффикс.

```
In [46]: p2.name,p2.stem,p2.suffix
```

```
Out[46]: ('b101_numbers.ipynb', 'b101_numbers', '.ipynb')
```

Метод `stat` возвращает всякую ценную информацию о файле.

```
In [47]: s=p2.stat()  
s
```

```
Out[47]: os.stat_result(st_mode=33188, st_ino=2097706, st_dev=2052, st_nlink=1, st_uid=1000, st_
```

Например, его размер в байтах.

```
In [48]: s.st_size
```

```
Out[48]: 17223
```

Я написал полезную утилиту для поиска одинаковых файлов. Ей передаётся произвольное число аргументов — директорий и файлов. Она рекурсивно обходит директории, находит размер всех файлов (симлинки игнорируются) и строит словарь, сопоставляющий каждому размеру список файлов, имеющих такой размер. Это простой этап, не требующий чтения (возможно больших) файлов. После этого файлы из тех списков, длина которых > 1 , сравниваются внешней программой `diff` (что, конечно, требует их чтения).

В питоне можно работать с переменными окружения как с обычным словарём.

```
In [49]: from os import environ
```

```
In [50]: environ['PATH']
```

```
Out[50]: '/usr/lib/python-exec/python3.6:/home/grozin/bin:/home/grozin/reduce-3783/bin:/usr/loca
```

```
In [51]: environ['ABCD']
```

```
-----

KeyError                                Traceback (most recent call last)

<ipython-input-51-71e016be80d8> in <module>()
----> 1 environ['ABCD']

/usr/lib64/python3.6/os.py in __getitem__(self, key)
    667         except KeyError:
    668             # raise KeyError with the original key value
--> 669             raise KeyError(key) from None
    670         return self.decodevalue(value)
    671

KeyError: 'ABCD'
```

```
In [52]: environ['ABCD']='abcd'
```

```
In [53]: environ['ABCD']
```

```
Out[53]: 'abcd'
```

Мы не просто добавили пару ключ-значение в словарь, а действительно добавили новую переменную к текущему окружению. Если теперь вызвать из питона какую-нибудь внешнюю программу, то она эту переменную увидит. Эта переменная исчезнет, когда закончится выполнение текущей программы на питоне (или интерактивная сессия).

Глава 2

Пакеты для научных вычислений

2.1 numpy

Пакет `numpy` предоставляет n -мерные однородные массивы (все элементы одного типа); в них нельзя вставить или удалить элемент в произвольном месте. В `numpy` реализовано много операций над массивами в целом. Если задачу можно решить, произведя некоторую последовательность операций над массивами, то это будет столь же эффективно, как в `C` или `matlab` — львиная доля времени тратится в библиотечных функциях, написанных на `C`.

2.1.1 Одномерные массивы

```
In [1]: from numpy import (array,zeros,ones,arange,linspace,logspace,
                        float64,int64,sin,cos,pi,exp,log,sqrt,abs,
                        nan,inf,any,all,sort,hstack,vstack,hsplit,
                        delete,insert,append,eye,fromfunction,
                        trace,diag,average,std,outer,meshgrid)
```

Можно преобразовать список в массив.

```
In [2]: a=array([0,2,1])
        a,type(a)
```

```
Out[2]: (array([0, 2, 1]), numpy.ndarray)
```

`print` печатает массивы в удобной форме.

```
In [3]: print(a)
```

```
[0 2 1]
```

Класс `ndarray` имеет много методов.

```
In [4]: set(dir(a))-set(dir(object))
```

```
Out[4]: {'T',
        '__abs__',
        '__add__',
        '__and__',
        '__array__',
        '__array_finalize__',
        '__array_interface__',
        '__array_prepare__',
        '__array_priority__',
        '__array_struct__',
        '__array_wrap__',
        '__bool__',
        '__complex__',
        '__contains__',
        '__copy__',
        '__deepcopy__',
```

```
'__delitem__',  
'__divmod__',  
'__float__',  
'__floordiv__',  
'__getitem__',  
'__iadd__',  
'__iand__',  
'__ifloordiv__',  
'__ilshift__',  
'__imatmul__',  
'__imod__',  
'__imul__',  
'__index__',  
'__int__',  
'__invert__',  
'__ior__',  
'__ipow__',  
'__irshift__',  
'__isub__',  
'__iter__',  
'__itrueidiv__',  
'__ixor__',  
'__len__',  
'__lshift__',  
'__matmul__',  
'__mod__',  
'__mul__',  
'__neg__',  
'__or__',  
'__pos__',  
'__pow__',  
'__radd__',  
'__rand__',  
'__rdivmod__',  
'__rfloordiv__',  
'__rlshift__',  
'__rmatmul__',  
'__rmod__',  
'__rmul__',  
'__ror__',  
'__rpow__',  
'__rrshift__',  
'__rshift__',  
'__rsub__',  
'__rtrueidiv__',  
'__rxor__',  
'__setitem__',  
'__setstate__',
```

```
'__sub__',  
'__truediv__',  
'__xor__',  
'all',  
'any',  
'argmax',  
'argmin',  
'argpartition',  
'argsort',  
'astype',  
'base',  
'byteswap',  
'choose',  
'clip',  
'compress',  
'conj',  
'conjugate',  
'copy',  
'ctypes',  
'cumprod',  
'cumsum',  
'data',  
'diagonal',  
'dot',  
'dtype',  
'dump',  
'dumps',  
'fill',  
'flags',  
'flat',  
'flatten',  
'getfield',  
'imag',  
'item',  
'itemset',  
'itemsizes',  
'max',  
'mean',  
'min',  
'nbytes',  
'ndim',  
'newbyteorder',  
'nonzero',  
'partition',  
'prod',  
'ptp',  
'put',  
'ravel',
```

```
'real',
'repeat',
'reshape',
'resize',
'round',
'searchsorted',
'setfield',
'setflags',
'shape',
'size',
'sort',
'squeeze',
'std',
'strides',
'sum',
'swapaxes',
'take',
'tobytes',
'tofile',
'tolist',
'tostring',
'trace',
'transpose',
'var',
'view']}
```

Наш массив одномерный.

```
In [5]: a.ndim
```

```
Out[5]: 1
```

В n -мерном случае возвращается кортеж размеров по каждой координате.

```
In [6]: a.shape
```

```
Out[6]: (3,)
```

`size` — это полное число элементов в массиве; `len` — размер по первой координате (в 1-мерном случае это то же самое).

```
In [7]: len(a), a.size
```

```
Out[7]: (3, 3)
```

`numpy` предоставляет несколько типов для целых (`int16`, `int32`, `int64`) и чисел с плавающей точкой (`float32`, `float64`).

```
In [8]: a.dtype, a.dtype.name, a.itemsize
```

```
Out[8]: (dtype('int64'), 'int64', 8)
```

Индексировать массив можно обычным образом.

```
In [9]: a[1]
```

```
Out[9]: 2
```

Массивы — изменяемые объекты.

```
In [10]: a[1]=3
         print(a)
```

```
[0 3 1]
```

Массивы, разумеется, можно использовать в `for` циклах. Но при этом теряется главное преимущество `numpy` — быстроедействие. Всегда, когда это возможно, лучше использовать операции над массивами как едиными целыми.

```
In [11]: for i in a:
         print(i)
```

```
0
3
1
```

Массив чисел с плавающей точкой.

```
In [12]: b=array([0.,2,1])
         b.dtype
```

```
Out[12]: dtype('float64')
```

Точно такой же массив.

```
In [13]: c=array([0,2,1],dtype=float64)
         print(c)
```

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

```
In [ ]: array([1.2,1.5,1.8],dtype=int64)
```

Массив, значения которого вычисляются функцией. Функции передаётся массив. Так что в ней можно использовать только такие операции, которые применимы к массивам.

```
In [14]: def f(i):
         print(i)
         return i**2
         a=fromfunction(f,(5,),dtype=int64)
         print(a)
```

```
[0 1 2 3 4]
[ 0  1  4  9 16]
```

```
In [15]: a=fromfunction(f,(5,),dtype=float64)
         print(a)
```

```
[ 0.  1.  2.  3.  4.]
[ 0.  1.  4.  9. 16.]
```

Массивы, заполненные нулями или единицами. Часто лучше сначала создать такой массив, а потом присваивать значения его элементам.

```
In [16]: a=zeros(3)
         print(a)
```

```
[ 0.  0.  0.]
```

```
In [17]: b=ones(3,dtype=int64)
         print(b)
```

```
[1 1 1]
```

Функция `arange` подобна `range`. Аргументы могут быть с плавающей точкой. Следует избегать ситуаций, когда `(-)/` — целое число, потому что в этом случае включение последнего элемента зависит от ошибок округления. Лучше, чтобы конец диапазона был где-то посередине шага.

```
In [18]: a=arange(0,9,2)
         print(a)
```

```
[0 2 4 6 8]
```

```
In [19]: b=arange(0.,9,2)
         print(b)
```

```
[ 0.  2.  4.  6.  8.]
```

Последовательности чисел с постоянным шагом можно также создавать функцией `linspace`. Начало и конец диапазона включаются; последний аргумент — число точек.

```
In [20]: a=linspace(0,8,5)
         print(a)
```

```
[ 0.  2.  4.  6.  8.]
```

Последовательность чисел с постоянным шагом по логарифмической шкале от 10^0 до 10^1 .

```
In [21]: b=logspace(0,1,5)
         print(b)
```

```
[ 1.          1.77827941  3.16227766  5.62341325 10.          ]
```

Массив случайных чисел.

```
In [22]: from numpy.random import random,normal
         print(random(5))
```

```
[ 0.63038745  0.24031792  0.75969506  0.98191274  0.9023238 ]
```

Случайные числа с нормальным (гауссовым) распределением (среднее 0, среднеквадратичное отклонение 1).

```
In [24]: print(normal(size=5))
```

```
[-0.06409544  0.26656068 -1.83718422 -1.01312915  0.0445343 ]
```

2.1.2 Операции над одномерными массивами

Арифметические операции проводятся поэлементно.

```
In [25]: print(a+b)
```

```
[ 1.          3.77827941  7.16227766 11.62341325 18.          ]
```

```
In [26]: print(a-b)
```

```
[-1.          0.22172059  0.83772234  0.37658675 -2.          ]
```

```
In [27]: print(a*b)
```

```
[ 0.          3.55655882 12.64911064 33.74047951 80.          ]
```

Скалярное произведение

```
In [28]: a@b
```

```
Out[28]: 129.9461489721723
```

```
In [29]: print(a/b)
```



```
[ 0.          1.12468265  1.26491106  1.06696765  0.8          ]
```

```
In [30]: print(a**2)
```

```
[ 0.   4.  16.  36.  64.]
```

Когда операнды разных типов, они приводятся к большему типу.

```
In [31]: i=ones(5,dtype=int64)
         print(a+i)
```

```
[ 1.  3.  5.  7.  9.]
```

`numpy` содержит элементарные функции, которые тоже применяются к массивам поэлементно. Они называются универсальными функциями (`ufunc`).

```
In [32]: sin,type(sin)
```

```
Out[32]: (<ufunc 'sin'>, numpy.ufunc)
```

```
In [33]: print(sin(a))
```

```
[ 0.          0.90929743 -0.7568025  -0.2794155   0.98935825]
```

Один из операндов может быть скаляром, а не массивом.

```
In [34]: print(a+1)
```

```
[ 1.  3.  5.  7.  9.]
```

```
In [35]: print(2*a)
```

```
[ 0.   4.   8.  12.  16.]
```

Сравнения дают булевы массивы.

```
In [36]: print(a>b)
```

```
[False  True  True  True False]
```

```
In [37]: print(a==b)
```

```
[False False False False False]
```

```
In [38]: c=a>5
         print(c)
```

```
[False False False  True  True]
```

Кванторы “существует” и “для всех”.

```
In [39]: any(c),all(c)
```

```
Out[39]: (True, False)
```

Модификация на месте.

```
In [40]: a+=1
         print(a)
```

```
[ 1.  3.  5.  7.  9.]
```

```
In [41]: b*=2
         print(b)
```

```
[ 2.          3.55655882  6.32455532 11.2468265  20.          ]
```

```
In [42]: b/=a
         print(b)
```

```
[ 2.          1.18551961  1.26491106  1.6066895  2.22222222]
```

Так делать можно.

```
In [43]: a+=i
```

А так нельзя.

```
In [44]: i+=a
```

```
-----
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-44-300abee39dd1> in <module>()
```

```
----> 1 i+=a
```

```
TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with cas
```

При выполнении операций над массивами деление на 0 не возбуждает исключения, а даёт значения `np.nan` или `np.inf`.

```
In [45]: print(array([0.0,0.0,1.0,-1.0])/array([1.0,0.0,0.0,0.0]))
```

```
[ 0.  nan  inf -inf]
```

```
/usr/lib64/python3.6/site-packages/ipykernel/__main__.py:1: RuntimeWarning: divide by zero encountered in divide
  if __name__ == '__main__':
/usr/lib64/python3.6/site-packages/ipykernel/__main__.py:1: RuntimeWarning: invalid value encountered in divide
  if __name__ == '__main__':
```

```
In [46]: nan+1,inf+1,inf*0,1./inf,inf/inf
```

```
Out[46]: (nan, inf, nan, 0.0, nan)
```

```
In [47]: nan==nan,inf==inf
```

```
Out[47]: (False, True)
```

Сумма и произведение всех элементов массива; максимальный и минимальный элемент; среднее и среднеквадратичное отклонение.

```
In [48]: b.sum(),b.prod(),b.max(),b.min(),b.mean(),b.std()
```

```
Out[48]: (8.2793423935260435,
          10.708241812210389,
          2.2222222222222223,
          1.1855196066926152,
          1.6558684787052087,
          0.40390033426607452)
```

```
In [49]: x=normal(size=1000)
          x.mean(),x.std()
```

```
Out[49]: (-0.048736395274562645, 0.98622825985036244)
```

Функция `sort` возвращает отсортированную копию, метод `sort` сортирует на месте.

```
In [50]: print(sort(b))
          print(b)
```

```
[ 1.18551961  1.26491106  1.6066895   2.          2.22222222]
[ 2.          1.18551961  1.26491106  1.6066895   2.22222222]
```

```
In [51]: b.sort()
          print(b)
```

```
[ 1.18551961  1.26491106  1.6066895  2.          2.22222222]
```

Объединение массивов.

```
In [52]: a=hstack((a,b))
         print(a)
```

```
[ 2.          4.          6.          8.          10.         1.18551961
 1.26491106  1.6066895  2.          2.22222222]
```

Расщепление массива в позициях 3 и 6.

```
In [53]: hsplit(a,[3,6])
```

```
Out[53]: [array([ 2.,  4.,  6.]),
          array([ 8.          , 10.          ,  1.18551961]),
          array([ 1.26491106,  1.6066895 ,  2.          ,  2.22222222])]
```

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

```
In [54]: a=delete(a,[5,7])
         print(a)
```

```
[ 2.          4.          6.          8.          10.         1.26491106
 2.          2.22222222]
```

```
In [55]: a=insert(a,2,[0,0])
         print(a)
```

```
[ 2.          4.          0.          0.          6.          8.          10.
 1.26491106  2.          2.22222222]
```

```
In [56]: a=append(a,[1,2,3])
         print(a)
```

```
[ 2.          4.          0.          0.          6.          8.          10.
 1.26491106  2.          2.22222222  1.          2.          3.          ]
```

Есть несколько способов индексации массива. Вот обычный индекс.

```
In [57]: a=linspace(0,1,11)
         print(a)
```

```
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

```
In [58]: b=a[2]
         print(b)
```

0.2

Диапазон индексов. Создаётся новый заголовок массива, указывающий на те же данные. Изменения, сделанные через такой массив, видны и в исходном массиве.

```
In [59]: b=a[2:6]
         print(b)
```

[0.2 0.3 0.4 0.5]

```
In [60]: b[0]=-0.2
         print(b)
```

[-0.2 0.3 0.4 0.5]

```
In [61]: print(a)
```

[0. 0.1 -0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]

Диапазон с шагом 2.

```
In [62]: b=a[1:10:2]
         print(b)
```

[0.1 0.3 0.5 0.7 0.9]

```
In [63]: b[0]=-0.1
         print(a)
```

[0. -0.1 -0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]

Массив в обратном порядке.

```
In [64]: b=a[len(a):0:-1]
         print(b)
```

[1. 0.9 0.8 0.7 0.6 0.5 0.4 0.3 -0.2 -0.1]

Подмассиву можно присвоить значение — массив правильного размера или скаляр.

```
In [65]: a[1:10:3]=0
         print(a)
```

```
[ 0.  0. -0.2  0.3  0.  0.5  0.6  0.  0.8  0.9  1. ]
```

Тут опять создаётся только новый заголовок, указывающий на те же данные.

```
In [66]: b=a[:]
         b[1]=0.1
         print(a)
```

```
[ 0.  0.1 -0.2  0.3  0.  0.5  0.6  0.  0.8  0.9  1. ]
```

Чтобы скопировать и данные массива, нужно использовать метод `copy`.

```
In [67]: b=a.copy()
         b[2]=0
         print(b)
         print(a)
```

```
[ 0.  0.1  0.  0.3  0.  0.5  0.6  0.  0.8  0.9  1. ]
[ 0.  0.1 -0.2  0.3  0.  0.5  0.6  0.  0.8  0.9  1. ]
```

Можно задать список индексов.

```
In [68]: print(a[[2,3,5]])
```

```
[-0.2  0.3  0.5]
```

```
In [69]: print(a[array([2,3,5])])
```

```
[-0.2  0.3  0.5]
```

Можно задать булев массив той же величины.

```
In [70]: b=a>0
         print(b)
```

```
[False  True False  True False  True  True False  True  True  True]
```

```
In [71]: print(a[b])
```

```
[ 0.1  0.3  0.5  0.6  0.8  0.9  1. ]
```

2.1.3 2-мерные массивы

```
In [72]: a=array([[0.0,1.0],[-1.0,0.0]])  
         print(a)
```

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

```
In [73]: a.ndim
```

```
Out[73]: 2
```

```
In [74]: a.shape
```

```
Out[74]: (2, 2)
```

```
In [75]: len(a),a.size
```

```
Out[75]: (2, 4)
```

```
In [76]: a[1,0]
```

```
Out[76]: -1.0
```

Атрибуту `shape` можно присвоить новое значение — кортеж размеров по всем координатам. Получится новый заголовок массива; его данные не изменятся.

```
In [77]: b=linspace(0,3,4)  
         print(b)
```

```
[ 0.  1.  2.  3.]
```

```
In [78]: b.shape
```

```
Out[78]: (4,)
```

```
In [79]: b.shape=2,2  
         print(b)
```

```
[[ 0.  1.]  
 [ 2.  3.]]
```

Поэлементное и матричное умножение.

```
In [80]: print(a*b)
```

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

```
In [81]: print(a@b)
```

```
[[ 2.  3.]
 [ 0. -1.]]
```

```
In [82]: print(b@a)
```

```
[[ -1.  0.]
 [ -3.  2.]]
```

Умножение матрицы на вектор.

```
In [83]: v=array([1,-1],dtype=float64)
         print(b@v)
```

```
[-1. -1.]
```

```
In [84]: print(v@b)
```

```
[-2. -2.]
```

Внешнее произведение $a_{ij} = u_i v_j$

```
In [85]: u=linspace(1,2,2)
         v=linspace(2,4,3)
         print(u)
         print(v)
```

```
[ 1.  2.]
 [ 2.  3.  4.]
```

```
In [86]: a=outer(u,v)
         print(a)
```

```
[[ 2.  3.  4.]
 [ 4.  6.  8.]]
```

Двумерные массивы, зависящие только от одного индекса: $x_{ij} = u_j$, $y_{ij} = v_i$

```
In [87]: x,y=meshgrid(u,v)
         print(x)
         print(y)
```



```
[[ 1.  2.]
 [ 1.  2.]
 [ 1.  2.]]
[[ 2.  2.]
 [ 3.  3.]
 [ 4.  4.]]
```

Единичная матрица.

```
In [88]: I=eye(4)
         print(I)
```

```
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

Метод `reshape` делает то же самое, что присваивание атрибуту `shape`.

```
In [89]: print(I.reshape(16))
```

```
[ 1.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  1.]
```

```
In [90]: print(I.reshape(2,8))
```

```
[[ 1.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  1.]]
```

Строка.

```
In [91]: print(I[1])
```

```
[ 0.  1.  0.  0.]
```

Цикл по строкам.

```
In [92]: for row in I:
         print(row)
```

```
[ 1.  0.  0.  0.]
[ 0.  1.  0.  0.]
[ 0.  0.  1.  0.]
[ 0.  0.  0.  1.]
```

Столбец.

```
In [93]: print(I[:,2])
[ 0.  0.  1.  0.]
```

Подматрица.

```
In [94]: print(I[0:2,1:3])
[[ 0.  0.]
 [ 1.  0.]]
```

Можно построить двумерный массив из функции.

```
In [95]: def f(i,j):
          print(i)
          print(j)
          return 10*i+j
          print(fromfunction(f,(4,4),dtype=int64))

[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]]
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]]
```

Транспонированная матрица.

```
In [96]: print(b.T)
[[ 0.  2.]
 [ 1.  3.]]
```

Соединение матриц по горизонтали и по вертикали.

```
In [97]: a=array([[0,1],[2,3]])
          b=array([[4,5,6],[7,8,9]])
          c=array([[4,5],[6,7],[8,9]])
          print(a)
          print(b)
          print(c)
```

```
[[0 1]
 [2 3]]
[[4 5 6]
 [7 8 9]]
[[4 5]
 [6 7]
 [8 9]]
```

```
In [98]: print(hstack((a,b)))
```

```
[[0 1 4 5 6]
 [2 3 7 8 9]]
```

```
In [99]: print(vstack((a,c)))
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

Сумма всех элементов; суммы столбцов; суммы строк.

```
In [100]: print(b.sum())
          print(b.sum(0))
          print(b.sum(1))
```

```
39
[11 13 15]
[15 24]
```

Аналогично работают `prod`, `max`, `min` и т.д.

```
In [101]: print(b.max(0))
          print(b.min(1))
```

```
[7 8 9]
[4 7]
```

След — сумма диагональных элементов.

```
In [102]: trace(a)
```

```
Out[102]: 3
```

2.1.4 Линейная алгебра

```
In [103]: from numpy.linalg import det,inv,solve,eig
          det(a)
```

```
Out[103]: -2.0
```

Обратная матрица.

```
In [104]: a1=inv(a)
          print(a1)
```

```
[[ -1.5  0.5]
 [  1.   0. ]]
```

```
In [105]: print(a@a1)
          print(a1@a)
```

```
[[ 1.  0.]
 [ 0.  1.]]
[[ 1.  0.]
 [ 0.  1.]]
```

Решение линейной системы $au = v$.

```
In [106]: v=array([0,1],dtype=float64)
          print(a1@v)
```

```
[ 0.5  0. ]
```

```
In [107]: u=solve(a,v)
          print(u)
```

```
[ 0.5  0. ]
```

Проверим.

```
In [108]: print(a@u-v)
```

```
[ 0.  0.]
```

Собственные значения и собственные векторы: $au_i = \lambda_i u_i$. λ — одномерный массив собственных значений λ_i , столбцы матрицы u — собственные векторы u_i .

```
In [109]: l,u=eig(a)
          print(l)
```

```
[-0.56155281  3.56155281]
```

```
In [110]: print(u)
```

```
[[-0.87192821 -0.27032301]
 [ 0.48963374 -0.96276969]]
```

Проверим.

```
In [111]: for i in range(2):
           print(a@u[:,i]-l[i]*u[:,i])
```

```
[ 0.00000000e+00  1.66533454e-16]
[ 1.11022302e-16  0.00000000e+00]
```

Функция `diag` от одномерного массива строит диагональную матрицу; от квадратной матрицы — возвращает одномерный массив её диагональных элементов.

```
In [112]: L=diag(l)
           print(L)
           print(diag(L))
```

```
[[-0.56155281  0.          ]
 [ 0.          3.56155281]]
[-0.56155281  3.56155281]
```

Все уравнения $au_i = \lambda_i u_i$ можно собрать в одно матричное уравнение $au = u\Lambda$, где Λ — диагональная матрица с собственными значениями λ_i по диагонали.

```
In [113]: print(a@u-u@L)
```

```
[[ 0.00000000e+00  1.11022302e-16]
 [ 1.66533454e-16  0.00000000e+00]]
```

Поэтому $u^{-1}au = \Lambda$.

```
In [114]: print(inv(u)@a@u)
```

```
[[-5.61552813e-01  0.00000000e+00]
 [-2.22044605e-16  3.56155281e+00]]
```

Найдём теперь левые собственные векторы $v_i a = \lambda_i v_i$ (собственные значения λ_i те же самые).

```
In [115]: l,v=eig(a.T)
           print(l)
           print(v)

[-0.56155281  3.56155281]
[[-0.96276969 -0.48963374]
 [ 0.27032301 -0.87192821]]
```

Собственные векторы нормированы на 1.

```
In [116]: print(u.T@u)
           print(v.T@v)

[[ 1.          -0.23570226]
 [-0.23570226  1.          ]]
[[ 1.          0.23570226]
 [ 0.23570226  1.          ]]
```

Левые и правые собственные векторы, соответствующие разным собственным значениям, ортогональны, потому что $v_i a u_j = \lambda_i v_i u_j = \lambda_j v_i u_j$.

```
In [117]: print(v.T@u)

[[ 9.71825316e-01  0.00000000e+00]
 [-5.55111512e-17  9.71825316e-01]]
```

2.1.5 Преобразование Фурье

```
In [118]: a=linspace(0,1,11)
           print(a)

[ 0.  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

```
In [119]: from numpy.fft import fft,ifft
           b=fft(a)
           print(b)

[ 5.50+0.j          -0.55+1.87312798j -0.55+0.85581671j -0.55+0.47657771j
 -0.55+0.25117658j -0.55+0.07907806j -0.55-0.07907806j -0.55-0.25117658j
 -0.55-0.47657771j -0.55-0.85581671j -0.55-1.87312798j]
```

Обратное преобразование Фурье.

```
In [120]: print(ifft(b))

[ 1.61486985e-15+0.j    1.00000000e-01+0.j    2.00000000e-01+0.j
 3.00000000e-01+0.j    4.00000000e-01+0.j    5.00000000e-01+0.j
 6.00000000e-01+0.j    7.00000000e-01+0.j    8.00000000e-01+0.j
 9.00000000e-01+0.j    1.00000000e+00+0.j]
```

2.1.6 Интегрирование

```
In [121]: from scipy.integrate import quad,odeint
          from scipy.special import erf
```

```
In [122]: def f(x):
          return exp(-x**2)
```

Адаптивное численное интегрирование (может быть до бесконечности). `err` — оценка ошибки.

```
In [123]: res,err=quad(f,0,inf)
          print(sqrt(pi)/2,res,err)
```

```
0.886226925453 0.8862269254527579 7.101318390472462e-09
```

```
In [124]: res,err=quad(f,0,1)
          print(sqrt(pi)/2*erf(1),res,err)
```

```
0.746824132812 0.7468241328124271 8.291413475940725e-15
```

2.1.7 Дифференциальные уравнения

Уравнение осциллятора с затуханием $\ddot{x} + 2a\dot{x} + x = 0$. Перепишем его в виде системы уравнений первого порядка для x , $v = \dot{x}$:

$$\frac{d}{dt} \begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -2av - x \end{pmatrix}$$

Решим эту систему численно при $a = 0.2$ с начальным условием $\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

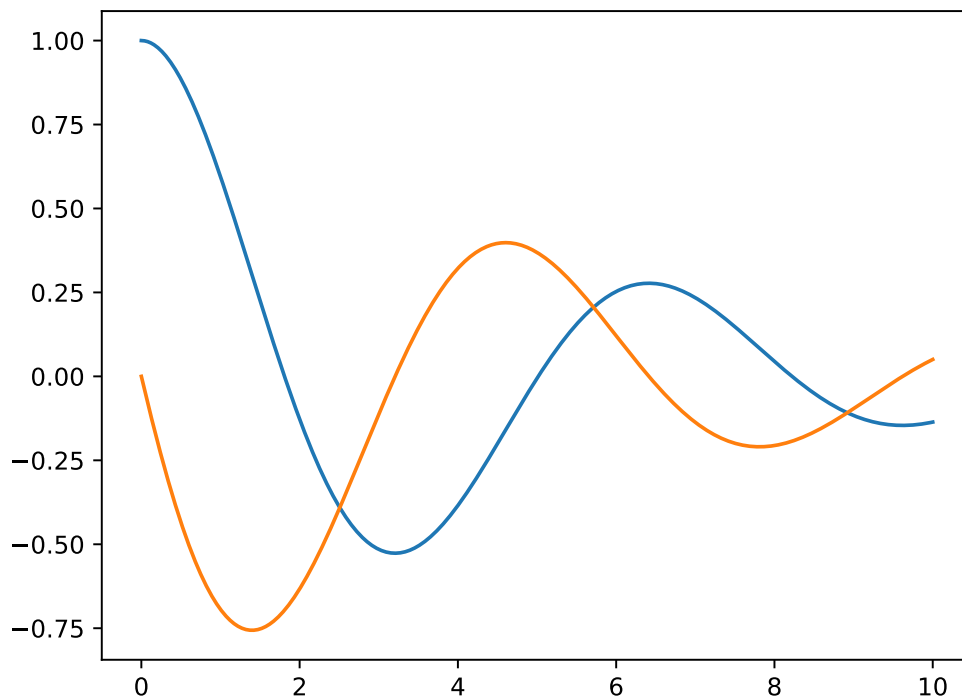
```
In [125]: a=0.2
          def f(x,t):
              global a
              return [x[1],-x[0]-2*a*x[1]]
```

```
In [126]: t=linspace(0,10,1000)
          x=odeint(f,[1,0],t)
```

Графики координаты и скорости.

```
In [127]: from matplotlib.pyplot import plot
          %matplotlib inline
          plot(t,x)
```

```
Out[127]: [<matplotlib.lines.Line2D at 0x7f2fade9a6d8>,
          <matplotlib.lines.Line2D at 0x7f2fade9a8d0>]
```



Точное решение для координаты.

```
In [128]: b=sqrt(1-a**2)
          x0=exp(-a*t)*(cos(b*t)+a/b*sin(b*t))
```

Максимальное отличие численного решения от точного.

```
In [129]: abs(x[:,0]-x0).max()
```

```
Out[129]: 7.4104573116740013e-08
```


2.2 matplotlib

Есть несколько пакетов для построения графиков. Один из наиболее популярных — `matplotlib`. Если в `jupyter notebook` выполнить специальную `ipython` команду `%matplotlib inline`, то графики будут строиться в том же окне браузера. Есть другие варианты, в которых графики показываются в отдельных окнах. Это удобно для трёхмерных графиков — тогда их можно вертеть мышкой (в случае `inline` графиков это невозможно). Графики можно также сохранять в файлы, как в векторных форматах (`eps`, `pdf`, `svg`), так и в растровых (`png`, `jpg`; конечно, растровые форматы годятся только для размещения графиков на web-страницах). `matplotlib` позволяет строить двумерные графики практически всех нужных типов, с достаточно гибкой регулировкой их параметров; он также поддерживает основные типы трёхмерных графиков, но для серьёзной трёхмерной визуализации данных лучше пользоваться более мощными специализированными системами.

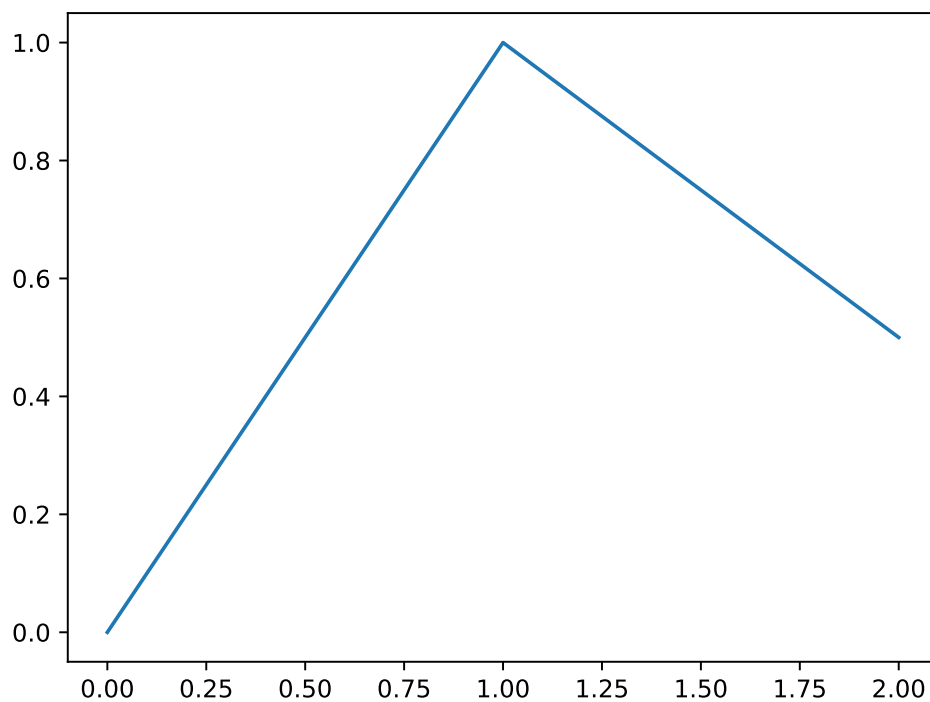
```
In [1]: from matplotlib.pyplot import (axes,axis,title,legend,figure,
                                         xlabel,ylabel,xticks,yticks,
                                         xscale,yscale,text,grid,
                                         plot,scatter,errorbar,hist,polar,
                                         contour,contourf,colorbar,clabel,
                                         imshow)

from mpl_toolkits.mplot3d import Axes3D
from numpy import (linspace,logspace,zeros,ones,outer,meshgrid,
                   pi,sin,cos,sqrt,exp)
from numpy.random import normal
%matplotlib inline
```

Список y координат; x координаты образуют последовательность 0, 1, 2, ...

```
In [2]: plot([0,1,0.5])
```

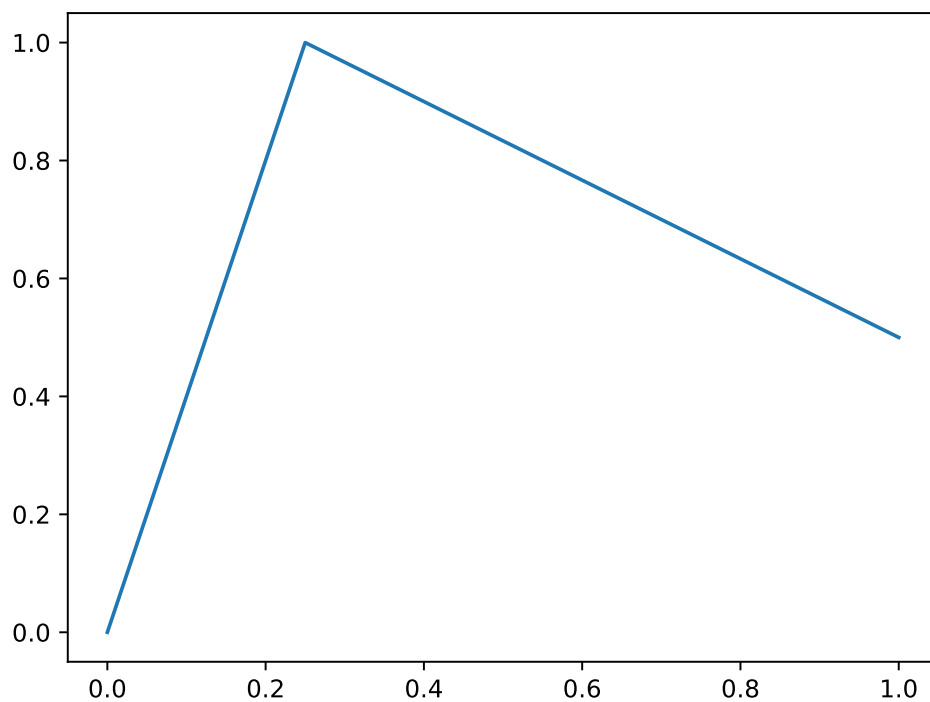
```
Out[2]: [<matplotlib.lines.Line2D at 0x7fcbd1a559e8>]
```



Списки x и y координат точек. Точки соединяются прямыми, т.е. строится ломаная линия.

```
In [3]: plot([0,0.25,1],[0,1,0.5])
```

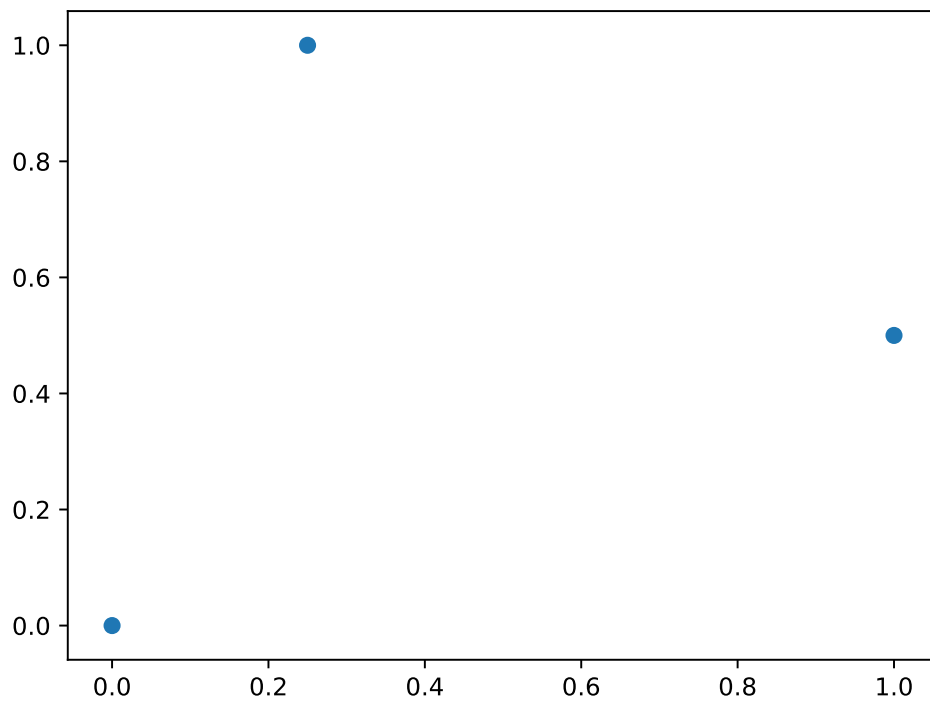
```
Out [3]: [<matplotlib.lines.Line2D at 0x7fcbd1939f98>]
```



`scatter` просто рисует точки, не соединяя их линиями.

```
In [4]: scatter([0,0.25,1],[0,1,0.5])
```

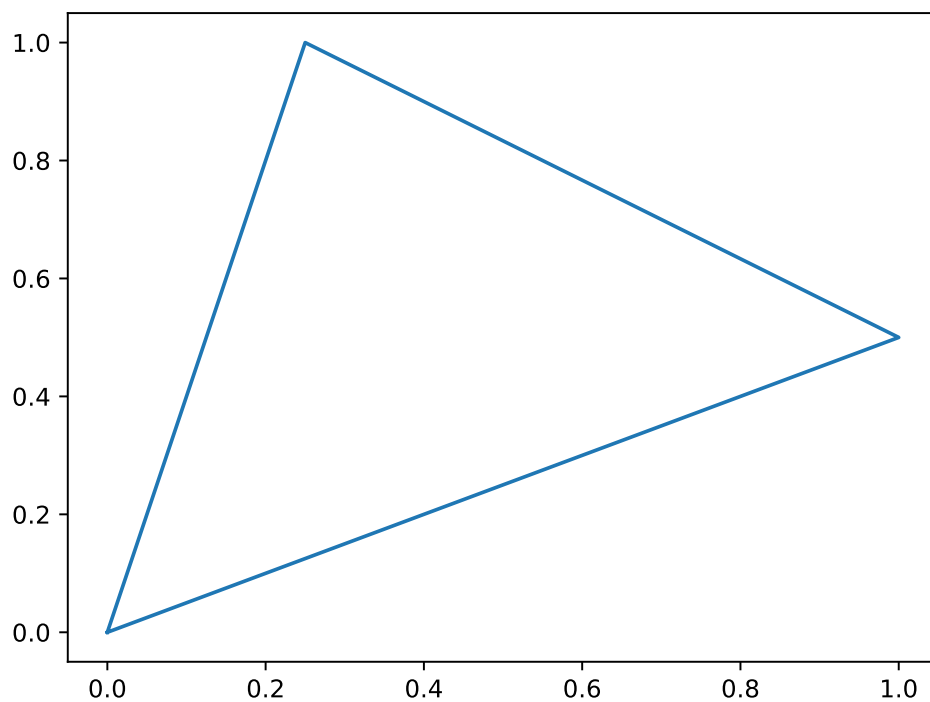
```
Out[4]: <matplotlib.collections.PathCollection at 0x7fcbd18d5358>
```



x координаты не обязаны монотонно возрастать. Тут, например, мы строим замкнутый многоугольник.

```
In [5]: plot([0,0.25,1,0],[0,1,0.5,0])
```

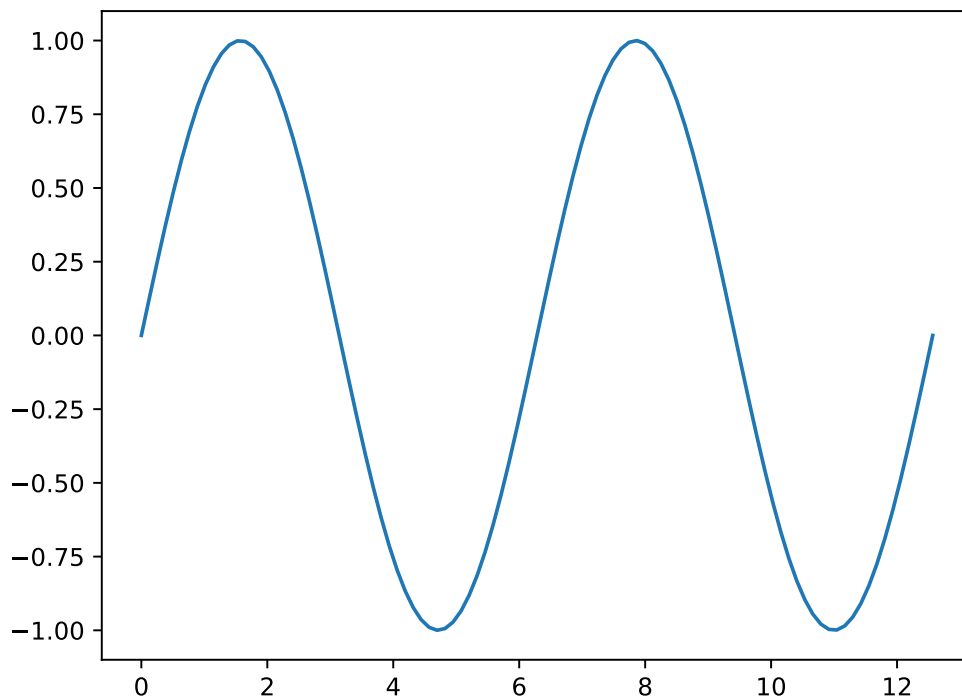
```
Out [5]: [<matplotlib.lines.Line2D at 0x7fcbd17ebdd8>]
```



Когда точек много, ломаная неотличима от гладкой кривой.

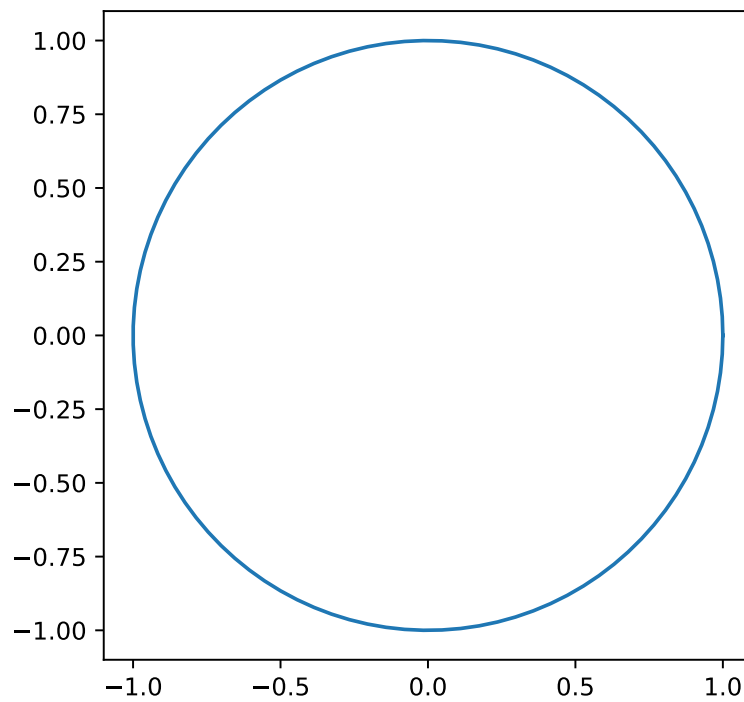
```
In [6]: x=linspace(0,4*pi,100)  
        plot(x,sin(x))
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x7fcbd1780cf8>]
```



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

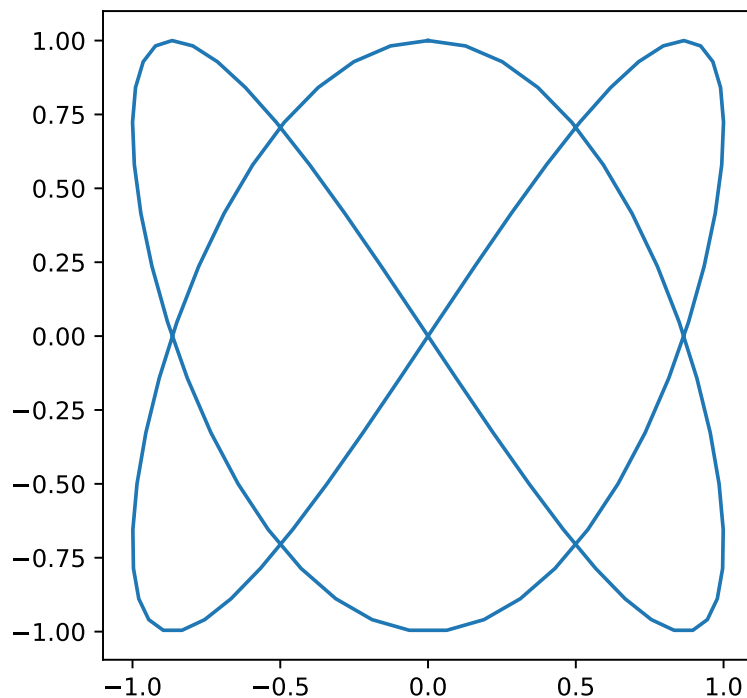
```
In [7]: t=linspace(0,2*pi,100)
        plot(cos(t),sin(t))
        axes().set_aspect(1)
```



Чтобы окружности выглядели как окружности, а не как эллипсы, (а квадраты как квадраты, а не как прямоугольники), нужно установить `aspect ratio`, равный 1.

А вот одна из фигур Лиссажу, которые все мы любили смотреть на осциллографе.

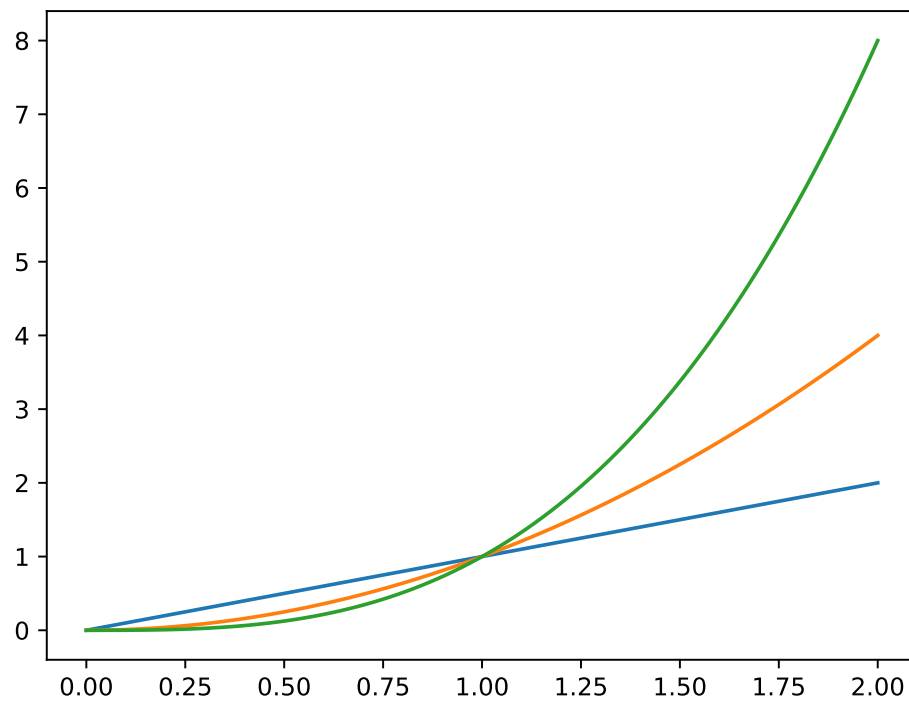
```
In [8]: plot(sin(2*t),cos(3*t))  
        axes().set_aspect(1)
```



Несколько кривых на одном графике. Каждая задаётся парой массивов — x и y координаты. По умолчанию, им присваиваются цвета из некоторой последовательности цветов; разумеется, их можно изменить.

```
In [9]: x=linspace(0,2,100)
        plot(x,x,x,x**2,x,x**3)
```

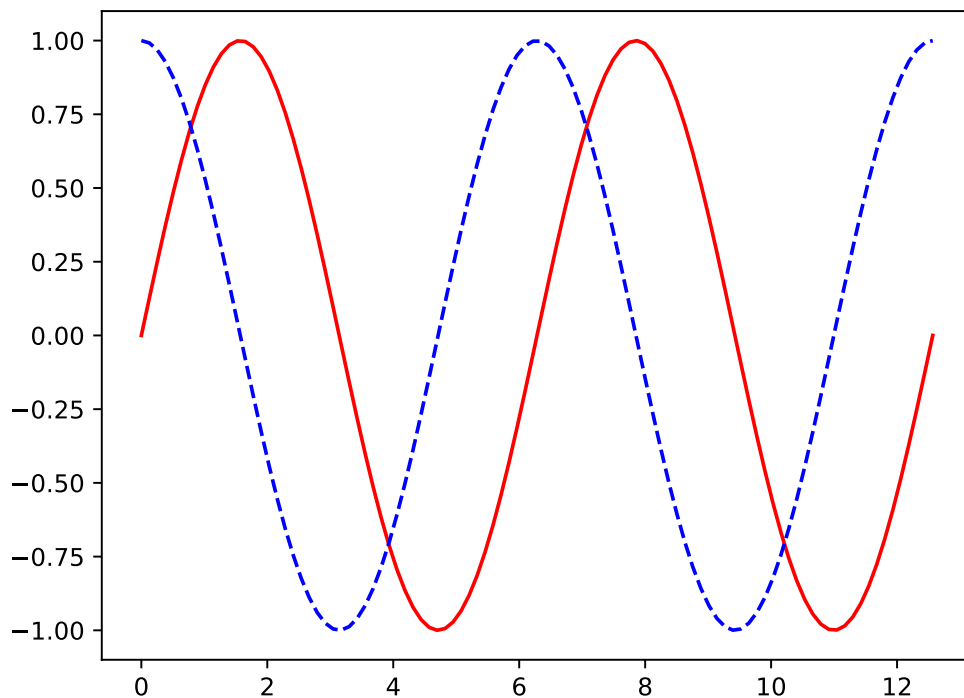
```
Out[9]: [<matplotlib.lines.Line2D at 0x7fcbd155e2e8>,
         <matplotlib.lines.Line2D at 0x7fcbd155e4a8>,
         <matplotlib.lines.Line2D at 0x7fcbd155ee48>]
```

Для простой регулировки цветов и типов линий после пары x и y координат вставляется форматная строка. Первая буква определяет цвет ('r' — красный, 'b' — синий и т.д.), дальше задаётся тип линии ('-' — сплошная, '--' — пунктирная, '-.' — штрих-пунктирная и т.д.).

```
In [10]: x=linspace(0,4*pi,100)
         plot(x,sin(x),'r-',x,cos(x),'b--')
```

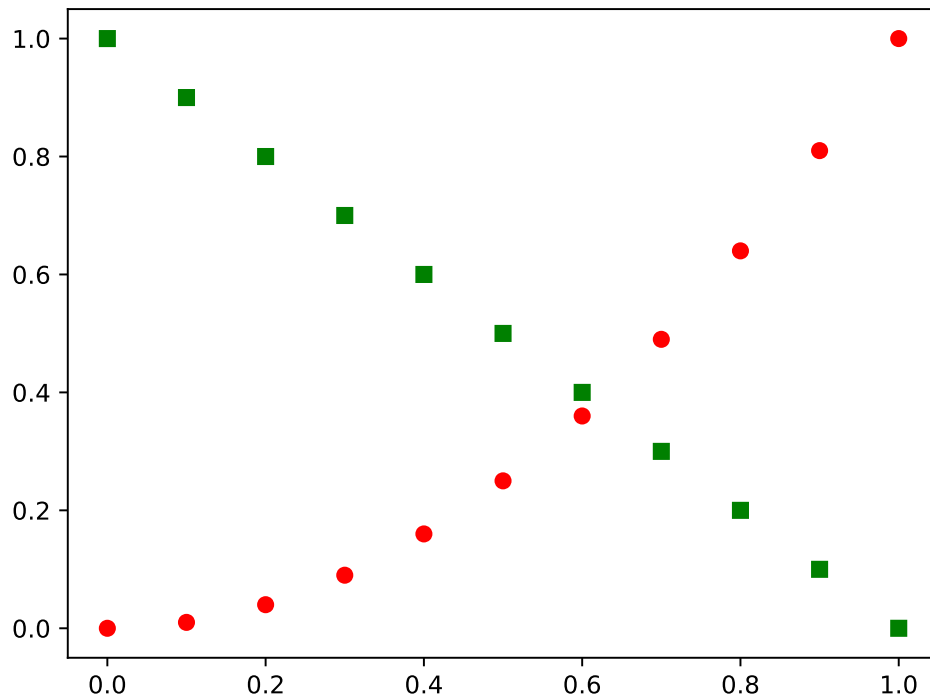
```
Out[10]: [<matplotlib.lines.Line2D at 0x7fcbd179b898>,
          <matplotlib.lines.Line2D at 0x7fcbd1705c50>]
```



Если в качестве “типа линии” указано 'o', то это означает рисовать точки кружочками и не соединять их линиями; аналогично, 's' означает квадратики. Конечно, такие графики имеют смысл только тогда, когда точек не очень много.

```
In [11]: x=linspace(0,1,11)
         plot(x,x**2,'ro',x,1-x,'gs')
```

```
Out[11]: [<matplotlib.lines.Line2D at 0x7fcbd16b9f60>,
          <matplotlib.lines.Line2D at 0x7fcbd16b9cc0>]
```

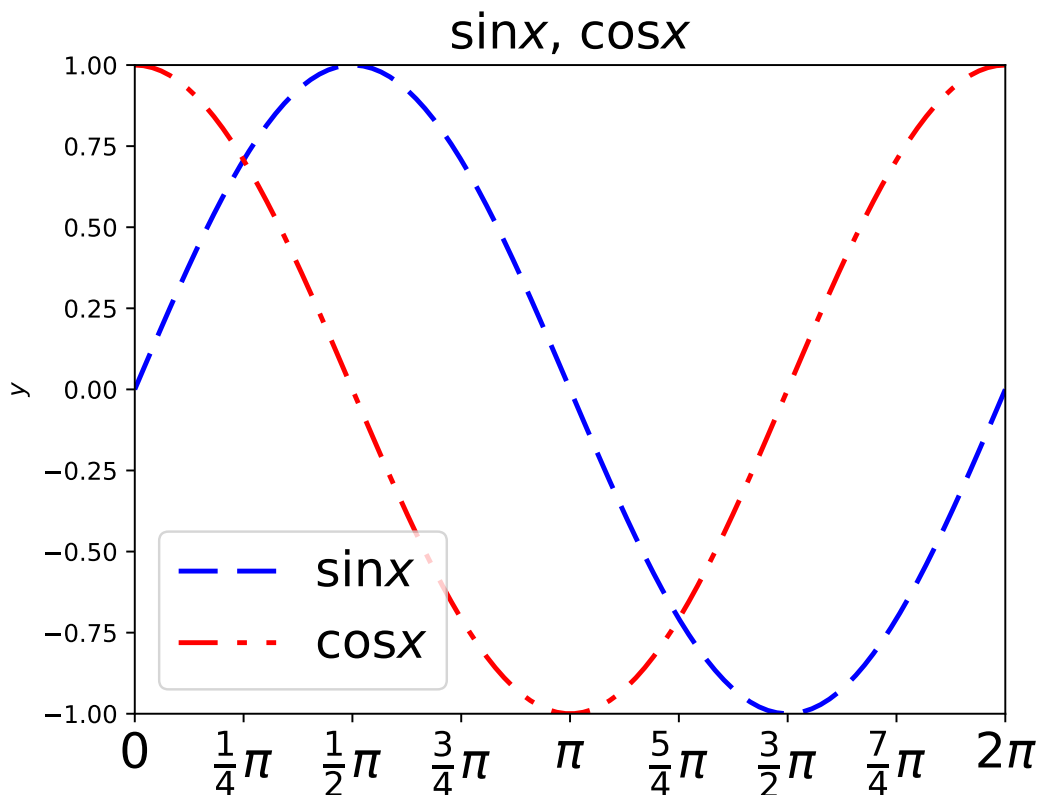


Вот пример настройки почти всего, что можно настроить. Можно задать последовательность засечек на оси x (и y) и подписи к ним (в них, как и в других текстах, можно использовать \LaTeX -овские обозначения). Задать подписи осей x и y и заголовок графика. Во всех текстовых элементах можно задать размер шрифта. Можно задать толщину линий и штрихи (так, на графике косинуса рисуется штрих длины 8, потом участок длины 4 не рисуется, потом участок длины 2 рисуется, потом участок длины 4 опять не рисуется, и так по циклу; поскольку толщина линии равна 2, эти короткие штрихи длины 2 фактически выглядят как точки). Можно задать подписи к кривым (legend); где разместить эти подписи тоже можно регулировать.

```
In [12]: axis([0,2*pi,-1,1])
         xticks(linspace(0,2*pi,9),
                 ('0',r'\frac{1}{4}\pi$',r'\frac{1}{2}\pi$',
                  r'\frac{3}{4}\pi$',r'\pi$',r'\frac{5}{4}\pi$',
                  r'\frac{3}{2}\pi$',r'\frac{7}{4}\pi$',r'2\pi$'),
                 fontsize=20)
         xlabel(r'$x$')
         ylabel(r'$y$')
         title(r'$\sin x$, $\cos x$',fontsize=20)
         x=linspace(0,2*pi,100)
         plot(x,sin(x),linewidth=2,color='b',dashes=[8,4],
              label=r'$\sin x$')
```

```
plot(x,cos(x),linewidth=2,color='r',dashes=[8,4,2,4],
      label=r'$\cos x$')
legend(fontsize=20)
```

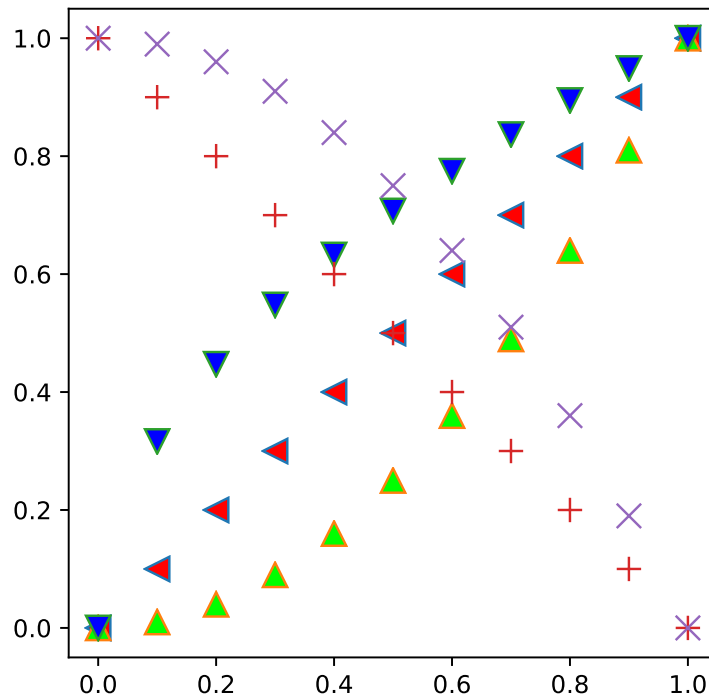
Out [12]: <matplotlib.legend.Legend at 0x7fcdb14af978>



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

```
In [13]: x=linspace(0,1,11)
axis([-0.05,1.05,-0.05,1.05])
axes().set_aspect(1)
plot(x,x,linestyle='',marker='<',markersize=10,
      markerfacecolor='#FF0000')
plot(x,x**2,linestyle='',marker='^',markersize=10,
      markerfacecolor='#00FF00')
plot(x,x**(1/2),linestyle='',marker='v',markersize=10,
      markerfacecolor='#0000FF')
plot(x,1-x,linestyle='',marker='+',markersize=10,
      markerfacecolor='#0F0F00')
plot(x,1-x**2,linestyle='',marker='x',markersize=10,
      markerfacecolor='#0F000F')
```

Out[13]: [`matplotlib.lines.Line2D` at 0x7fcbd12d6ef0>]

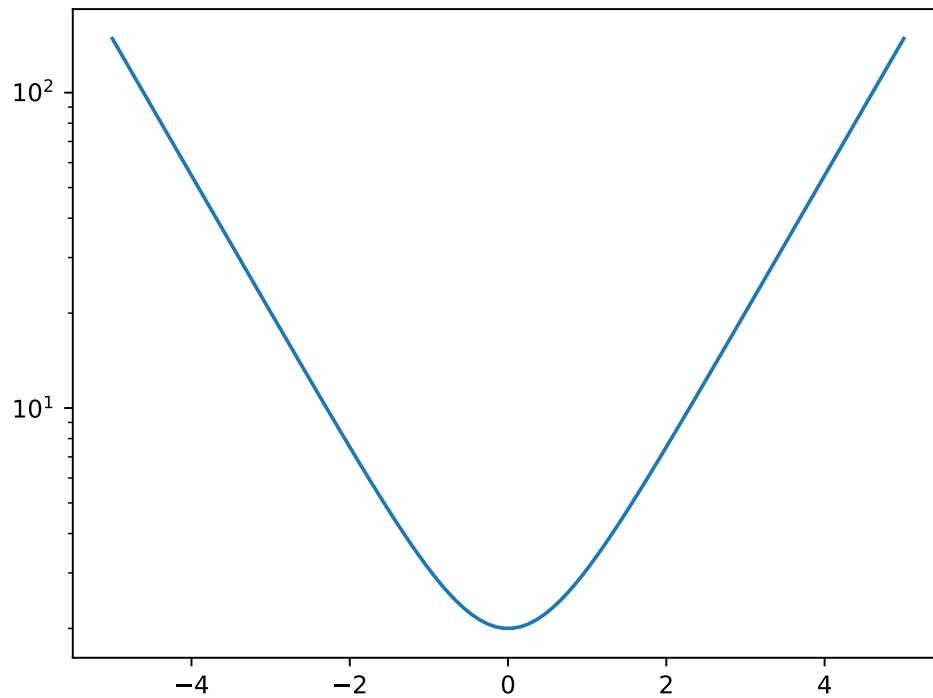


2.2.1 Логарифмический масштаб

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

```
In [14]: x=linspace(-5,5,100)
         yscale('log')
         plot(x,exp(x)+exp(-x))
```

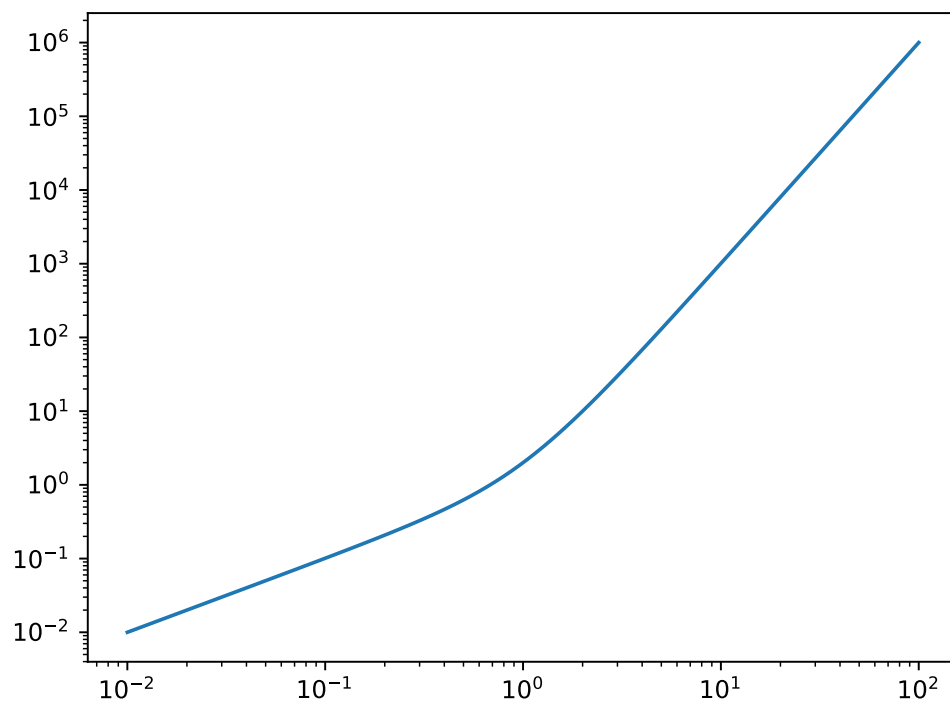
Out[14]: [`matplotlib.lines.Line2D` at 0x7fcbd12ff908>]



Можно задать логарифмический масштаб по обоим осям.

```
In [15]: x=logspace(-2,2,100)
         xscale('log')
         yscale('log')
         plot(x,x+x**3)
```

```
Out[15]: [<matplotlib.lines.Line2D at 0x7fcbd12a55f8>]
```

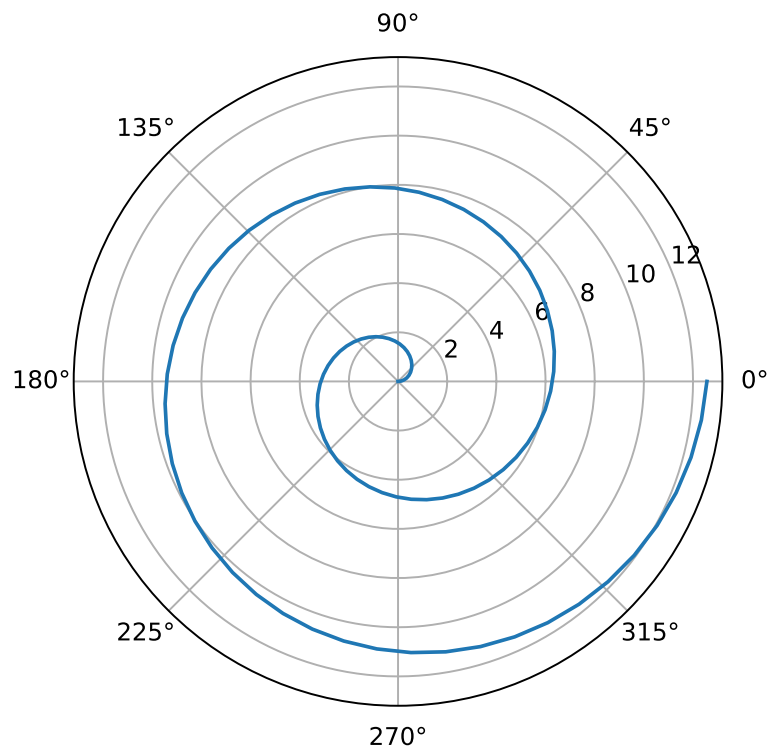


2.2.2 Полярные координаты

Первый массив — φ , второй — r . Вот спираль.

```
In [16]: t=linspace(0,4*pi,100)
         polar(t,t)
```

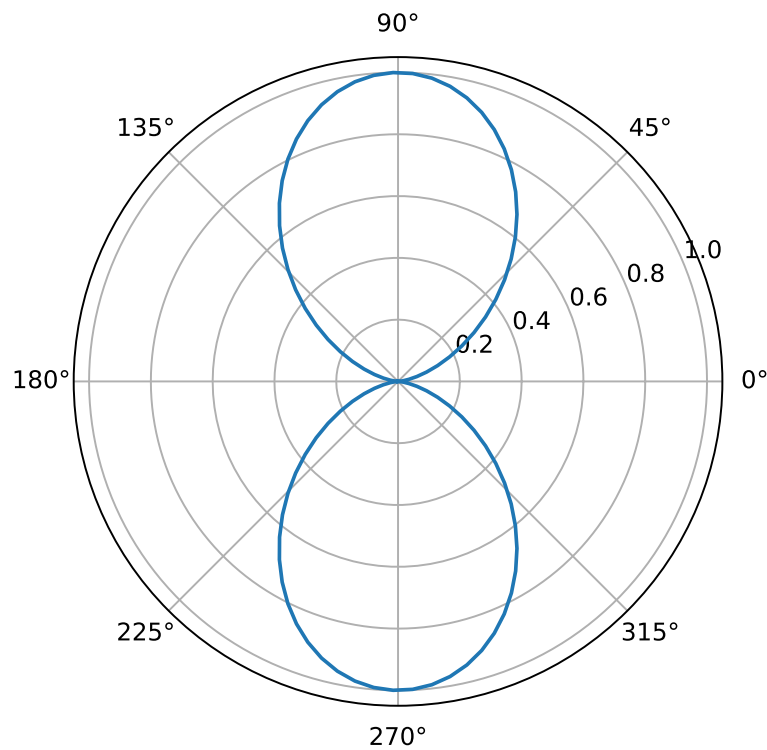
```
Out[16]: [<matplotlib.lines.Line2D at 0x7fcbd0e8f390>]
```



А это угловое распределение пионов в e^+e^- аннигиляции.

```
In [17]: phi=linspace(0,2*pi,100)
         polar(phi,sin(phi)**2)
```

```
Out[17]: [<matplotlib.lines.Line2D at 0x7fcbd0fe2c50>]
```

2.2.3 Экспериментальные данные

Допустим, имеется теоретическая кривая (резонанс без фона).

```
In [18]: xt=linspace(-4,4,101)
         yt=1/(xt**2+1)
```

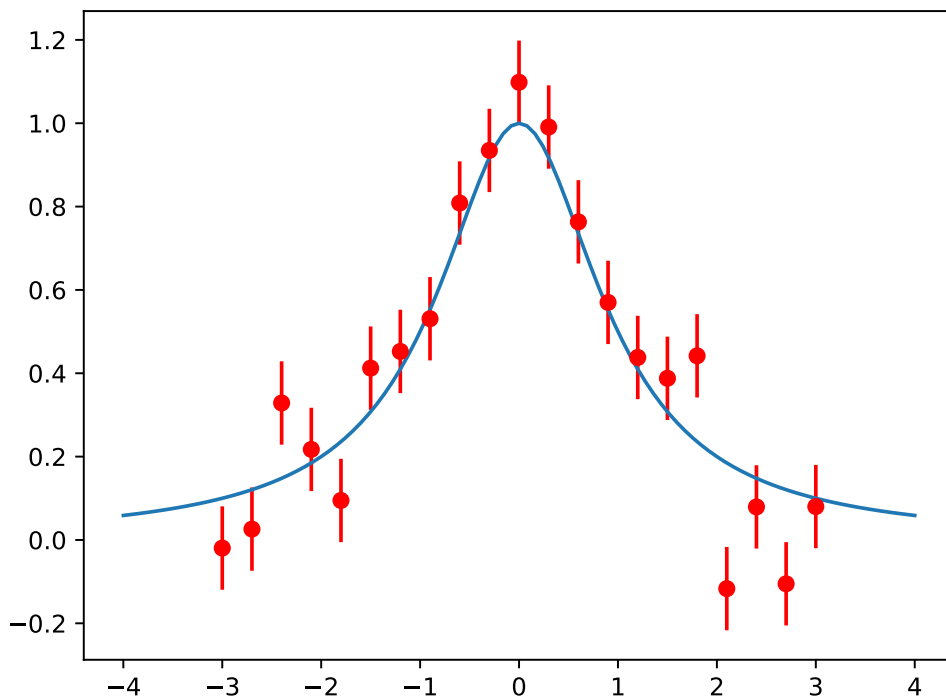
Поскольку реальных экспериментальных данных под рукой нет, мы их сгенерируем. Пусть они согласуются с теорией, и все статистические ошибки равны 0.1.

```
In [19]: xe=linspace(-3,3,21)
         yerr=0.1*ones(21)
         ye=1/(xe**2+1)+yerr*normal(size=21)
```

Экспериментальные точки с усами и теоретическая кривая на одном графике.

```
In [20]: plot(xt,yt)
         errorbar(xe,ye,fmt='ro',yerr=yerr)
```

```
Out[20]: <Container object of 3 artists>
```

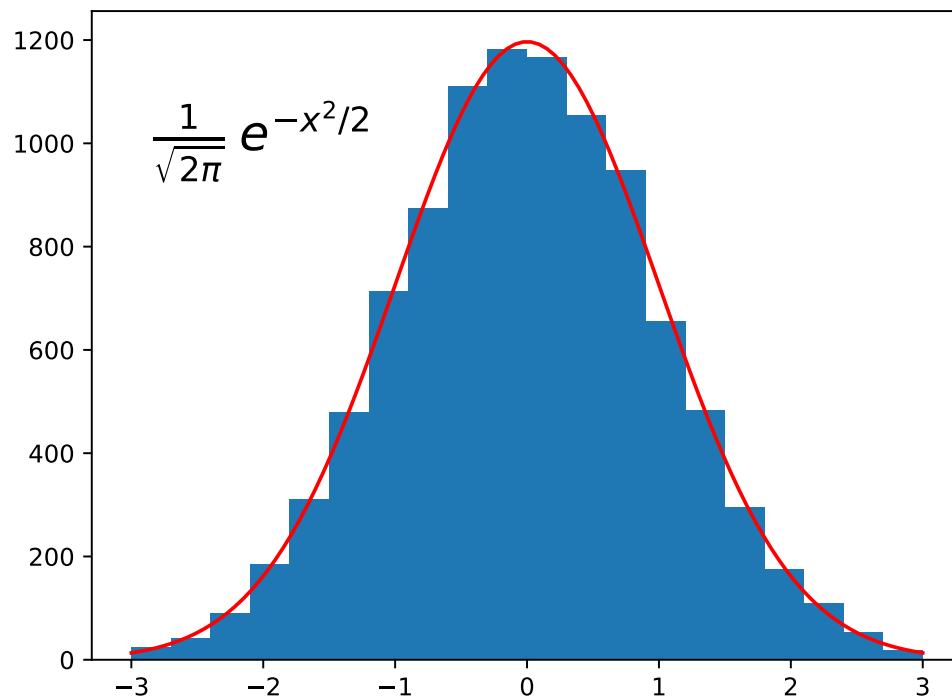


2.2.4 Гистограмма

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

```
In [21]: N=10000
         r=normal(size=N)
         n,bins,patches=hist(r,range=(-3,3),bins=20)
         x=linspace(-3,3,100)
         plot(x,N/sqrt(2*pi)*0.3*exp(-0.5*x**2),'r')
         text(-2,1000,r'$\frac{1}{\sqrt{2\pi}}\backslash,e^{-x^2/2}$',
              fontsize=20,horizontalalignment='center',
              verticalalignment='center')
```

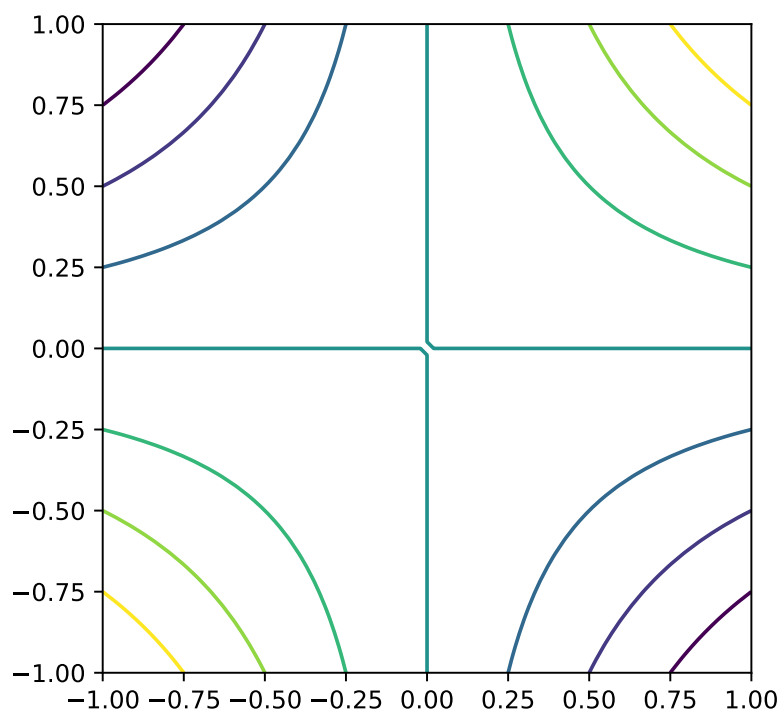
```
Out[21]: <matplotlib.text.Text at 0x7fcbd1011390>
```



2.2.5 Контурные графики

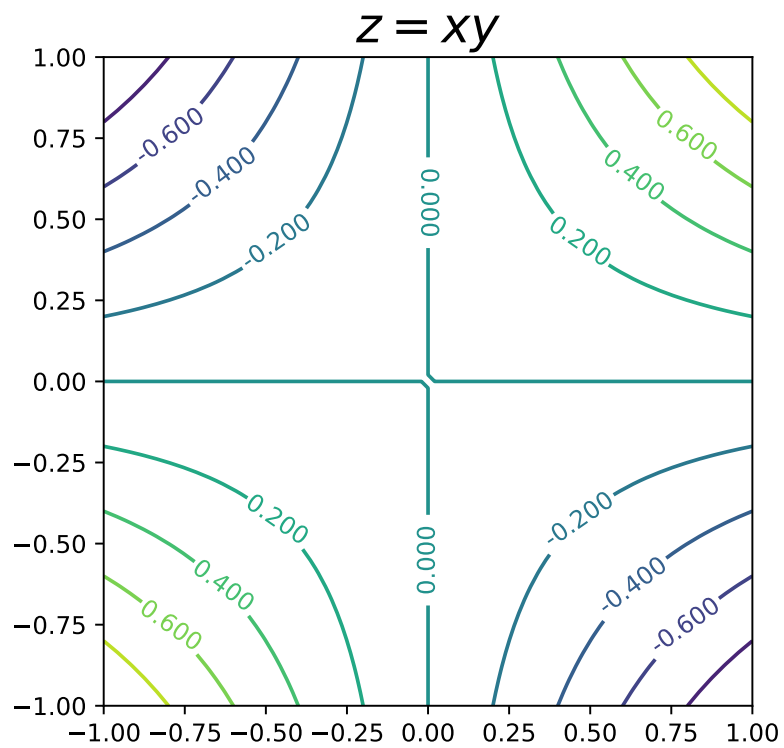
Пусть мы хотим изучить поверхность $z = xy$. Вот её горизонтали.

```
In [22]: x=linspace(-1,1,50)
         y=x
         z=outer(x,y)
         contour(x,y,z)
         axes().set_aspect(1)
```



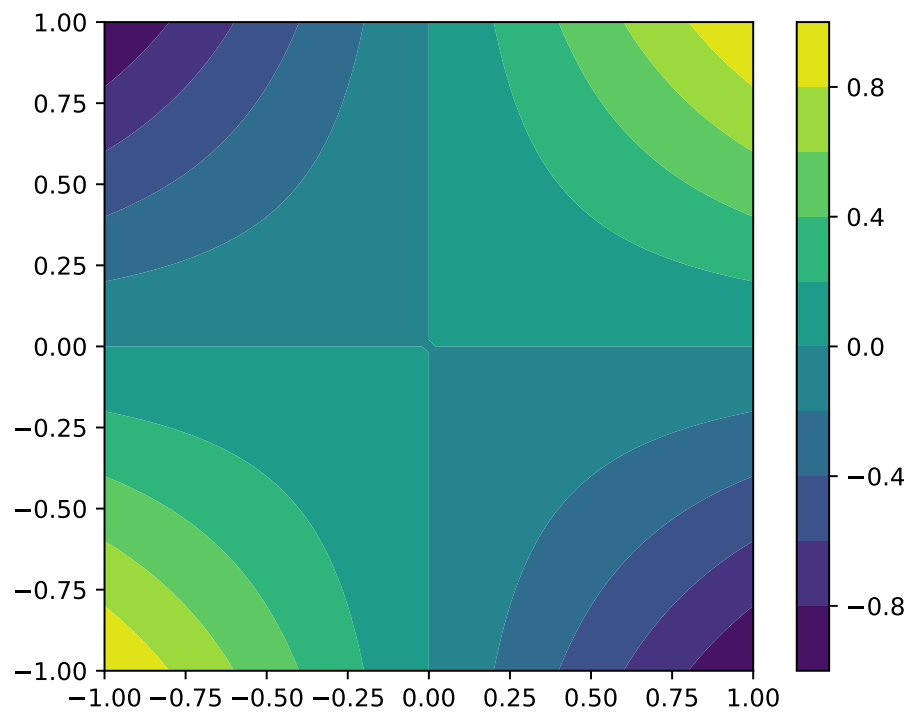
Что-то их маловато. Сделаем побольше и подпишем.

```
In [23]: title(r'$z=xy$', fontsize=20)
         curves=contour(x,y,z,linspace(-1,1,11))
         clabel(curves)
         axes().set_aspect(1)
```



А здесь высота даётся цветом, как на физических географических картах. `colorbar` показывает соответствие цветов и значений z .

```
In [24]: contourf(x,y,z,linspace(-1,1,11))
         colorbar()
         axes().set_aspect(1)
```

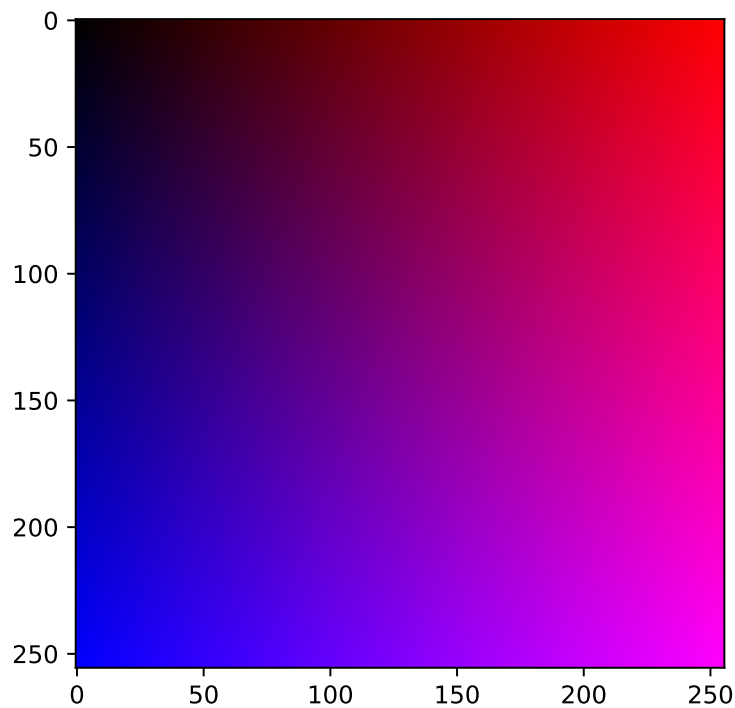


2.2.6 Images (пиксельные картинки)

Картинка задаётся массивом z : $z[i,j]$ — это цвет пикселя i,j , массив из 3 элементов (rgb, числа от 0 до 1).

```
In [25]: n=256
         u=linspace(0,1,n)
         x,y=meshgrid(u,u)
         z=zeros((n,n,3))
         z[:, :, 0]=x
         z[:, :, 2]=y
         imshow(z)
```

```
Out [25]: <matplotlib.image.AxesImage at 0x7fcbd0c9c390>
```



2.2.7 Трёхмерная линия

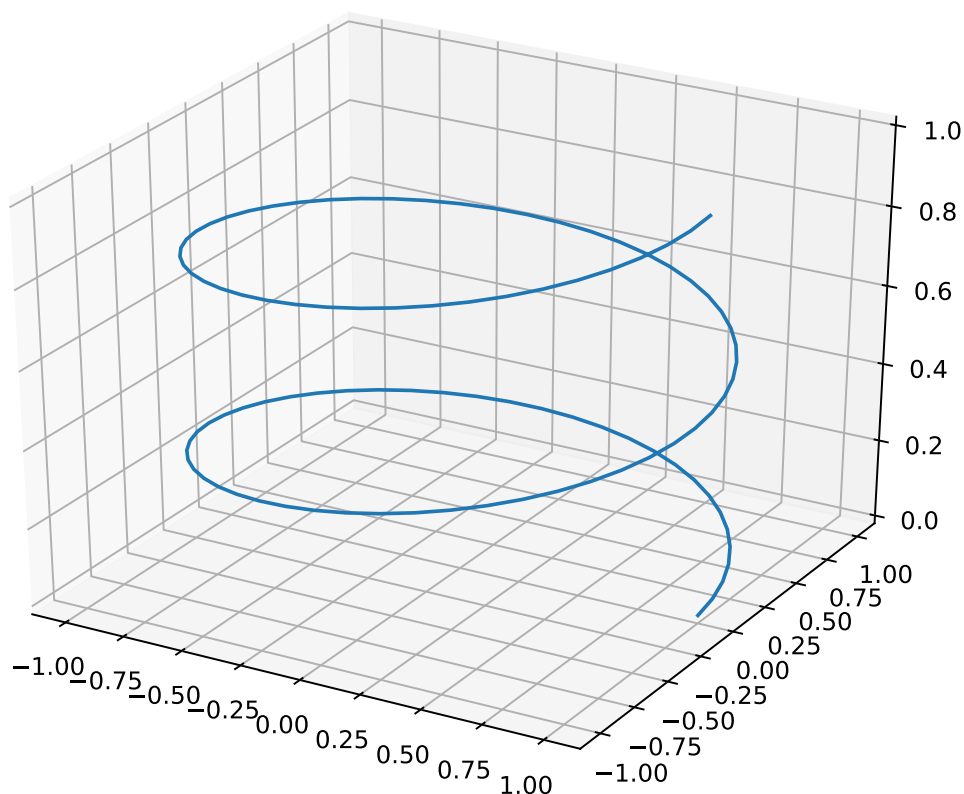
Задаётся параметрически: $x = x(t)$, $y = y(t)$, $z = z(t)$.

```
In [26]: t=linspace(0,4*pi,100)
         x=cos(t)
         y=sin(t)
         z=t/(4*pi)
```

Тут нужен объект класса `Axes3D` из пакета `mpl_toolkits.mplot3d`. `figure()` — это текущий рисунок, создаём в нём объект `ax`, потом используем его методы.

```
In [27]: fig=figure()
         ax=Axes3D(fig)
         ax.plot(x,y,z)
```

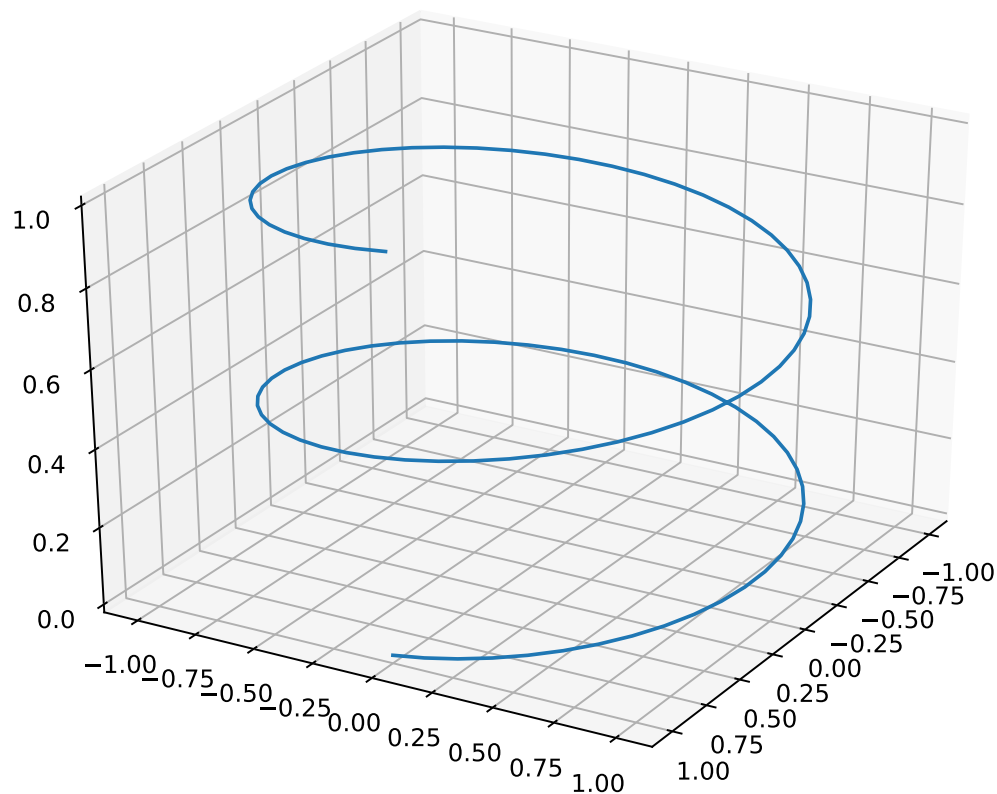
```
Out [27]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7fcbd0c7f780>]
```



К сожалению, `inline` трёхмерную картинку нельзя вертеть мышкой (это можно делать с трёхмерными картинками в отдельных окнах). Но можно задать, с какой стороны мы смотрим.

```
In [28]: fig=figure()
         ax=Axes3D(fig)
         ax.elev,ax.azim=30,30
         ax.plot(x,y,z)
```

```
Out [28]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7fcbd09ff390>]
```

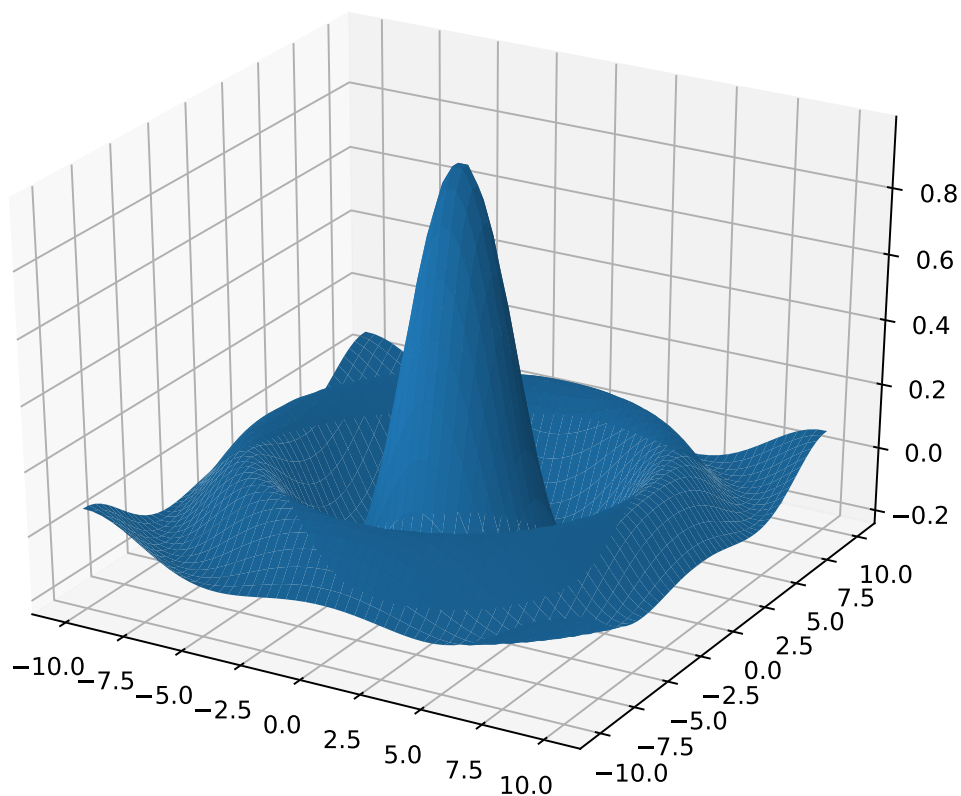



2.2.8 Поверхности

Все поверхности параметрические: $x = x(u, v)$, $y = y(u, v)$, $z = z(u, v)$. Если мы хотим построить явную поверхность $z = z(x, y)$, то удобно создать массивы $x = u$ и $y = v$ функцией `meshgrid`.

```
In [29]: X=10
         N=50
         u=linspace(-X,X,N)
         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)
```

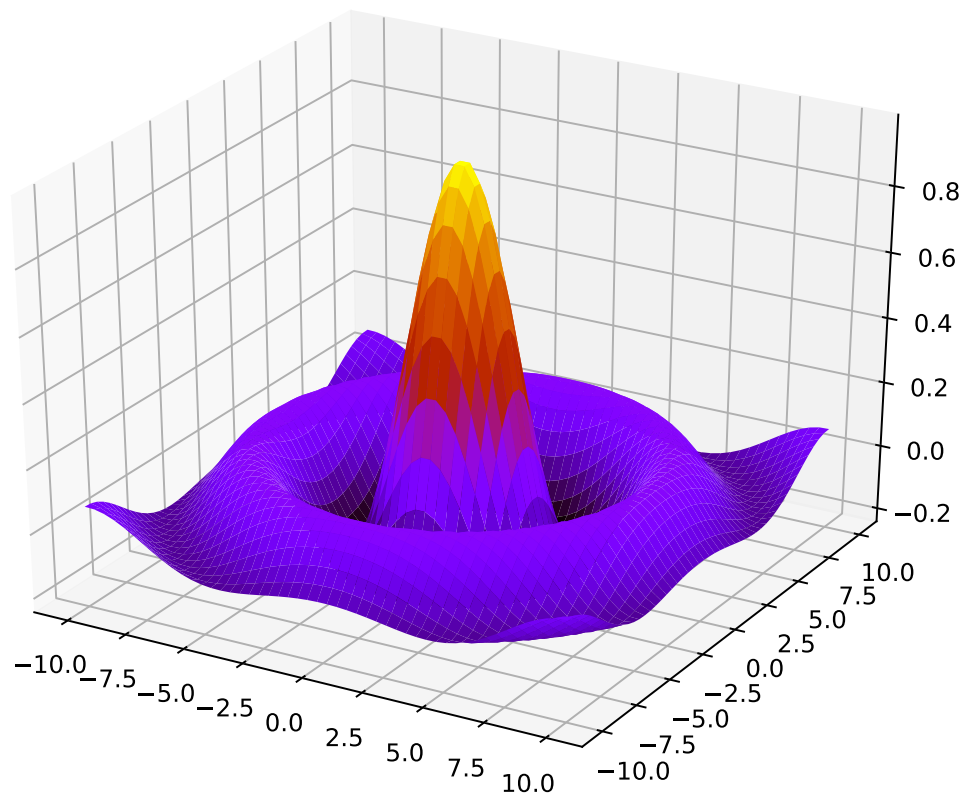
```
Out [29]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fcbd0c95320>
```



Есть много встроенных способов раскраски поверхностей. Так, в методе `gnuplot` цвет зависит от высоты z .

```
In [30]: fig=figure()
         ax=Axes3D(fig)
         ax.plot_surface(x,y,z,rstride=1,cstride=1,cmap='gnuplot')
```

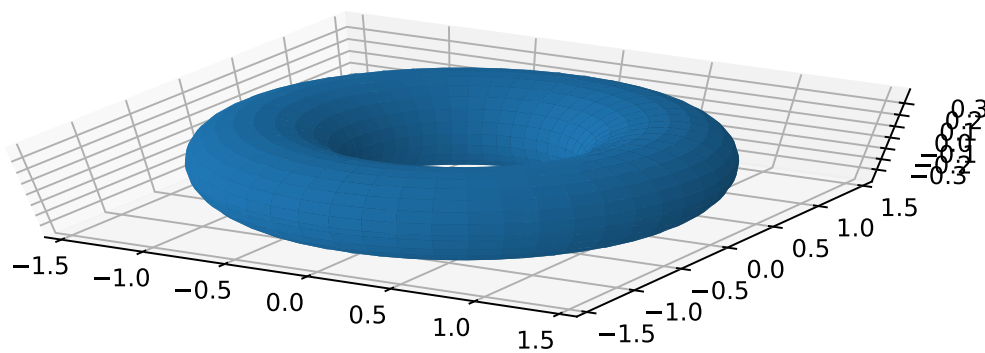
```
Out [30]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fcbd09d1048>
```



Построим бублик — параметрическую поверхность с параметрами ϑ , φ .

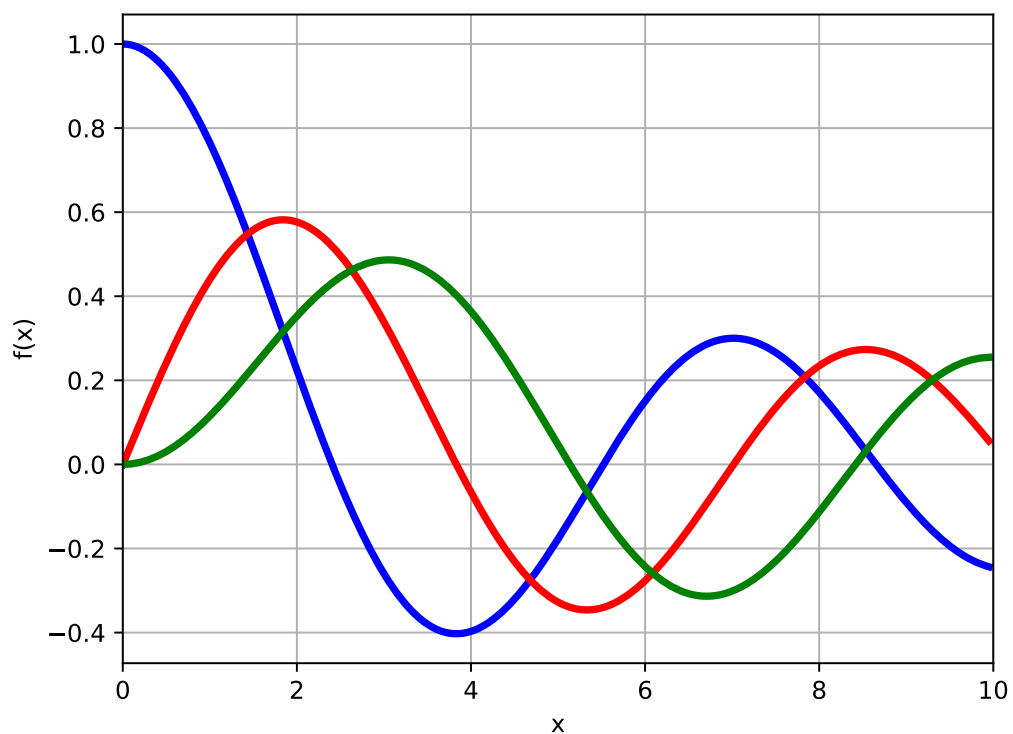
```
In [31]: t=linspace(0,2*pi,50)
         th,ph=meshgrid(t,t)
         r=0.4
         x,y,z=(1+r*cos(ph))*cos(th),(1+r*cos(ph))*sin(th),r*sin(ph)
         fig=figure()
         ax=Axes3D(fig)
         ax.elev=60
         ax.set_aspect(r/(1+r))
         ax.plot_surface(x,y,z,rstride=2,cstride=1)
```

```
Out [31]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fcbd06e6f28>
```

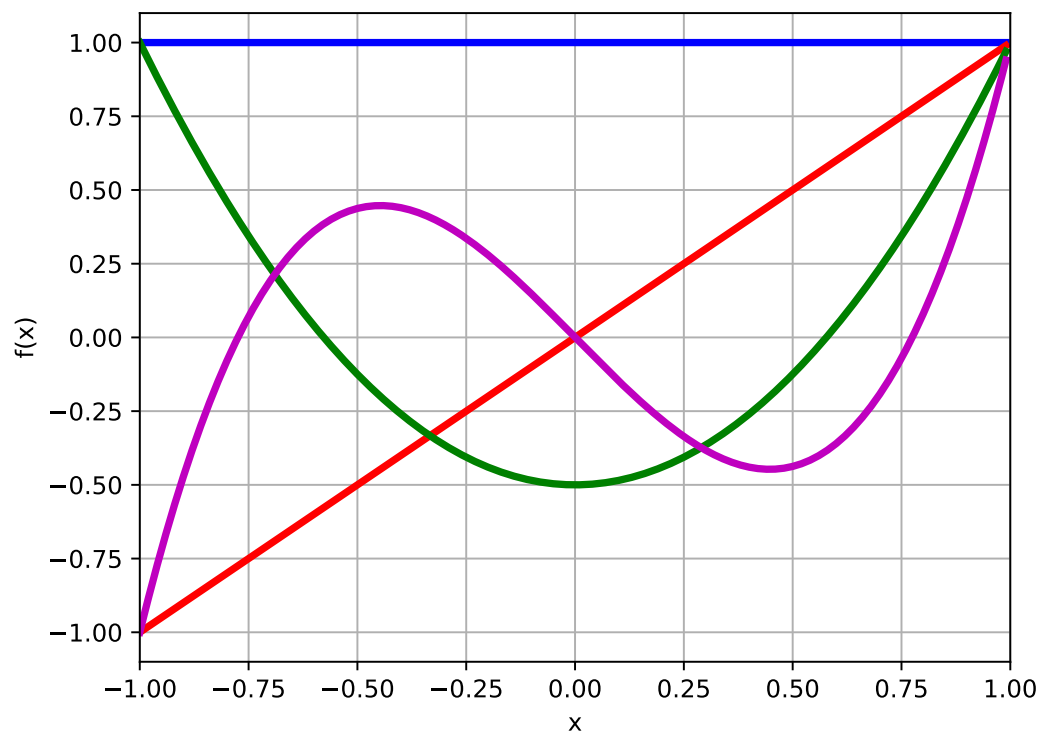


2.3.1 Специальные функции

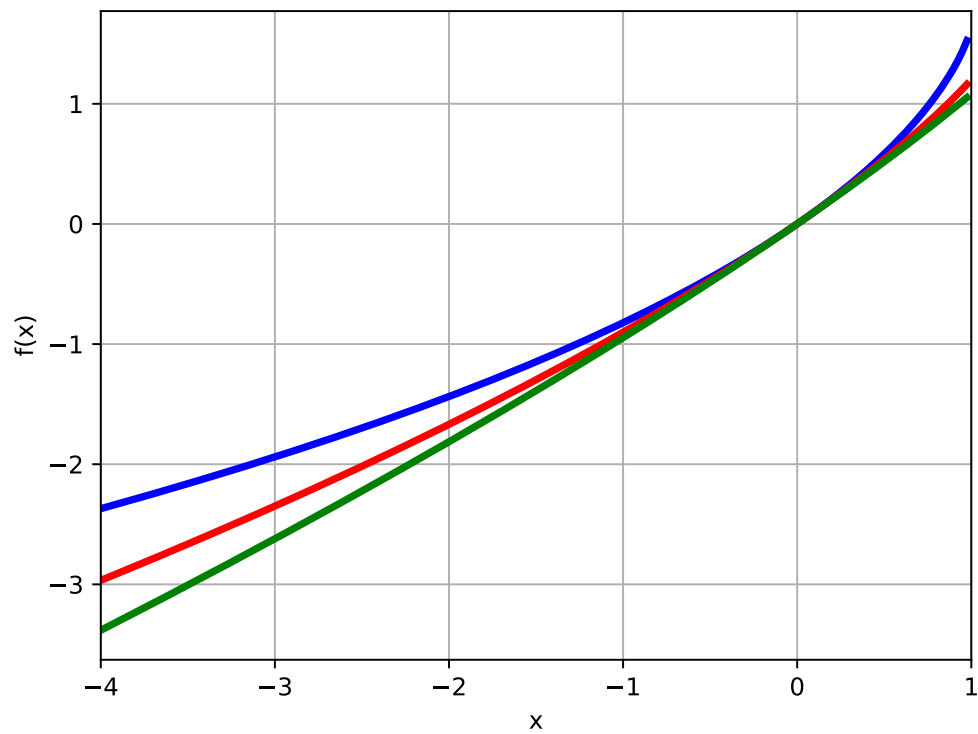
```
In [13]: plot([lambda x:besselj(0,x),  
              lambda x:besselj(1,x),  
              lambda x:besselj(2,x)], [0,10])
```



```
In [14]: plot([lambda x:legendre(0,x),  
              lambda x:legendre(1,x),  
              lambda x:legendre(2,x),  
              lambda x:legendre(3,x)], [-1,1])
```



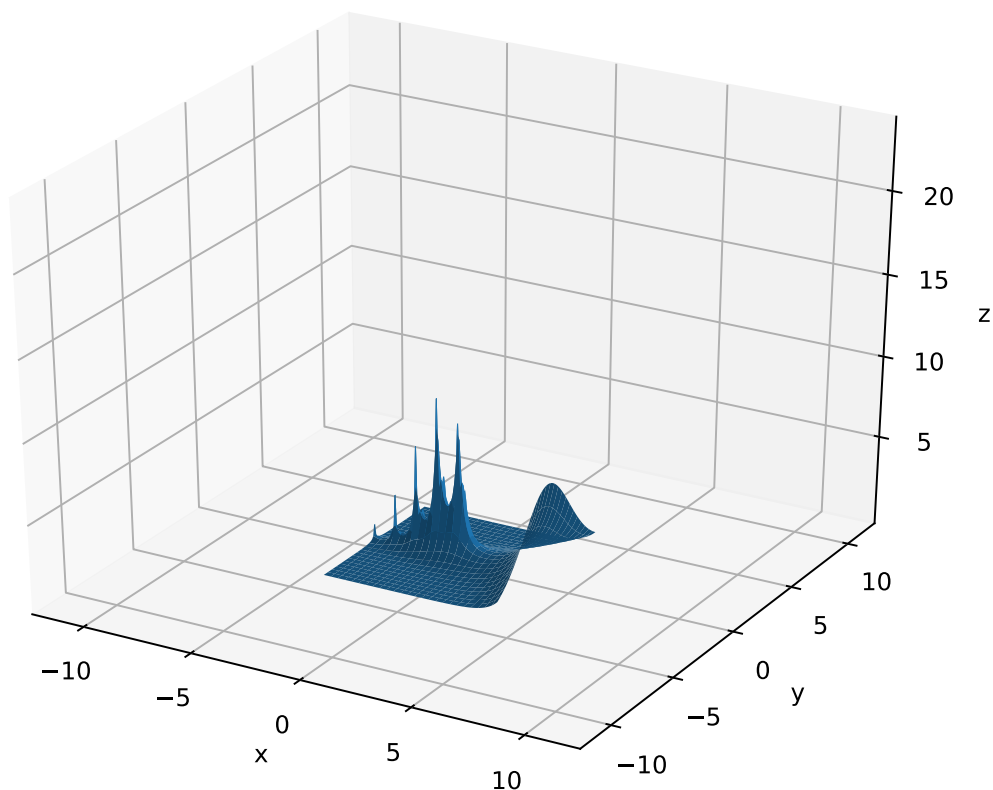
```
In [15]: plot([lambda x:polylog(2,x),  
              lambda x:polylog(3,x),  
              lambda x:polylog(4,x)],[-4,1])
```

```
In [16]: print(pi**2/zeta(2),pi**4/zeta(4))
```

```
6.0 90.0
```

```
In [17]: plot(lambda x,y:abs(gamma(x+j*y)), [-4,4], [-4,4])
```



```
In [18]: gamma(1.5)/sqrt(pi)
```

```
Out[18]: mpf('0.5')
```

2.3.2 Решение уравнений

Корни многочлена

```
In [19]: l=[1,0,0,0,1,1]
         r=polyroots(l)
         for x in r:
             print(x)
```

```
-0.75487766624669276004950889635852869189460661777279
(0.8774388331233463800247544481792643459473033088864 - 0.744861766619744236593170428604392367240j)
(0.8774388331233463800247544481792643459473033088864 + 0.744861766619744236593170428604392367240j)
(-0.5 + 0.86602540378443864676372317075293618347140262690519j)
(-0.5 - 0.86602540378443864676372317075293618347140262690519j)
```

```
In [20]: for x in r:
         print(polyval(l,x))
```

```

0.0
(2.672764710092195646140536467151481878815196880105e-51 - 4.651220902690007103613554345031721715
(2.672764710092195646140536467151481878815196880105e-51 + 4.651220902690007103613554345031721715
(0.0 + 0.0j)
(0.0 + 0.0j)

```

Решение уравнения

```

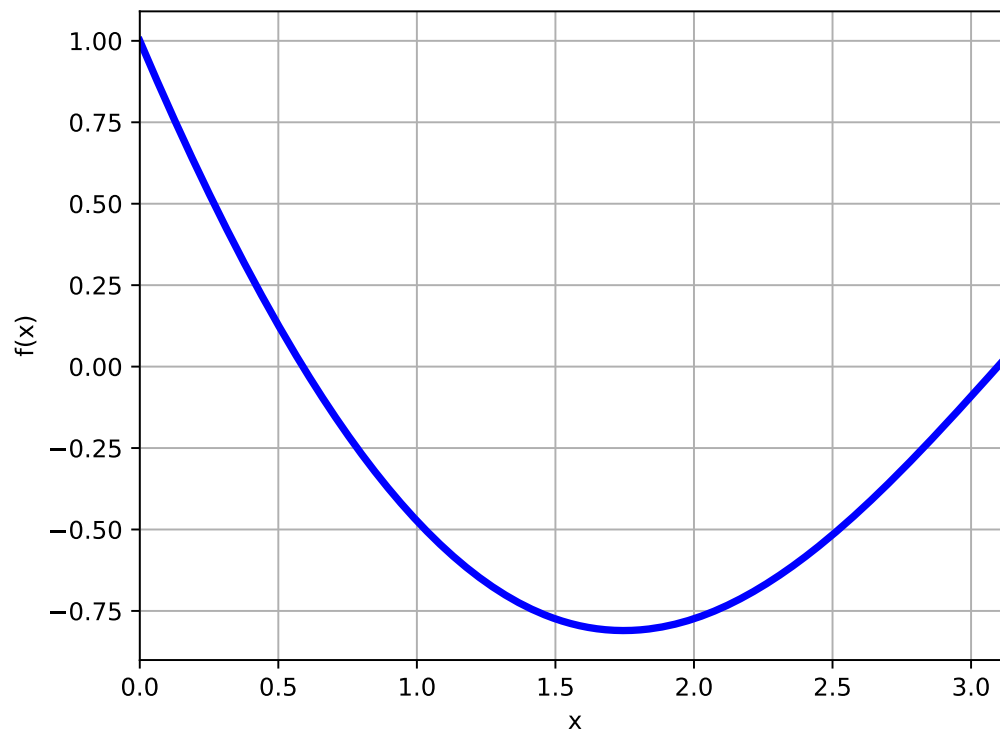
In [21]: def f(x):
         return exp(-x)-sin(x)

```

```

In [22]: plot(f,[0,pi])

```



```

In [23]: findroot(f,(0.5,0.7))

```

```

Out[23]: mpf('0.58853274398186107743245204570290368853127151610903053')

```

Решение системы уравнений

```

In [24]: findroot([lambda x,y:x**2+y**2-1,lambda x,y:x*y-1/4],(1,0.25))

```

```

Out[24]: matrix(
[[ '0.9659258262890682867497431997288973676339048390084' ],
[ '0.25881904510252076234889883762404832834906890131993' ]])

```

2.3.3 Производные

```
In [25]: diff(f,0.5)
```

```
Out [25]: mpf('-1.4841132216030061397200811175950101054335633325969314')
```

```
In [26]: diff(f,0.5,2)
```

```
Out [26]: mpf('1.085956198316836423877087470206751841523721503427788')
```

```
In [27]: diff(lambda x,y:sin(x)*cos(y),(pi,pi),(1,2))
```

```
Out [27]: mpf('-1.0')
```

2.3.4 Интегралы

При вычислении этого интеграла все вычисления будут производиться с точностью, на 5 значащих цифр большей; затем она вернётся к прежней.

```
In [28]: with extradps(5):
          I=quad(lambda x:log(x)**2/(1+x),(0,1))
          print(I)
```

```
1.803085354739391428099607242267174986147479438510748322
```

Допустим, у нас есть причины подозревать, что этот интеграл равен $\zeta(3)$, умноженному на рациональное число (с не очень большими числителем и знаменателем). `pslq([x1,x2,...])` находит целые числа n_1, n_2, \dots такие, что $n_1 x_1 + n_2 x_2 + \dots = 0$. Это — метод нахождения тождеств, называемый *экспериментальной математикой*. Для этого часто требуются вычисления с очень высокой точностью.

```
In [29]: pslq([I,zeta(3)])
```

```
Out [29]: [-2, 3]
```

То есть наш интеграл равен $\frac{3}{2}\zeta(3)$. Это, конечно, не доказательство. Но если мы ещё увеличим точность вычисления интеграла, а результат `pslq` не изменится, то мы можем быть практически уверены, что этот результат верен.

Двойной интеграл:

```
In [30]: quad(lambda x,y:1/(1+x*y),[0,1],[0,1])
```

```
Out [30]: mpf('0.82246703342411321823620758332301259460947495060339899')
```

2.3.5 Сумма ряда

```
In [31]: with extradps(5):
          s=nsun(lambda n: (-1)**(n-1)/n**4, (1,inf))
          print(s)
```

```
0.9470328294972459175765032344735219149279070829288860442
```

```
In [32]: pslq([s,pi**4])
```

```
Out [32]: [-720, 7]
```

То есть эта сумма, вероятно, равна $\frac{7}{720}\pi^4$.

2.3.6 Дифференциальные уравнения

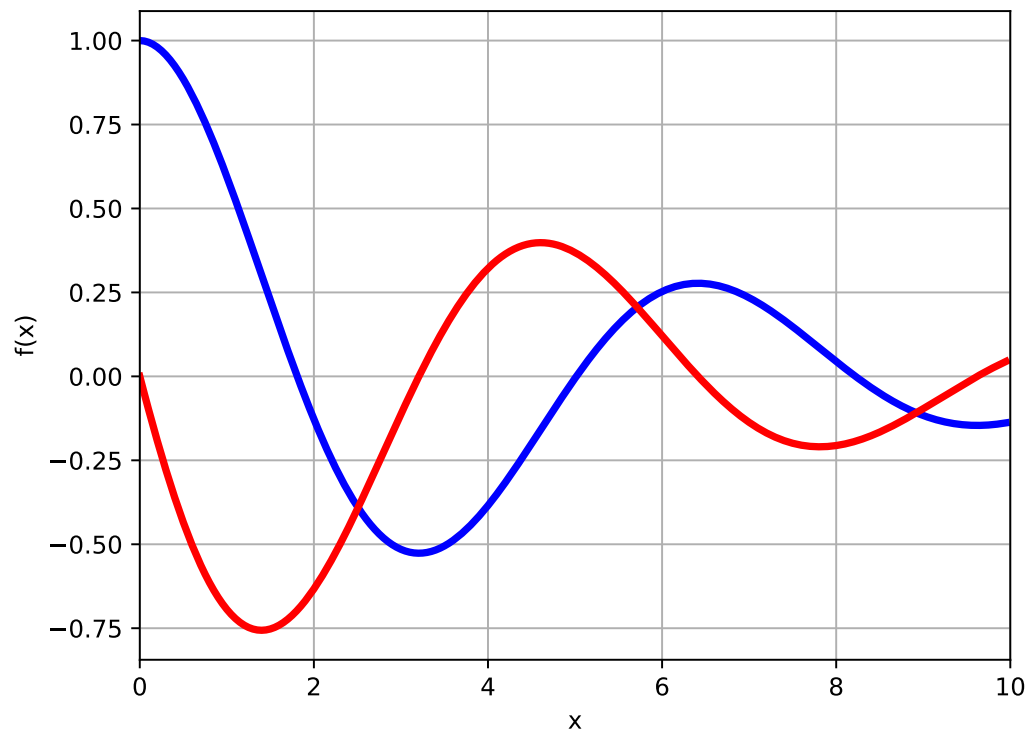
```
In [33]: a=mpf('0.2')
          def f(t,x):
              global a
              return [x[1], -x[0]-2*a*x[1]]
```

```
In [34]: x=odefun(f,0,[1,0])
```

```
In [35]: x(1)
```

```
Out [35]: [mpf('0.59496623263788777500734762840237378880987158574261147'),
           mpf('-0.69387986210972080683187214187798497495035321299936026')]
```

```
In [36]: plot([lambda t:x(t)[0],lambda t:x(t)[1]], [0,10])
```



2.3.7 Матрицы

Матрицы разреженные, реализованы как словари. Квадратная матрица

```
In [37]: matrix(2)
```

```
Out[37]: matrix(
  [['0.0', '0.0'],
   ['0.0', '0.0']])
```

Прямоугольная матрица

```
In [38]: M=matrix(2,3)
M
```

```
Out[38]: matrix(
  [['0.0', '0.0', '0.0'],
   ['0.0', '0.0', '0.0']])
```

```
In [39]: M.rows,M.cols
```

```
Out[39]: (2, 3)
```

```
In [40]: M[0,1]=1
         M
```

```
Out[40]: matrix(
  [['0.0', '1.0', '0.0'],
   ['0.0', '0.0', '0.0']])
```

Операции с матрицами

```
In [41]: M1=matrix([[0,1],[1,0]])
         M2=matrix([[1,0],[0,-1]])
```

```
In [42]: M1+M2
```

```
Out[42]: matrix(
  [['1.0', '1.0'],
   ['1.0', '-1.0']])
```

```
In [43]: M1*M2
```

```
Out[43]: matrix(
  [['0.0', '-1.0'],
   ['1.0', '0.0']])
```

```
In [44]: M2*M1
```

```
Out[44]: matrix(
  [['0.0', '1.0'],
   ['-1.0', '0.0']])
```

```
In [45]: M1**(-1)
```

```
Out[45]: matrix(
  [['0.0', '1.0'],
   ['1.0', '0.0']])
```

Решение системы линейных уравнений

```
In [46]: A=matrix([[1,2],[3,4]])
         b=matrix([1,-1])
         b
```

```
Out[46]: matrix(
  [['1.0'],
   ['-1.0']])
```

```
In [47]: x=lu_solve(A,b)
         x
```

```
Out[47]: matrix(
  [['-3.0'],
   ['2.0']])
```

In [48]: $A \cdot x - b$

Out [48]: matrix(
 [['0.0'],
 ['0.0']])

Собственные значения и собственные векторы

In [49]: $\lambda, u = \text{eig}(A)$
 λ

Out [49]: [mpf('-0.37228132326901432992530573410946465911013222899139797'),
 mpf('5.3722813232690143299253057341094646591101322289914067')]

In [50]: u

Out [50]: matrix(
 [['-0.82456484013239376536905071707877267896095335074304', '-0.422229150415260453359290',
 ['0.56576746496899228472288762798052673125191630934726', '-0.9230523142501933331886156

Диагональная матрица

In [51]: $L = \text{diag}(\lambda)$
 L

Out [51]: matrix(
 [['-0.3722813232690143299253057341094646591101322289914', '0.0'],
 ['0.0', '5.3722813232690143299253057341094646591101322289914']])

In [52]: $A \cdot u - u \cdot L$

Out [52]: matrix(
 [['-2.0045735325691467346054023503636114091113976600788e-51', '5.3455294201843912922810',
 ['0.0', '1.069105884036878258456214586860592751526078752042e-50']])

2.4 pandas

Пакет для статистической обработки данных, по функциональности близкий к R.

```
In [1]: import numpy as np
        import pandas as pd
```

2.4.1 Series

Одномерный набор данных. Отсутствующие данные записываются как `np.nan` (в этот день термометр сломался или метеоролог был пьян); они не участвуют в вычислении средних, среднеквадратичных отклонений и т.д.

```
In [2]: l=[1,3,5,np.nan,6,8]
        s=pd.Series(l)
        s
```

```
Out[2]: 0    1.0
        1    3.0
        2    5.0
        3    NaN
        4    6.0
        5    8.0
        dtype: float64
```

Основная информация о наборе данных: среднее, среднеквадратичное отклонение, минимум, максимум, медиана (которая отличается от среднего для несимметричных распределений).

```
In [3]: s.describe()
```

```
Out[3]: count    5.000000
        mean     4.600000
        std      2.701851
        min      1.000000
        25%      3.000000
        50%      5.000000
        75%      6.000000
        max      8.000000
        dtype: float64
```

Обычная индексация.

```
In [4]: s[2]
```

```
Out[4]: 5.0
```

```
In [5]: s[2]=7
        s
```

```
Out [5]: 0    1.0
          1    3.0
          2    7.0
          3   NaN
          4    6.0
          5    8.0
          dtype: float64
```

```
In [6]: s[2:5]
```

```
Out [6]: 2    7.0
          3   NaN
          4    6.0
          dtype: float64
```

```
In [7]: s1=s[1:]
          s1
```

```
Out [7]: 1    3.0
          2    7.0
          3   NaN
          4    6.0
          5    8.0
          dtype: float64
```

```
In [8]: s2=s[:-1]
          s2
```

```
Out [8]: 0    1.0
          1    3.0
          2    7.0
          3   NaN
          4    6.0
          dtype: float64
```

В сумме `s1+s2` складываются данные с одинаковыми индексами. Поскольку в `s1` нет дан-ного и индексом 0, а в `s2` — с индексом 5, в `s1+s2` в соответствующих позициях будет `NaN`.

```
In [9]: s1+s2
```

```
Out [9]: 0    NaN
          1    6.0
          2   14.0
          3    NaN
          4   12.0
          5    NaN
          dtype: float64
```

К наборам данных можно применять функции из `numpy`.

```
In [10]: np.exp(s)
```

```
Out[10]: 0      2.718282
         1     20.085537
         2    1096.633158
         3         NaN
         4     403.428793
         5    2980.957987
         dtype: float64
```

При создании набора данных `s` мы не указали, что будет играть роль индекса. По умолчанию это последовательность целых чисел 0, 1, 2, ...

```
In [11]: s.index
```

```
Out[11]: RangeIndex(start=0, stop=6, step=1)
```

Но можно создавать наборы данных с индексом, заданным списком.

```
In [12]: i=list('abcdef')
         i
```

```
Out[12]: ['a', 'b', 'c', 'd', 'e', 'f']
```

```
In [13]: s=pd.Series(l,index=i)
         s
```

```
Out[13]: a      1.0
         b      3.0
         c      5.0
         d      NaN
         e      6.0
         f      8.0
         dtype: float64
```

```
In [14]: s['c']
```

```
Out[14]: 5.0
```

Если индекс — строка, то вместо `s['c']` можно писать `s.c`.

```
In [15]: s.c
```

```
Out[15]: 5.0
```

Набор данных можно создать из словаря.

```
In [16]: s=pd.Series({'a':1,'b':2,'c':0})
         s
```

```
Out[16]: a    1
         b    2
         c    0
         dtype: int64
```

Можно отсортировать набор данных.

```
In [17]: s.sort_values()
```

```
Out[17]: c    0
         a    1
         b    2
         dtype: int64
```

Роль индекса может играть, скажем, последовательность дат (или времён измерения и т.д.).

```
In [2]: d=pd.date_range('20170101',periods=10)
        d
```

```
Out[2]: DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04',
                        '2017-01-05', '2017-01-06', '2017-01-07', '2017-01-08',
                        '2017-01-09', '2017-01-10'],
                        dtype='datetime64[ns]', freq='D')
```

```
In [3]: s=pd.Series(np.random.normal(size=10),index=d)
        s
```

```
Out[3]: 2017-01-01    0.356263
        2017-01-02   -0.149695
        2017-01-03    0.823284
        2017-01-04    1.936065
        2017-01-05    0.309854
        2017-01-06    0.642161
        2017-01-07    0.499560
        2017-01-08    0.004974
        2017-01-09    0.245381
        2017-01-10    0.951140
        Freq: D, dtype: float64
```

Операции сравнения возвращают наборы булевых данных.

```
In [4]: s>0
```

```
Out[4]: 2017-01-01    True
        2017-01-02   False
        2017-01-03    True
        2017-01-04    True
        2017-01-05    True
        2017-01-06    True
```

```

2017-01-07    True
2017-01-08    True
2017-01-09    True
2017-01-10    True
Freq: D, dtype: bool

```

Если такой булев набор использовать для индексации, получится поднабор только из тех данных, для которых условие есть `True`.

```
In [5]: s[s>0]
```

```

Out [5]: 2017-01-01    0.356263
         2017-01-03    0.823284
         2017-01-04    1.936065
         2017-01-05    0.309854
         2017-01-06    0.642161
         2017-01-07    0.499560
         2017-01-08    0.004974
         2017-01-09    0.245381
         2017-01-10    0.951140
         dtype: float64

```

Кумулятивные максимумы — от первого элемента до текущего.

```
In [6]: s.cummax()
```

```

Out [6]: 2017-01-01    0.356263
         2017-01-02    0.356263
         2017-01-03    0.823284
         2017-01-04    1.936065
         2017-01-05    1.936065
         2017-01-06    1.936065
         2017-01-07    1.936065
         2017-01-08    1.936065
         2017-01-09    1.936065
         2017-01-10    1.936065
         Freq: D, dtype: float64

```

Кумулятивные суммы.

```
In [7]: s=s.cumsum()
         s
```

```

Out [7]: 2017-01-01    0.356263
         2017-01-02    0.206568
         2017-01-03    1.029852
         2017-01-04    2.965918
         2017-01-05    3.275771
         2017-01-06    3.917933

```

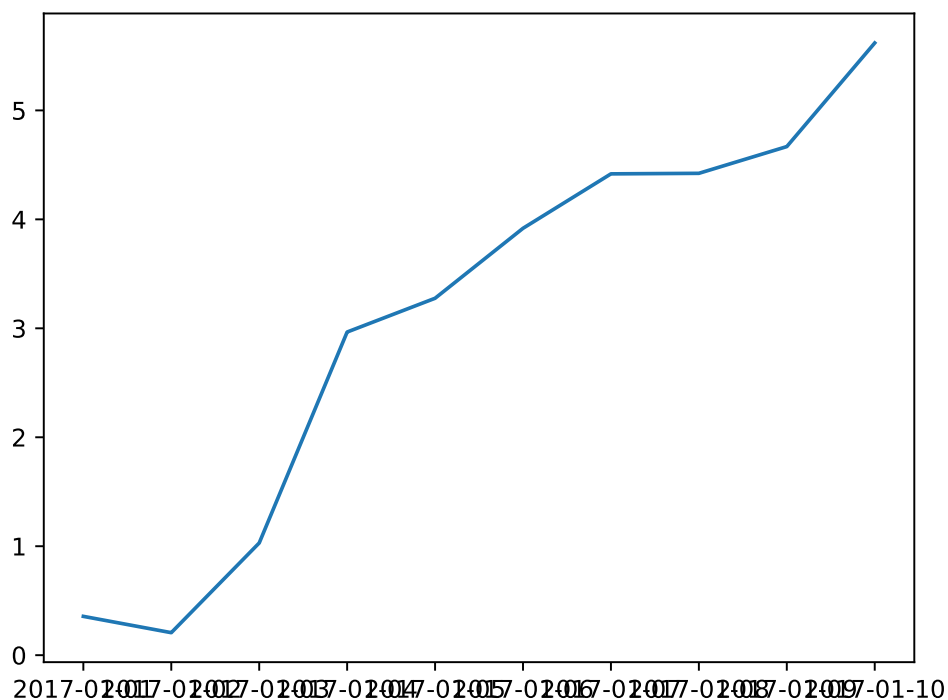
```
2017-01-07    4.417493
2017-01-08    4.422467
2017-01-09    4.667848
2017-01-10    5.618988
Freq: D, dtype: float64
```

Построим график.

```
In [24]: import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [25]: plt.plot(s)
```

```
Out[25]: [<matplotlib.lines.Line2D at 0x7f13629c9080>]
```



2.4.2 DataFrame

Двумерная таблица данных. Имеет индекс и набор столбцов (возможно, имеющих разные типы). Таблицу можно построить, например, из словаря, значениями в котором являются одномерные наборы данных.

```
In [26]: d={'one':pd.Series([1,2,3],index=['a','b','c']),
          'two':pd.Series([1,2,3,4],index=['a','b','c','d'])}
df=pd.DataFrame(d)
df
```

```
Out[26]:
```

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

```
In [27]: df.index
```

```
Out[27]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [28]: df.columns
```

```
Out[28]: Index(['one', 'two'], dtype='object')
```

Если в качестве индекса указать имя столбца, получится одномерный набор данных.

```
In [29]: df['one']
```

```
Out[29]:
```

a	1.0
b	2.0
c	3.0
d	NaN

Name: one, dtype: float64

```
In [30]: df.one
```

```
Out[30]:
```

a	1.0
b	2.0
c	3.0
d	NaN

Name: one, dtype: float64

```
In [31]: df['one']['c']
```

```
Out[31]: 3.0
```

Однако если указать диапазон индексов, то это означает диапазон строк. Причём последняя строка включается в таблицу.

```
In [32]: df['b':'d']
```

```
Out[32]:
```

	one	two
b	2.0	2
c	3.0	3
d	NaN	4

Диапазон целых чисел даёт диапазон строк с такими номерами, не включая последнюю строку (как обычно при индексировании списков). Всё это кажется довольно нелогичным.

```
In [33]: df[1:3]
```

```
Out[33]:      one  two
b    2.0    2
c    3.0    3
```

Логичнее работает атрибут `loc`: первая позиция — всегда индекс строки, а вторая — столбца.

```
In [34]: df.loc['b']
```

```
Out[34]: one    2.0
two    2.0
Name: b, dtype: float64
```

```
In [35]: df.loc['b', 'one']
```

```
Out[35]: 2.0
```

```
In [36]: df.loc['a': 'b', 'one']
```

```
Out[36]: a    1.0
b    2.0
Name: one, dtype: float64
```

```
In [37]: df.loc['a': 'b', :]
```

```
Out[37]:      one  two
a    1.0    1
b    2.0    2
```

```
In [38]: df.loc[:, 'one']
```

```
Out[38]: a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

К таблице можно добавлять новые столбцы.

```
In [39]: df['three']=df['one']*df['two']
df['flag']=df['two']>2
df
```

```
Out[39]:      one  two  three  flag
a    1.0    1    1.0  False
b    2.0    2    4.0  False
c    3.0    3    9.0   True
d    NaN    4    NaN   True
```


И удалять имеющиеся.

```
In [40]: del df['two']
         df['foo']=0.
         df
```

```
Out[40]:
```

	one	three	flag	foo
a	1.0	1.0	False	0.0
b	2.0	4.0	False	0.0
c	3.0	9.0	True	0.0
d	NaN	NaN	True	0.0

Добавим копию столбца `one`, в которую входят только строки до второй.

```
In [41]: df['one_tr']=df['one'][:2]
         df
```

```
Out[41]:
```

	one	three	flag	foo	one_tr
a	1.0	1.0	False	0.0	1.0
b	2.0	4.0	False	0.0	2.0
c	3.0	9.0	True	0.0	NaN
d	NaN	NaN	True	0.0	NaN

```
In [42]: df=df.loc[:,['one','one_tr']]
         df
```

```
Out[42]:
```

	one	one_tr
a	1.0	1.0
b	2.0	2.0
c	3.0	NaN
d	NaN	NaN

Можно объединять таблицы по вертикали и по горизонтали.

```
In [43]: df2=pd.DataFrame({'one':{'e':0,'f':1},'one_tr':{'e':2}})
         df2
```

```
Out[43]:
```

	one	one_tr
e	0	2.0
f	1	NaN

```
In [44]: pd.concat([df,df2])
```

```
Out[44]:
```

	one	one_tr
a	1.0	1.0
b	2.0	2.0
c	3.0	NaN
d	NaN	NaN
e	0.0	2.0
f	1.0	NaN

```
In [45]: df2=pd.DataFrame({'two':{'a':0,'b':1},'three':{'c':2,'d':3}})
df2
```

```
Out[45]:
```

	three	two
a	NaN	0.0
b	NaN	1.0
c	2.0	NaN
d	3.0	NaN

```
In [46]: pd.concat([df,df2],axis=1)
```

```
Out[46]:
```

	one	one_tr	three	two
a	1.0	1.0	NaN	0.0
b	2.0	2.0	NaN	1.0
c	3.0	NaN	2.0	NaN
d	NaN	NaN	3.0	NaN

Создадим таблицу из массива случайных чисел.

```
In [10]: df=pd.DataFrame(np.random.randn(10,4),
                           columns=['A','B','C','D'])
df
```

```
Out[10]:
```

	A	B	C	D
0	0.706305	-0.789569	-0.692519	0.340655
1	0.277662	1.168946	-0.456736	-0.824495
2	-1.742185	-1.623243	-0.188642	-0.164130
3	-0.486917	-0.404665	0.828688	1.960935
4	-0.009105	1.221108	0.399887	3.075690
5	0.440579	1.513206	0.251294	0.838322
6	1.800180	1.877250	-0.721819	-1.637900
7	0.518764	1.372829	-0.513600	-0.544454
8	-0.823112	-1.188590	1.861447	1.272845
9	1.103270	-1.380487	0.227549	0.769414

```
In [11]: df2=pd.DataFrame(np.random.randn(7,3),columns=['A','B','C'])
df+df2
```

```
Out[11]:
```

	A	B	C	D
0	-0.396002	-0.388489	-0.938762	NaN
1	0.025981	-0.822861	-1.221219	NaN
2	-1.859502	-0.265075	-1.399910	NaN
3	-0.937928	-0.118183	-0.413946	NaN
4	-0.038995	1.159641	2.223911	NaN
5	0.478176	-0.599153	-1.214517	NaN
6	1.387845	0.992897	-0.214836	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

```
In [12]: 2*df+3
```

```
Out [12]:
```

	A	B	C	D
0	4.412610	1.420863	1.614962	3.681311
1	3.555324	5.337893	2.086527	1.351011
2	-0.484370	-0.246485	2.622715	2.671739
3	2.026166	2.190670	4.657376	6.921870
4	2.981789	5.442216	3.799774	9.151381
5	3.881158	6.026413	3.502587	4.676644
6	6.600359	6.754500	1.556363	-0.275800
7	4.037527	5.745658	1.972800	1.911092
8	1.353775	0.622821	6.722893	5.545690
9	5.206539	0.239026	3.455098	4.538828

```
In [13]: np.sin(df)
```

```
Out [13]:
```

	A	B	C	D
0	0.649027	-0.710050	-0.638478	0.334105
1	0.274108	0.920339	-0.441021	-0.734205
2	-0.985349	-0.998625	-0.187526	-0.163394
3	-0.467903	-0.393711	0.737045	0.924856
4	-0.009105	0.939480	0.389314	0.065855
5	0.426463	0.998342	0.248657	0.743522
6	0.973807	0.953409	-0.660751	-0.997749
7	0.495807	0.980468	-0.491316	-0.517951
8	-0.733266	-0.927844	0.958058	0.955940
9	0.892686	-0.981946	0.225590	0.695714

```
In [14]: df.describe()
```

```
Out [14]:
```

	A	B	C	D
count	10.000000	10.000000	10.000000	10.000000
mean	0.178544	0.176679	0.099555	0.508688
std	1.007094	1.373741	0.805262	1.391815
min	-1.742185	-1.623243	-0.721819	-1.637900
25%	-0.367464	-1.088834	-0.499384	-0.449373
50%	0.359121	0.382141	0.019453	0.555035
75%	0.659420	1.334899	0.362739	1.164214
max	1.800180	1.877250	1.861447	3.075690

```
In [15]: df.sort_values(by='B')
```

```
Out [15]:
```

	A	B	C	D
2	-1.742185	-1.623243	-0.188642	-0.164130
9	1.103270	-1.380487	0.227549	0.769414
8	-0.823112	-1.188590	1.861447	1.272845
0	0.706305	-0.789569	-0.692519	0.340655
3	-0.486917	-0.404665	0.828688	1.960935
1	0.277662	1.168946	-0.456736	-0.824495

```

4 -0.009105  1.221108  0.399887  3.075690
7  0.518764  1.372829 -0.513600 -0.544454
5  0.440579  1.513206  0.251294  0.838322
6  1.800180  1.877250 -0.721819 -1.637900

```

Атрибут `iloc` подобен `loc`: первый индекс — номер строки, второй — номер столбца. Это целые числа, конец диапазона на включается (как обычно в питоне).

```
In [16]: df.iloc[2]
```

```

Out[16]: A    -1.742185
         B    -1.623243
         C    -0.188642
         D    -0.164130
         Name: 2, dtype: float64

```

```
In [17]: df.iloc[1:3]
```

```

Out[17]:
         A         B         C         D
1  0.277662  1.168946 -0.456736 -0.824495
2 -1.742185 -1.623243 -0.188642 -0.164130

```

```
In [18]: df.iloc[1:3,0:2]
```

```

Out[18]:
         A         B
1  0.277662  1.168946
2 -1.742185 -1.623243

```

Построим графики кумулятивных сумм — мировые линии четырёх пьяных, у которых величина каждого шага — гауссова случайная величина.

```
In [19]: cs=df.cumsum()
         cs
```

```

Out[19]:
         A         B         C         D
0  0.706305 -0.789569 -0.692519  0.340655
1  0.983967  0.379378 -1.149255 -0.483839
2 -0.758218 -1.243865 -1.337898 -0.647969
3 -1.245135 -1.648530 -0.509210  1.312965
4 -1.254240 -0.427422 -0.109322  4.388656
5 -0.813661  1.085784  0.141971  5.226978
6  0.986519  2.963035 -0.579848  3.589078
7  1.505282  4.335864 -1.093447  3.044624
8  0.682170  3.147274  0.767999  4.317469
9  1.785440  1.766788  0.995548  5.086883

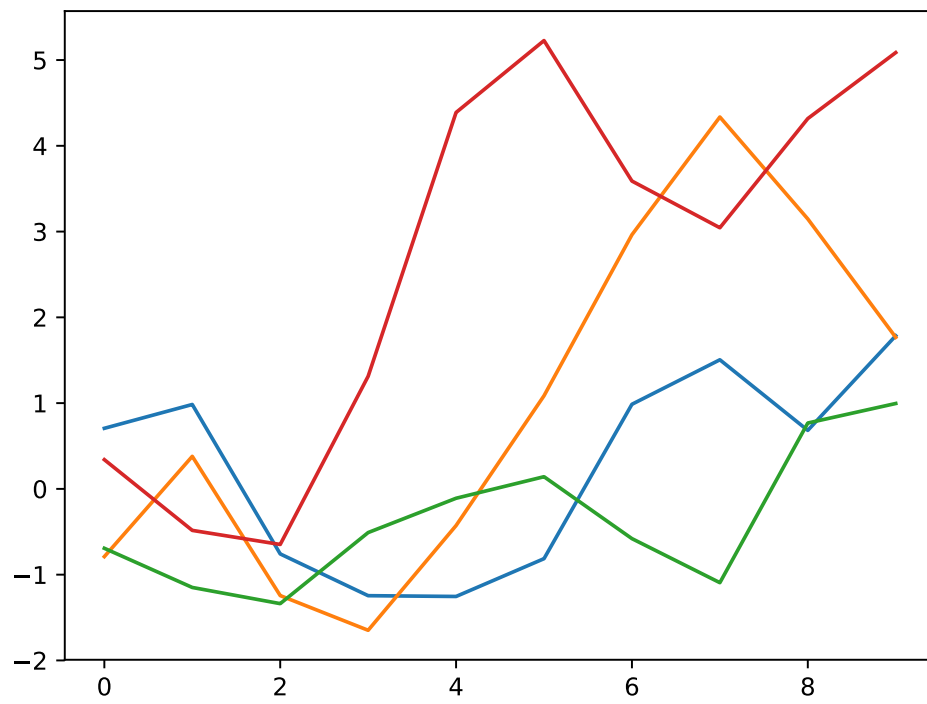
```

```
In [57]: plt.plot(cs)
```

```

Out[57]: [<matplotlib.lines.Line2D at 0x7f13628a3390>,
         <matplotlib.lines.Line2D at 0x7f13628a3550>,
         <matplotlib.lines.Line2D at 0x7f13628a3748>,
         <matplotlib.lines.Line2D at 0x7f13628a3940>]

```



2.5 SymPy

SymPy — это пакет для символьных вычислений на питоне, подобный системе *Mathematica*. Он работает с выражениями, содержащими символы.

```
In [1]: from sympy import *
        init_printing()
```

Основными кирпичиками, из которых строятся выражения, являются символы. Символ имеет имя, которое используется при печати выражений. Объекты класса `Symbol` нужно создавать и присваивать переменным питона, чтобы их можно было использовать. В принципе, имя символа и имя переменной, которой мы присваиваем этот символ — две независимые вещи, и можно написать `abc=Symbol('xyz')`. Но тогда при вводе программы Вы будете использовать `abc`, а при печати результатов SymPy будет использовать `xyz`, что приведёт к ненужной путанице. Поэтому лучше, чтобы имя символа совпадало с именем переменной питона, которой он присваивается.

В языках, специально предназначенных для символьных вычислений, таких, как *Mathematica*, если Вы используете переменную, которой ничего не было присвоено, то она автоматически воспринимается как символ с тем же именем. Питон не был изначально предназначен для символьных вычислений. Если Вы используете переменную, которой ничего не было присвоено, Вы получите сообщение об ошибке. Объекты типа `Symbol` нужно создавать явно.

```
In [2]: x=Symbol('x')
```

```
In [3]: a=x**2-1
        a
```

```
Out [3]:
```

$$x^2 - 1$$

```
In [4]: type(a)
```

```
Out [4]: sympy.core.add.Add
```

Можно определить несколько символов одновременно. Строка разбивается на имена по пробелам.

```
In [5]: y,z=symbols('y z')
```

Подставим вместо x выражение $y + 1$.

```
In [6]: a.subs(x,y+1)
```

```
Out [6]:
```

$$(y + 1)^2 - 1$$

2.5.1 Многочлены и рациональные функции

SymPy не раскрывает скобки автоматически. Для этого используется функция `expand`.

```
In [7]: a=(x+y-z)**6
        a
```

```
Out [7]:
```

$$(x + y - z)^6$$

```
In [8]: a=expand(a)
        a
```

```
Out [8]:
```

$$x^6 + 6x^5y - 6x^5z + 15x^4y^2 - 30x^4yz + 15x^4z^2 + 20x^3y^3 - 60x^3y^2z + 60x^3yz^2 - 20x^3z^3 + 15x^2y^4 - 60x^2y^3z + 90x^2y^2z^2 - 60x^2yz^3 + 15x^2z^4 - 60xyz^4 + 15y^6 - 30y^5z + 15y^4z^2 - 30y^3z^3 + 15y^2z^4 - 6yz^5 + z^6$$

Степень многочлена a по x .

```
In [9]: degree(a,x)
```

```
Out [9]:
```

$$6$$

Соберём вместе члены с определёнными степенями x .

```
In [10]: collect(a,x)
```

```
Out [10]:
```

$$x^6 + x^5(6y - 6z) + x^4(15y^2 - 30yz + 15z^2) + x^3(20y^3 - 60y^2z + 60yz^2 - 20z^3) + x^2(15y^4 - 60y^3z + 90y^2z^2 - 60yz^3 + 15z^4) + x(y^5 - 5y^4z + 10y^3z^2 - 10y^2z^3 + 5yz^4 - z^5) + z^6$$

Многочлен с целыми коэффициентами можно записать в виде произведения таких многочленов (причём каждый сомножитель уже невозможно расфакторизовать дальше, оставаясь в рамках многочленов с целыми коэффициентами). Существуют эффективные алгоритмы для решения этой задачи.

```
In [11]: a=factor(a)
        a
```

```
Out [11]:
```

$$(x + y - z)^6$$

SymPy не сокращает отношения многочленов на их наибольший общий делитель автоматически. Для этого используется функция `cancel`.

```
In [12]: a=(x**3-y**3)/(x**2-y**2)
        a
```

Out [12]:

$$\frac{x^3 - y^3}{x^2 - y^2}$$

In [13]: `cancel(a)`

Out [13]:

$$\frac{x^2 + xy + y^2}{x + y}$$

SymPy не приводит суммы рациональных выражений к общему знаменателю автоматически. Для этого используется функция `together`.

In [14]: `a=y/(x-y)+x/(x+y)`
`a`

Out [14]:

$$\frac{x}{x+y} + \frac{y}{x-y}$$

In [15]: `together(a)`

Out [15]:

$$\frac{x(x-y) + y(x+y)}{(x-y)(x+y)}$$

Функция `simplify` пытается переписать выражение *в наиболее простом виде*. Это понятие не имеет чёткого определения (в разных ситуациях *наиболее простыми* могут считаться разные формы выражения), и не существует алгоритма такого упрощения. Функция `simplify` работает эвристически, и невозможно заранее предугадать, какие упрощения она попытается сделать. Поэтому её удобно использовать в интерактивных сессиях, чтобы посмотреть, удастся ли ей записать выражение в каком-нибудь разумном виде, но нежелательно использовать в программах. В них лучше применять более специализированные функции, которые выполняют одно определённое преобразование выражения.

In [16]: `simplify(a)`

Out [16]:

$$\frac{x^2 + y^2}{x^2 - y^2}$$

Разложение на элементарные дроби по отношению к x и y .

In [17]: `apart(a,x)`

Out [17]:

$$-\frac{y}{x+y} + \frac{y}{x-y} + 1$$


```
In [18]: apart(a,y)
```

```
Out[18]:
```

$$\frac{x}{x+y} + \frac{x}{x-y} - 1$$

Подставим конкретные численные значения вместо переменных x и y .

```
In [19]: a=a.subs({x:1,y:2})
a
```

```
Out[19]:
```

$$-\frac{5}{3}$$

А сколько это будет численно?

```
In [20]: a.n()
```

```
Out[20]:
```

$$-1.666666666666667$$

2.5.2 Элементарные функции

SymPy автоматически применяет упрощения элементарных функций (которые справедливы во всех случаях).

```
In [21]: sin(-x)
```

```
Out[21]:
```

$$-\sin(x)$$

```
In [22]: cos(pi/4),tan(5*pi/6)
```

```
Out[22]:
```

$$\left(\frac{\sqrt{2}}{2}, -\frac{\sqrt{3}}{3} \right)$$

SymPy может работать с числами с плавающей точкой, имеющими сколь угодно большую точность. Вот π с 100 значащими цифрами.

```
In [23]: pi.n(100)
```

```
Out[23]:
```

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211700
```

E — это основание натуральных логарифмов.

```
In [24]: log(1),log(E)
```

```
Out[24]:
```

$$(0, \quad 1)$$

```
In [25]: exp(log(x)),log(exp(x))
```

```
Out[25]:
```

$$(x, \quad \log(e^x))$$

А почему не x ? Попробуйте подставить $x = 2\pi i$.

```
In [26]: sqrt(0)
```

```
Out[26]:
```

$$0$$

```
In [27]: sqrt(x)**4,sqrt(x**4)
```

```
Out[27]:
```

$$\left(x^2, \quad \sqrt{x^4}\right)$$

А почему не x^2 ? Попробуйте подставить $x = i$.

Символы могут иметь некоторые свойства. Например, они могут быть положительными. Тогда SymPy может сильнее упростить квадратные корни.

```
In [28]: p,q=symbols('p q',positive=True)
         sqrt(p**2)
```

```
Out[28]:
```

$$p$$

```
In [29]: sqrt(12*x**2*y),sqrt(12*p**2*y)
```

```
Out[29]:
```

$$\left(2\sqrt{3}\sqrt{x^2y}, \quad 2\sqrt{3}p\sqrt{y}\right)$$

Пусть символ n будет целым (I — это мнимая единица).

```
In [30]: n=Symbol('n',integer=True)
         simplify(exp(2*pi*I*n))
```

```
Out[30]:
```

```
In [31]: sin(pi*n)
```

```
Out [31]:
```

$$0$$

Метод `rewrite` пытается переписать выражение в терминах заданной функции.

```
In [32]: cos(x).rewrite(exp), exp(I*x).rewrite(cos)
```

```
Out [32]:
```

$$\left(\frac{e^{ix}}{2} + \frac{1}{2}e^{-ix}, \quad i \sin(x) + \cos(x) \right)$$

```
In [33]: asin(x).rewrite(log)
```

```
Out [33]:
```

$$-i \log \left(ix + \sqrt{-x^2 + 1} \right)$$

Функция `trigsimp` пытается переписать тригонометрическое выражение в *наиболее простом виде*. В программах лучше использовать более специализированные функции.

```
In [34]: trigsimp(2*sin(x)**2+3*cos(x)**2)
```

```
Out [34]:
```

$$\cos^2(x) + 2$$

Функция `expand_trig` разлагает синусы и косинусы сумм и кратных углов.

```
In [35]: expand_trig(sin(x-y)), expand_trig(sin(2*x))
```

```
Out [35]:
```

$$(\sin(x) \cos(y) - \sin(y) \cos(x), \quad 2 \sin(x) \cos(x))$$

Чаще нужно обратное преобразование — произведений и степеней синусов и косинусов в выражения, линейные по этим функциям. Например, пусть мы работаем с отрезком ряда Фурье.

```
In [36]: a1,a2,b1,b2=symbols('a1 a2 b1 b2')
          a=a1*cos(x)+a2*cos(2*x)+b1*sin(x)+b2*sin(2*x)
          a
```

```
Out [36]:
```

$$a_1 \cos(x) + a_2 \cos(2x) + b_1 \sin(x) + b_2 \sin(2x)$$

Мы хотим возвести его в квадрат и опять получить отрезок ряда Фурье.

```
In [37]: a=(a**2).rewrite(exp).expand().rewrite(cos).expand()
a
```

```
Out [37]:
```

$$\frac{a_1^2}{2} \cos(2x) + \frac{a_1^2}{2} + a_1 a_2 \cos(x) + a_1 a_2 \cos(3x) + a_1 b_1 \sin(2x) + a_1 b_2 \sin(x) + a_1 b_2 \sin(3x) + \frac{a_2^2}{2} \cos(4x) + \frac{a_2^2}{2} - a_2 b_1 \sin(x) +$$

```
In [38]: a.collect([cos(x),cos(2*x),cos(3*x),sin(x),sin(2*x),sin(3*x)])
```

```
Out [38]:
```

$$\frac{a_1^2}{2} + a_1 b_1 \sin(2x) + \frac{a_2^2}{2} \cos(4x) + \frac{a_2^2}{2} + a_2 b_2 \sin(4x) + \frac{b_1^2}{2} - \frac{b_2^2}{2} \cos(4x) + \frac{b_2^2}{2} + \left(\frac{a_1^2}{2} - \frac{b_1^2}{2} \right) \cos(2x) + (a_1 a_2 - b_1 b_2) \cos(3x) +$$

Функция `expand_log` преобразует логарифмы произведений и степеней в суммы логарифмов (только для положительных величин); `logcombine` производит обратное преобразование.

```
In [39]: a=expand_log(log(p*q**2))
a
```

```
Out [39]:
```

$$\log(p) + 2 \log(q)$$

```
In [40]: logcombine(a)
```

```
Out [40]:
```

$$\log(pq^2)$$

Функция `expand_power_exp` переписывает степени, показатели которых — суммы, через произведения степеней.

```
In [41]: expand_power_exp(x**(p+q))
```

```
Out [41]:
```

$$x^p x^q$$

Функция `expand_power_base` переписывает степени, основания которых — произведения, через произведения степеней.

```
In [42]: expand_power_base((x*y)**n)
```

```
Out [42]:
```

$$x^n y^n$$

Функция `powsimp` выполняет обратные преобразования.

```
In [43]: powsimp(exp(x)*exp(2*y)),powsimp(x**n*y**n)
```

```
Out [43]:
```

$$(e^{x+2y}, (xy)^n)$$

Можно вводить функции пользователя. Они могут иметь произвольное число аргументов.

```
In [44]: f=Function('f')
         f(x)+f(x,y)
```

```
Out [44]:
```

$$f(x) + f(x, y)$$

2.5.3 Структура выражений

Внутреннее представление выражения — это дерево. Функция `srepr` возвращает строку, представляющую его.

```
In [45]: srepr(x+1)
```

```
Out [45]: "Add(Symbol('x'), Integer(1))"
```

```
In [46]: srepr(x-1)
```

```
Out [46]: "Add(Symbol('x'), Integer(-1))"
```

```
In [47]: srepr(x-y)
```

```
Out [47]: "Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```

```
In [48]: srepr(2*x*y/3)
```

```
Out [48]: "Mul(Rational(2, 3), Symbol('x'), Symbol('y'))"
```

```
In [49]: srepr(x/y)
```

```
Out [49]: "Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```

Вместо бинарных операций `+`, `*`, `**` и т.д. можно использовать функции `Add`, `Mul`, `Pow` и т.д.

```
In [50]: Mul(x,Pow(y,-1))==x/y
```

```
Out [50]: True
```

```
In [51]: srepr(f(x,y))
```

```
Out [51]: "Function('f')(Symbol('x'), Symbol('y'))"
```

Атрибут `func` — это функция верхнего уровня в выражении, а `args` — список её аргументов.

```
In [52]: a=2*x*y**2
         a.func
```

```
Out[52]: sympy.core.mul.Mul
```

```
In [53]: a.args
```

```
Out[53]:
```

$$(2, \quad x, \quad y^2)$$

```
In [54]: a.args[0]
```

```
Out[54]:
```

$$2$$

```
In [55]: for i in a.args:
         print(i)
```

```
2
```

```
x
```

```
y**2
```

Функция `subs` заменяет переменную на выражение.

```
In [56]: a.subs(y,2)
```

```
Out[56]:
```

$$8x$$

Она может заменить несколько переменных. Для этого ей передаётся список кортежей или словарь.

```
In [57]: a.subs([(x,pi),(y,2)])
```

```
Out[57]:
```

$$8\pi$$

```
In [58]: a.subs({x:pi,y:2})
```

```
Out[58]:
```

$$8\pi$$

Она может заменить не переменную, а подвыражение — функцию с аргументами.

```
In [59]: a=f(x)+f(y)
         a.subs(f(y),1)
```

Out [59] :

$$f(x) + 1$$

In [60] : `(2*x*y*z).subs(x*y,z)`

Out [60] :

$$2z^2$$

In [61] : `(x+x**2+x**3+x**4).subs(x**2,y)`

Out [61] :

$$x^3 + x + y^2 + y$$

Подстановки производятся последовательно. В данном случае сначала x заменился на y , получилось $y^3 + y^2$; потом в этом результате y заменилось на x .

In [62] : `a=x**2+y**3`
`a.subs([(x,y),(y,x)])`

Out [62] :

$$x^3 + x^2$$

Если написать эти подстановки в другом порядке, результат будет другим.

In [63] : `a.subs([(y,x),(x,y)])`

Out [63] :

$$y^3 + y^2$$

Но можно передать функции `subs` ключевой параметр `simultaneous=True`, тогда подстановки будут производиться одновременно. Таким образом можно, например, поменять местами x и y .

In [64] : `a.subs([(x,y),(y,x)],simultaneous=True)`

Out [64] :

$$x^3 + y^2$$

Можно заменить функцию на другую функцию.

In [65] : `g=Function('g')`
`a=f(x)+f(y)`
`a.subs(f,g)`

Out [65] :

$$g(x) + g(y)$$

Метод `replace` ищет подвыражения, соответствующие образцу (содержащему произвольные переменные), и заменяет их на заданное выражение (оно может содержать те же произвольные переменные).

```
In [66]: a=Wild('a')
         (f(x)+f(x+y)).replace(f(a),a**2)
```

Out [66] :

$$x^2 + (x + y)^2$$

```
In [67]: (f(x,x)+f(x,y)).replace(f(a,a),a**2)
```

Out [67] :

$$x^2 + f(x, y)$$

```
In [68]: a=x**2+y**2
         a.replace(x,x+1)
```

Out [68] :

$$y^2 + (x + 1)^2$$

Соответствовать образцу должно целое подвыражение, это не может быть часть сомножителей в произведении или меньшая степень в большей.

```
In [69]: a=2*x*y*z
         a.replace(x*y,z)
```

Out [69] :

$$2xyz$$

```
In [70]: (x+x**2+x**3+x**4).replace(x**2,y)
```

Out [70] :

$$x^4 + x^3 + x + y$$

2.5.4 Решение уравнений

In [71]: `a,b,c,d,e,f=symbols('a b c d e f')`

Уравнение записывается как функция `Eq` с двумя параметрами. Функция `solve` возвращает список решений.

In [72]: `solve(Eq(a*x,b),x)`

Out [72]:

$$\left[\frac{b}{a} \right]$$

Впрочем, можно передать функции `solve` просто выражение. Подразумевается уравнение, что это выражение равно 0.

In [73]: `solve(a*x+b,x)`

Out [73]:

$$\left[-\frac{b}{a} \right]$$

Квадратное уравнение имеет 2 решения.

In [74]: `solve(a*x**2+b*x+c,x)`

Out [74]:

$$\left[\frac{1}{2a} \left(-b + \sqrt{-4ac + b^2} \right), -\frac{1}{2a} \left(b + \sqrt{-4ac + b^2} \right) \right]$$

Система линейных уравнений.

In [75]: `solve([a*x+b*y-e,c*x+d*y-f],[x,y])`

Out [75]:

$$\left\{ x : \frac{-bf + de}{ad - bc}, \quad y : \frac{af - ce}{ad - bc} \right\}$$

Функция `roots` возвращает корни многочлена с их множественностями.

In [76]: `roots(x**3-3*x+2,x)`

Out [76]:

$$\{-2 : 1, \quad 1 : 2\}$$

Функция `solve_poly_system` решает систему полиномиальных уравнений, строя их базис Грёбнера.

```
In [77]: p1=x**2+y**2-1
          p2=4*x*y-1
          solve_poly_system([p1,p2],x,y)
```

Out [77]:

$$\left[\left(4 \left(-1 - \sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} \right) \sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} \left(-\sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} + 1 \right), -\sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} \right), \left(-4 \left(-1 + \sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} \right) \sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} \left(\sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} + 1 \right), \sqrt{-\frac{\sqrt{3}}{4} + \frac{1}{2}} \right) \right]$$

2.5.5 Ряды

```
In [78]: exp(x).series(x,0,5)
```

Out [78]:

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \mathcal{O}(x^5)$$

Ряд может начинаться с отрицательной степени.

```
In [79]: cot(x).series(x,n=5)
```

Out [79]:

$$\frac{1}{x} - \frac{x}{3} - \frac{x^3}{45} + \mathcal{O}(x^5)$$

И даже идти по полуцелым степеням.

```
In [80]: sqrt(x*(1-x)).series(x,n=5)
```

Out [80]:

$$\sqrt{x} - \frac{x^{\frac{3}{2}}}{2} - \frac{x^{\frac{5}{2}}}{8} - \frac{x^{\frac{7}{2}}}{16} - \frac{5x^{\frac{9}{2}}}{128} + \mathcal{O}(x^5)$$

```
In [81]: log(gamma(1+x)).series(x,n=6).rewrite(zeta)
```

Out [81]:

$$-\gamma x + \frac{\pi^2 x^2}{12} - \frac{x^3 \zeta(3)}{3} + \frac{\pi^4 x^4}{360} - \frac{x^5 \zeta(5)}{5} + \mathcal{O}(x^6)$$

Подготовим 3 ряда.

```
In [82]: sinx=series(sin(x),x,0,8)
          sinx
```

Out [82]:

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}(x^8)$$

```
In [83]: cosx=series(cos(x),x,n=8)
          cosx
```

```
Out [83]:
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \mathcal{O}(x^8)$$

```
In [84]: tanx=series(tan(x),x,n=8)
          tanx
```

```
Out [84]:
```

$$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \mathcal{O}(x^8)$$

Произведения и частные рядов не вычисляются автоматически, к ним надо применить функцию `series`.

```
In [85]: series(tanx*cosx,n=8)
```

```
Out [85]:
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}(x^8)$$

```
In [86]: series(sinx/cosx,n=8)
```

```
Out [86]:
```

$$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \mathcal{O}(x^8)$$

А этот ряд должен быть равен 1. Но поскольку `sinx` и `cosx` известны лишь с ограниченной точностью, мы получаем 1 с той же точностью.

```
In [87]: series(sinx**2+cosx**2,n=8)
```

```
Out [87]:
```

$$1 + \mathcal{O}(x^8)$$

Здесь первые члены сократились, и ответ можно получить лишь с меньшей точностью.

```
In [88]: series((1-cosx)/x**2,n=6)
```

```
Out [88]:
```

$$\frac{1}{2} - \frac{x^2}{24} + \frac{x^4}{720} + \mathcal{O}(x^6)$$

Ряды можно дифференцировать и интегрировать.

```
In [89]: diff(sinx,x)
```

Out [89] :

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \mathcal{O}(x^7)$$

In [90]: `integrate(cosx,x)`

Out [90] :

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \mathcal{O}(x^9)$$

Можно подставить ряд (если он начинается с малого члена) вместо переменной разложения в другой ряд. Вот ряды для $\sin(\tan(x))$ и $\tan(\sin(x))$.

In [91]: `st=series(sinx.subs(x,tanx),n=8)`
`st`

Out [91] :

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{55x^7}{1008} + \mathcal{O}(x^8)$$

In [92]: `ts=series(tanx.subs(x,sinx),n=8)`
`ts`

Out [92] :

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{107x^7}{5040} + \mathcal{O}(x^8)$$

In [93]: `series(ts-st,n=8)`

Out [93] :

$$\frac{x^7}{30} + \mathcal{O}(x^8)$$

В ряд нельзя подставлять численное значение переменной разложения (а значит, нельзя и строить график). Для этого нужно сначала убрать \mathcal{O} член, превратив отрезок ряда в много-член.

In [94]: `a=sinx.removeO()`

In [95]: `a.subs(x,0.1)`

Out [95] :

0.0998334166468254

2.5.6 Производные

```
In [96]: a=x*sin(x+y)
         diff(a,x)
```

Out[96]:

$$x \cos(x + y) + \sin(x + y)$$

```
In [97]: diff(a,y)
```

Out[97]:

$$x \cos(x + y)$$

Вторая производная по x и первая по y .

```
In [98]: diff(a,x,2,y)
```

Out[98]:

$$-x \cos(x + y) + 2 \sin(x + y)$$

Можно дифференцировать выражения, содержащие неопределённые функции.

```
In [99]: a=x*f(x**2)
         b=diff(a,x)
         b
```

Out[99]:

$$2x^2 \left. \frac{d}{d\xi_1} f(\xi_1) \right|_{\xi_1=x^2} + f(x^2)$$

Что это за зверь такой получился?

```
In [100]: print(b)
```

```
2*x**2*Subs(Derivative(f(_xi_1), _xi_1), (_xi_1,), (x**2,)) + f(x**2)
```

Функция `Derivative` представляет невычисленную производную. Её можно вычислить методом `doit`.

```
In [101]: a=Derivative(sin(x),x)
         Eq(a,a.doit())
```

Out[101]:

$$\frac{d}{dx} \sin(x) = \cos(x)$$

2.5.7 Интегралы

```
In [102]: integrate(1/(x*(x**2-2)**2),x)
```

```
Out[102]:
```

$$\frac{1}{4} \log(x) - \frac{1}{8} \log(x^2 - 2) - \frac{1}{4x^2 - 8}$$

```
In [103]: integrate(1/(exp(x)+1),x)
```

```
Out[103]:
```

$$x - \log(e^x + 1)$$

```
In [104]: integrate(log(x),x)
```

```
Out[104]:
```

$$x \log(x) - x$$

```
In [105]: integrate(x*sin(x),x)
```

```
Out[105]:
```

$$-x \cos(x) + \sin(x)$$

```
In [106]: integrate(x*exp(-x**2),x)
```

```
Out[106]:
```

$$-\frac{e^{-x^2}}{2}$$

```
In [107]: a=integrate(x**x,x)
          a
```

```
Out[107]:
```

$$\int x^x dx$$

Получился невычисленный интеграл.

```
In [108]: print(a)
```

```
Integral(x**x, x)
```

```
In [109]: a=Integral(sin(x),x)
          Eq(a,a.doit())
```

Out[109]:

$$\int \sin(x) dx = -\cos(x)$$

Определённые интегралы.

In [110]: `integrate(sin(x),(x,0,pi))`

Out[110]:

$$2$$

`oo` — это ∞ .

In [111]: `integrate(exp(-x**2),(x,0,oo))`

Out[111]:

$$\frac{\sqrt{\pi}}{2}$$

In [112]: `integrate(log(x)/(1-x),(x,0,1))`

Out[112]:

$$-\frac{\pi^2}{6}$$

2.5.8 Суммирование рядов

In [113]: `summation(1/n**2,(n,1,oo))`

Out[113]:

$$\frac{\pi^2}{6}$$

In [114]: `summation((-1)**n/n**2,(n,1,oo))`

Out[114]:

$$-\frac{\pi^2}{12}$$

In [115]: `summation(1/n**4,(n,1,oo))`

Out[115]:

$$\frac{\pi^4}{90}$$

Невычисленная сумма обозначается `Sum`.

In [116]: `a=Sum(x**n/factorial(n),(n,0,oo))`
`Eq(a,a.doit())`

Out[116]:

$$\sum_{n=0}^{\infty} \frac{x^n}{n!} = e^x$$

2.5.9 Пределы

In [117]: `limit((tan(sin(x))-sin(tan(x)))/x**7,x,0)`

Out [117]:

$$\frac{1}{30}$$

Ну это простой предел, считается разложением числителя и знаменателя в ряды. А вот если в $x = 0$ существенно особая точка, дело сложнее. Посчитаем односторонние пределы.

In [118]: `limit((tan(sin(x))-sin(tan(x)))/(x**7+exp(-1/x)),x,0,'+')`

Out [118]:

$$\frac{1}{30}$$

In [119]: `limit((tan(sin(x))-sin(tan(x)))/(x**7+exp(-1/x)),x,0,'-')`

Out [119]:

$$0$$

2.5.10 Дифференциальные уравнения

In [120]: `t=Symbol('t')`
`x=Function('x')`
`p=Function('p')`

Первого порядка.

In [121]: `dsolve(diff(x(t),t)+x(t),x(t))`

Out [121]:

$$x(t) = C_1 e^{-t}$$

Второго порядка.

In [122]: `dsolve(diff(x(t),t,2)+x(t),x(t))`

Out [122]:

$$x(t) = C_1 \sin(t) + C_2 \cos(t)$$

Система уравнений первого порядка.

In [123]: `dsolve((diff(x(t),t)-p(t),diff(p(t),t)+x(t)))`

Out [123]:

$$[x(t) = C_1 \sin(t) + C_2 \cos(t), \quad p(t) = C_1 \cos(t) - C_2 \sin(t)]$$

2.5.11 Линейная алгебра

In [124]: `a,b,c,d,e,f=symbols('a b c d e f')`

Матрицу можно построить из списка списков.

In [125]: `M=Matrix([a,b,c],[d,e,f])`
 М

Out [125]:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

In [126]: `M.shape`

Out [126]:

$$(2, 3)$$

Матрица-строка.

In [127]: `Matrix([1,2,3])`

Out [127]:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Матрица-столбец.

In [128]: `Matrix([1,2,3])`

Out [128]:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Можно построить матрицу из функции.

In [129]: `def g(i,j):`
 `return Rational(1,i+j+1)`
 `Matrix(3,3,g)`

Out [129]:

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

Или из неопределённой функции.

```
In [130]: g=Function('g')
          M=Matrix(3,3,g)
          M
```

```
Out[130]:
```

$$\begin{bmatrix} g(0,0) & g(0,1) & g(0,2) \\ g(1,0) & g(1,1) & g(1,2) \\ g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

```
In [131]: M[1,2]
```

```
Out[131]:
```

$$g(1,2)$$

```
In [132]: M[1,2]=0
          M
```

```
Out[132]:
```

$$\begin{bmatrix} g(0,0) & g(0,1) & g(0,2) \\ g(1,0) & g(1,1) & 0 \\ g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

```
In [133]: M[2,:]
```

```
Out[133]:
```

$$\begin{bmatrix} g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

```
In [134]: M[:,1]
```

```
Out[134]:
```

$$\begin{bmatrix} g(0,1) \\ g(1,1) \\ g(2,1) \end{bmatrix}$$

```
In [135]: M[0:2,1:3]
```

```
Out[135]:
```

$$\begin{bmatrix} g(0,1) & g(0,2) \\ g(1,1) & 0 \end{bmatrix}$$

Единичная матрица.

```
In [136]: eye(3)
```

Out[136]:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Матрица из нулей.

In [137]: `zeros(3)`

Out[137]:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

In [138]: `zeros(2,3)`

Out[138]:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Диагональная матрица.

In [139]: `diag(1,2,3)`

Out[139]:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

In [140]: `M=Matrix([[a,1],[0,a]])`
`diag(1,M,2)`

Out[140]:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & a & 1 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Операции с матрицами.

In [141]: `A=Matrix([[a,b],[c,d]])`
`B=Matrix([[1,2],[3,4]])`
`A+B`

Out[141]:

$$\begin{bmatrix} a+1 & b+2 \\ c+3 & d+4 \end{bmatrix}$$

In [142]: A*B,B*A

Out[142]:

$$\left(\begin{bmatrix} a+3b & 2a+4b \\ c+3d & 2c+4d \end{bmatrix}, \begin{bmatrix} a+2c & b+2d \\ 3a+4c & 3b+4d \end{bmatrix} \right)$$

In [143]: A*B-B*A

Out[143]:

$$\begin{bmatrix} 3b-2c & 2a+3b-2d \\ -3a-3c+3d & -3b+2c \end{bmatrix}$$

In [144]: simplify(A**(-1))

Out[144]:

$$\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

In [145]: det(A)

Out[145]:

$$ad - bc$$

Собственные значения и векторы

In [146]: x=Symbol('x',real=True)

In [147]: M=Matrix([[(1-x)**3*(3+x), 4*x*(1-x**2), -2*(1-x**2)*(3-x)],
[4*x*(1-x**2), -(1+x)**3*(3-x), 2*(1-x**2)*(3+x)],
[-2*(1-x**2)*(3-x), 2*(1-x**2)*(3+x), 16*x]])

M

Out[147]:

$$\begin{bmatrix} (-x+1)^3(x+3) & 4x(-x^2+1) & (-x+3)(2x^2-2) \\ 4x(-x^2+1) & -(x+3)(x+1)^3 & (x+3)(-2x^2+2) \\ (-x+3)(2x^2-2) & (x+3)(-2x^2+2) & 16x \end{bmatrix}$$

In [148]: det(M)

Out[148]:

$$0$$

Значит, у этой матрицы есть нулевое подпространство (она обращает векторы из этого подпространства в 0). Базис этого подпространства.

```
In [149]: v=M.nullspace()
          len(v)
```

```
Out[149]:
```

1

Оно одномерно.

```
In [150]: v=simplify(v[0])
          v
```

```
Out[150]:
```

$$\begin{bmatrix} -\frac{2}{x-1} \\ \frac{2}{x+1} \\ 1 \end{bmatrix}$$

Проверим.

```
In [151]: simplify(M*v)
```

```
Out[151]:
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Собственные значения и их кратности.

```
In [152]: M.eigenvals()
```

```
Out[152]:
```

$$\left\{ 0 : 1, \quad -(x^2 + 3)^2 : 1, \quad (x^2 + 3)^2 : 1 \right\}$$

Если нужны не только собственные значения, но и собственные векторы, то нужно использовать метод `eigenvects`. Он возвращает список кортежей. В каждом из них нулевой элемент — собственное значение, первый — его кратность, и последний — список собственных векторов, образующих базис (их столько, какова кратность).

```
In [153]: v=M.eigenvects()
          len(v)
```

```
Out[153]:
```

3

```
In [154]: for i in range(len(v)):
          v[i][2][0]=simplify(v[i][2][0])
          v
```

Out [154]:

$$\left[\left(0, 1, \begin{bmatrix} -\frac{2}{x-1} \\ \frac{2}{x+1} \\ 1 \end{bmatrix} \right), \left(-(x^2+3)^2, 1, \begin{bmatrix} \frac{x}{2} + \frac{1}{2} \\ \frac{x+1}{x-1} \\ 1 \end{bmatrix} \right), \left((x^2+3)^2, 1, \begin{bmatrix} \frac{x-1}{x+1} \\ -\frac{x}{2} + \frac{1}{2} \\ 1 \end{bmatrix} \right) \right]$$

Проверим.

```
In [155]: for i in range(len(v)):
           z=M*v[i][2][0]-v[i][0]*v[i][2][0]
           pprint(simplify(z))
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Жорданова нормальная форма

```
In [156]: M=Matrix([[Rational(13,9),-Rational(2,9),Rational(1,3),Rational(4,9),Rational(2,3)],
                    [-Rational(2,9),Rational(10,9),Rational(2,15),-Rational(2,9),-Rational(11,15)],
                    [Rational(1,5),-Rational(2,5),Rational(41,25),-Rational(2,5),Rational(12,25)],
                    [Rational(4,9),-Rational(2,9),Rational(14,15),Rational(13,9),-Rational(2,15)],
                    [-Rational(4,15),Rational(8,15),Rational(12,25),Rational(8,15),Rational(34,25)]])
M
```

Out [156]:

$$\begin{bmatrix} \frac{13}{9} & -\frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \frac{2}{3} \\ -\frac{2}{9} & \frac{10}{9} & \frac{2}{15} & -\frac{2}{9} & -\frac{11}{15} \\ \frac{1}{5} & -\frac{2}{5} & \frac{41}{25} & -\frac{2}{5} & \frac{12}{25} \\ \frac{4}{9} & -\frac{2}{9} & \frac{14}{15} & \frac{13}{9} & -\frac{2}{15} \\ -\frac{4}{15} & \frac{8}{15} & \frac{12}{25} & \frac{8}{15} & \frac{34}{25} \end{bmatrix}$$

Метод `M.jordan_form()` возвращает пару матриц, матрицу преобразования P и собственно жорданову форму J : $M = PJP^{-1}$.

```
In [157]: P,J=M.jordan_form()
J
```

Out [157]:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1-i & 0 \\ 0 & 0 & 0 & 0 & 1+i \end{bmatrix}$$

```
In [158]: P=simplify(P)
          P
```

```
Out[158]:
```

$$\begin{bmatrix} -2 & \frac{10}{9} & 0 & \frac{5i}{12} & -\frac{5i}{12} \\ -2 & -\frac{5}{9} & 0 & -\frac{5i}{6} & \frac{5i}{6} \\ 0 & 0 & \frac{4}{3} & -\frac{4}{3} & -\frac{3}{4} \\ 1 & \frac{10}{9} & 0 & -\frac{5i}{6} & \frac{5i}{6} \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Проверим.

```
In [159]: Z=P*J*P**(-1)-M
          simplify(Z)
```

```
Out[159]:
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

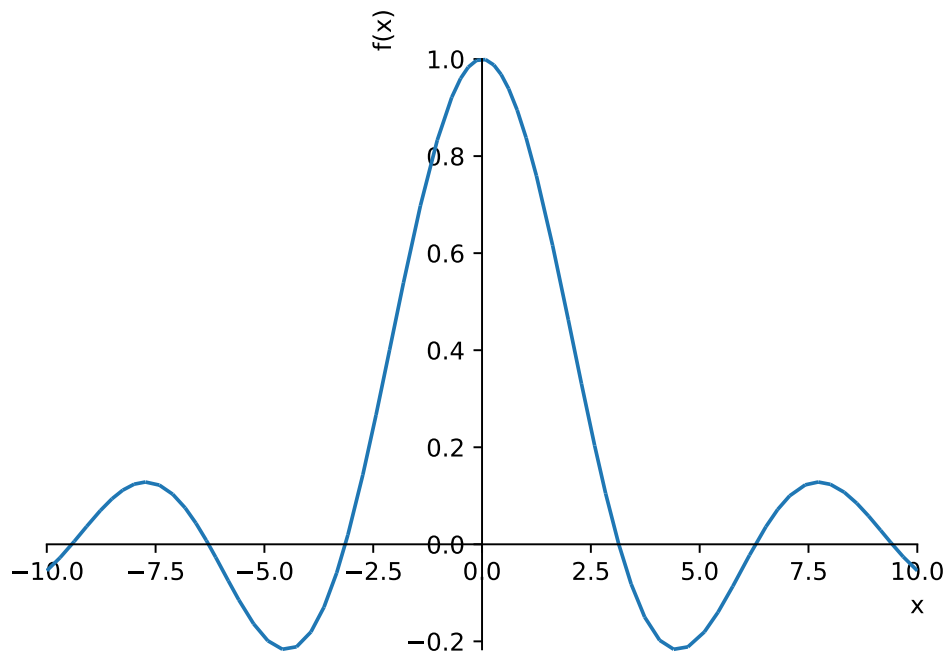
2.5.12 Графики

SymPy использует `matplotlib`. Однако он распределяет точки по x адаптивно, а не равномерно.

```
In [160]: %matplotlib inline
```

Одна функция.

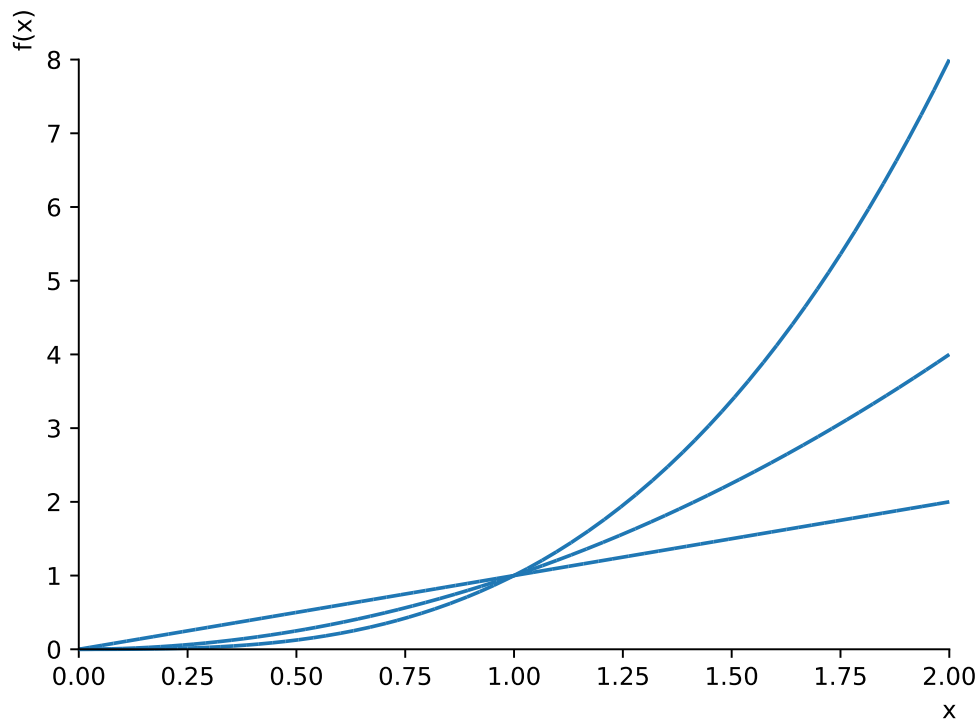
```
In [161]: plot(sin(x)/x,(x,-10,10))
```



Out[161]: <sympy.plotting.plot.Plot at 0x7fac5f8ff6d8>

Несколько функций.

In [162]: `plot(x,x**2,x**3,(x,0,2))`



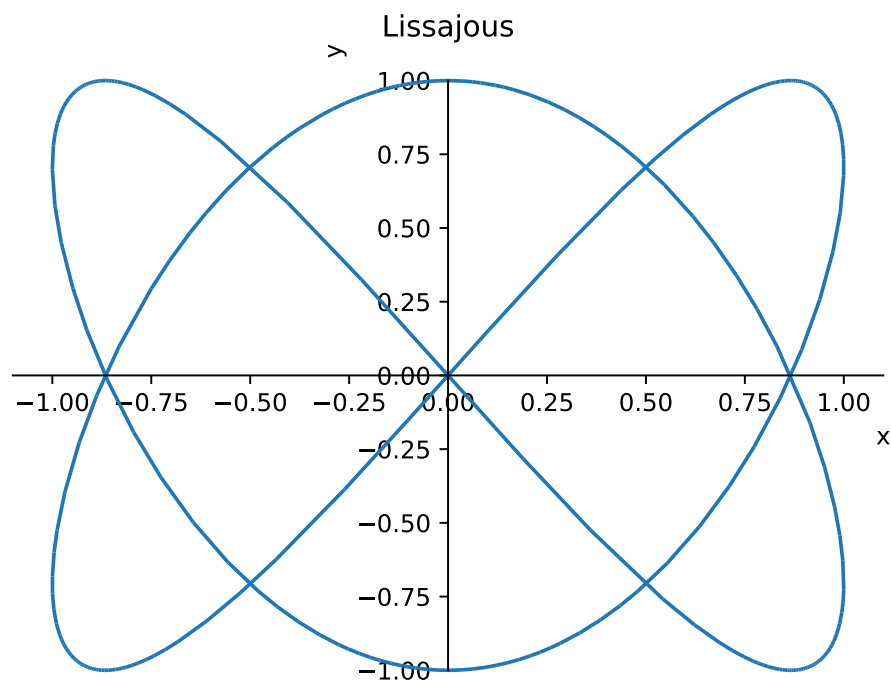
Out[162]: <sympy.plotting.plot.Plot at 0x7fac5f567390>

Другие функции надо импортировать из пакета `sympy.plotting`.

```
In [163]: from sympy.plotting import (plot_parametric, plot_implicit,
                                         plot3d, plot3d_parametric_line,
                                         plot3d_parametric_surface)
```

Параметрический график — фигура Лиссажу.

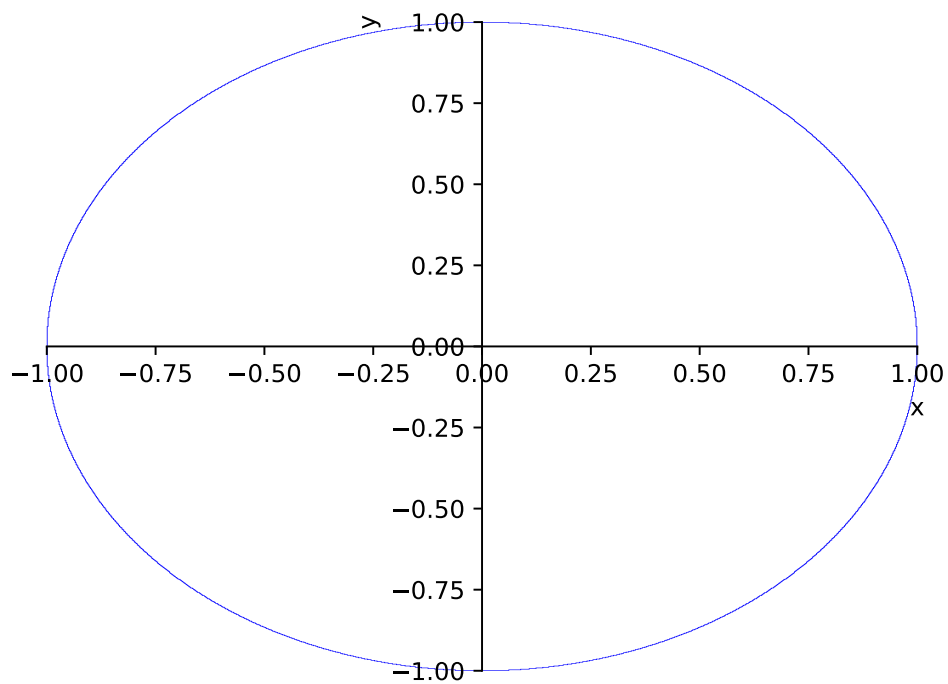
```
In [164]: t=Symbol('t')
           plot_parametric(sin(2*t), cos(3*t), (t, 0, 2*pi),
                           title='Lissajous', xlabel='x', ylabel='y')
```



Out[164]: <sympy.plotting.plot.Plot at 0x7fac5f514978>

Неявный график — окружность.

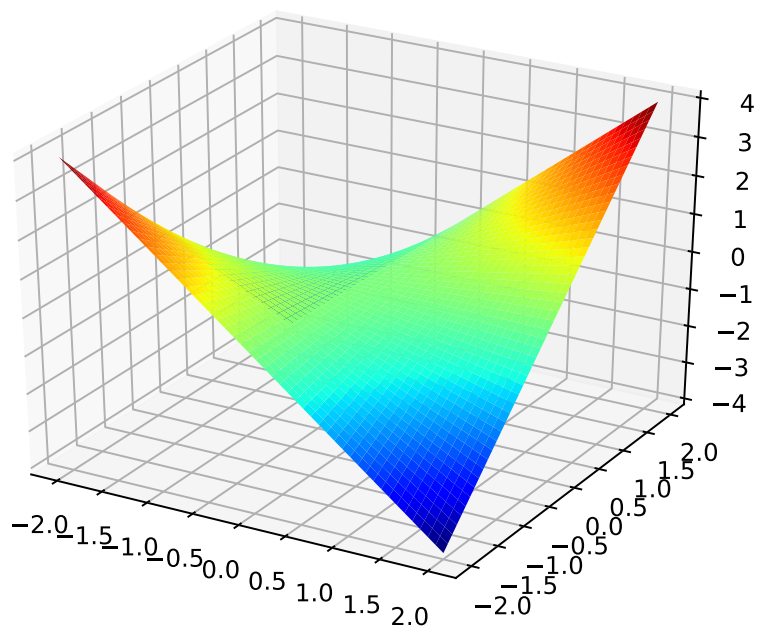
In [165]: `plot_implicit(x**2+y**2-1,(x,-1,1),(y,-1,1))`



```
Out[165]: <sympy.plotting.plot.Plot at 0x7fac5f625940>
```

Поверхность. Если она строится не `inline`, а в отдельном окне, то её можно вертеть мышкой.

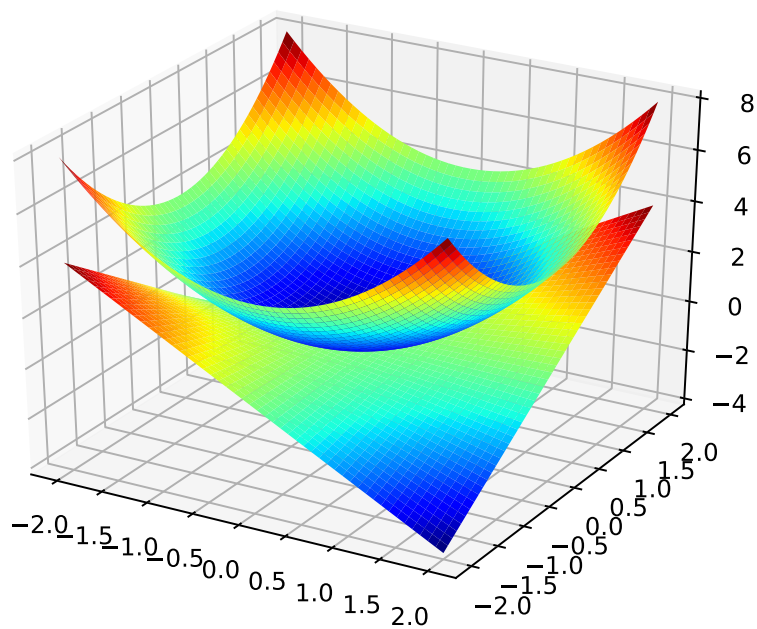
```
In [166]: plot3d(x*y,(x,-2,2),(y,-2,2))
```



```
Out[166]: <sympy.plotting.plot.Plot at 0x7fac71242ac8>
```

Несколько поверхностей.

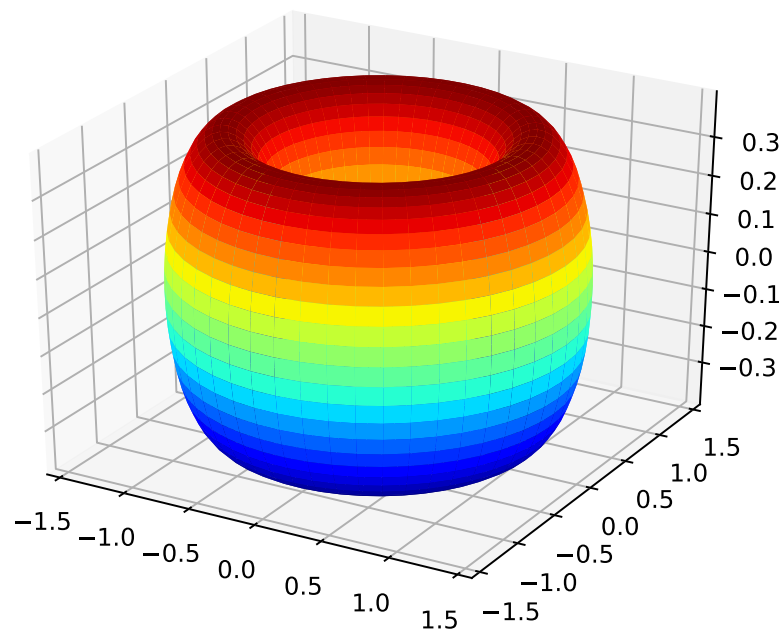
```
In [167]: plot3d(x**2+y**2,x*y,(x,-2,2),(y,-2,2))
```



Out[167]: <sympy.plotting.plot.Plot at 0x7fac5f3ac9b0>

Параметрическая пространственная линия — спираль.

```
In [168]: a=0.1  
plot3d_parametric_line(cos(t),sin(t),a*t,(t,0,4*pi))
```

Out[169]: <sympy.plotting.plot.Plot at 0x7fac5d151f60>

2.6 IMinuit

Minuit — программа численной минимизации функций многих переменных, широко применяемая в физике элементарных частиц. Есть два питонских интерфейса, PyMinuit и IMinuit (он особенно удобен в ipython).

```
In [1]: from iminuit import Minuit
        %pylab inline
```

Populating the interactive namespace from numpy and matplotlib

2.6.1 Простой пример

Определим квадратичную функцию от двух параметров.

```
In [2]: def f(a,b):
        return 10*a**2+10*b**2-16*a*b+12*a-24*b
```

Создадим объект класса Minuit. *a* и *b* — грубые догадки, около чего надо искать минимум; *error_a* и *error_b* — оценки точности этих догадок (в начале минимизации программа будет делать шаги порядка этих величин, потом они будут уменьшаться). Пределы изменения задавать не обязательно. Валичина *errordef* показывает, насколько функция должна быть выше своего минимума, чтобы это считалось отклонением на одну сигму; поскольку минимизируемая функция — это, как правило, χ^2 , значение 1 по умолчанию вполне годится.

```
In [3]: m=Minuit(f,a=0,error_a=1,limit_a=(-10,10),
                b=0,error_b=1,limit_b=(-10,10))
```

```
/usr/lib64/python3.6/site-packages/ipykernel_launcher.py:2: InitialParamWarning: errordef is not
```

Наиболее популярный метод минимизации — *migrad*.

```
In [4]: m.migrad()
```

```
Out[4]: ({'fval': -17.999997281186666, 'edm': 2.7187527367825896e-06, 'nfcn': 36, 'up': 1.0, 'is
        [{ 'number': 0, 'name': 'a', 'value': 0.9998227792225478, 'error': 0.5267876810263843, '
        { 'number': 1, 'name': 'b', 'value': 1.9993477581584944, 'error': 0.5267762744877214, '

```

Значения параметров.

```
In [5]: m.values
```

```
Out[5]: {'a': 0.9998227792225478, 'b': 1.9993477581584944}
```

Значение функции в точке минимума.

```
In [6]: m.fval
```



```
Out[6]: -17.999997281186666
```

Ошибки параметров.

```
In [7]: m.errors
```

```
Out[7]: {'a': 0.5267876810263843, 'b': 0.5267762744877214}
```

Если, скажем, a — наш окончательный физический результат, то мы напомним в статье $a = 1 \pm 0.5$. На самом деле у нас есть больше информации, поскольку ошибки a и b сильно скоррелированы. Матрица корреляции ошибок:

```
In [8]: m.matrix()
```

```
Out[8]: ((0.27776493841711364, 0.22220727646667665),
         (0.22220727646667665, 0.27776101882046583))
```

Минимизация квадратичной формы сводится к решению системы линейных уравнений, а матрица корреляции ошибок — обратная матрица этой системы. В таком простом случае не имеет смысла использовать инструмент минимизации произвольных функций, такой, как Minuit.

```
In [9]: M=array([[10., -8.], [-8., 10.]])
        M=inv(M)
        M
```

```
Out[9]: array([[ 0.27777778,  0.22222222],
               [ 0.22222222,  0.27777778]])
```

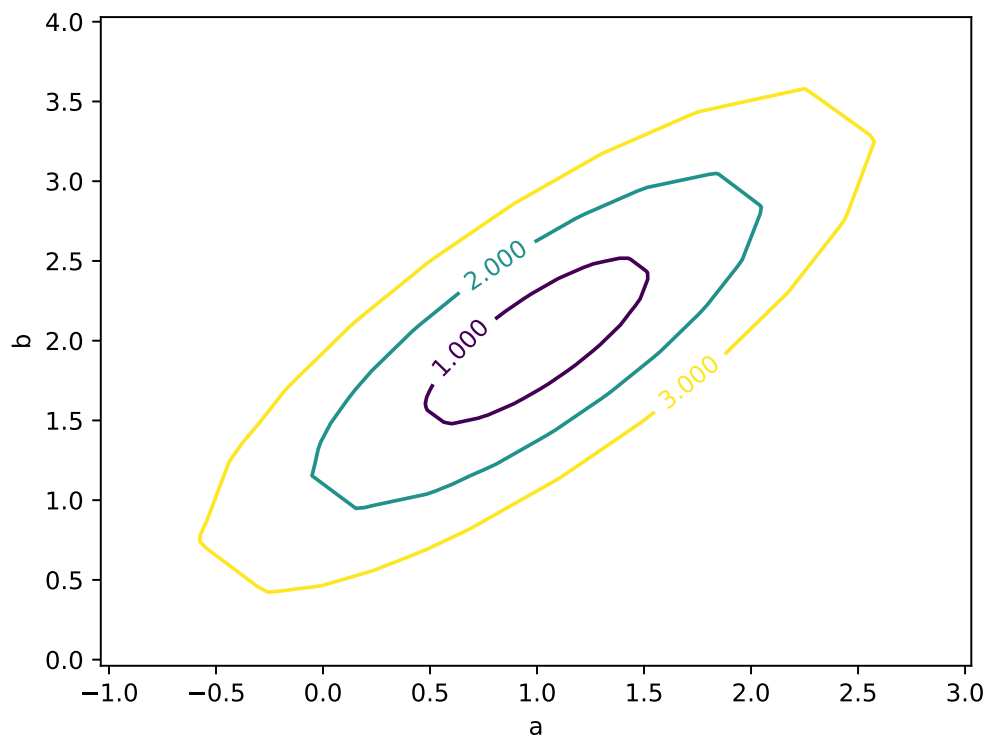
```
In [10]: M@array([[-6],[12]])
```

```
Out[10]: array([[ 1.],
               [ 2.]])
```

Нарисуем контуры, соответствующие отклонению на 1, 2 и 3 сигмы от оптимальной точки.

```
In [11]: m.draw_mncontour('a','b',nsigma=3)
```

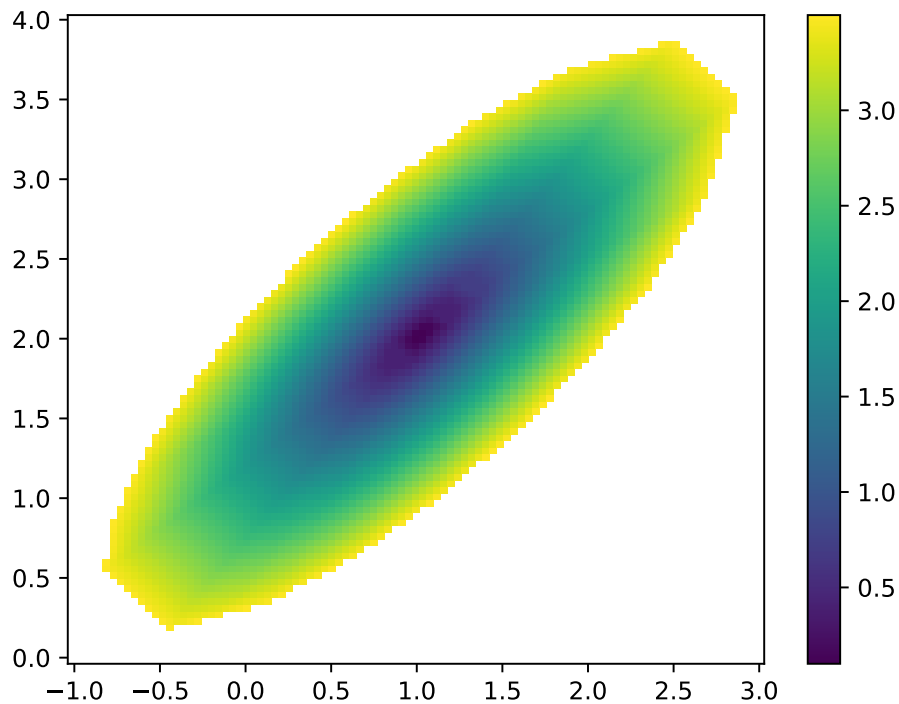
```
Out[11]: <matplotlib.contour.QuadContourSet at 0x7fb4cc2e0908>
```



То же в виде цветов.

```
In [12]: a,b,g,r=m.mncontour_grid('a','b',nsigma=3)
          pcolormesh(a,b,g)
          colorbar()
```

```
Out[12]: <matplotlib.colorbar.Colorbar at 0x7fb4cc204160>
```



2.6.2 Дайте мне 3 параметра, и я профитирую слона. С 4 параметрами он будет махать хоботом.

Пусть у нас есть экспериментальные данные, и мы хотим профитировать их прямой.

```
In [13]: def fit(a,b,x):
         return a*x+b
```

Данные не настоящие, а сгенерированные. Все имеют ошибки 0.1.

```
In [14]: x=linspace(0,1,11)
         dy=0.1*ones(11)
         y=x+dy*normal(size=11)
```

Функция χ^2 .

```
In [15]: def chi2(a,b):
         global x,y,dy
         return ((y-fit(a,b,x))/dy)**2).sum()
```

Минимизируем.

```
In [16]: m=Minuit(chi2,a=0,b=0,error_a=1,error_b=1)
```

```
/usr/lib64/python3.6/site-packages/ipykernel_launcher.py:1: InitialParamWarning: errordef is not
    """Entry point for launching an IPython kernel.
```

```
In [17]: m.migrad()
```

```
Out[12]: ({'fval': 6.64280758124837, 'edm': 6.94531024194041e-23, 'nfcn': 32, 'up': 1.0, 'is_val
    [{'number': 0, 'name': 'a', 'value': 0.947405656327807, 'error': 0.0953462580730457, '
    {'number': 1, 'name': 'b', 'value': 0.02158642903770963, 'error': 0.05640760704592273
```

```
In [18]: m.values
```

```
Out[13]: {'a': 0.947405656327807, 'b': 0.02158642903770963}
```

```
In [19]: m.fval
```

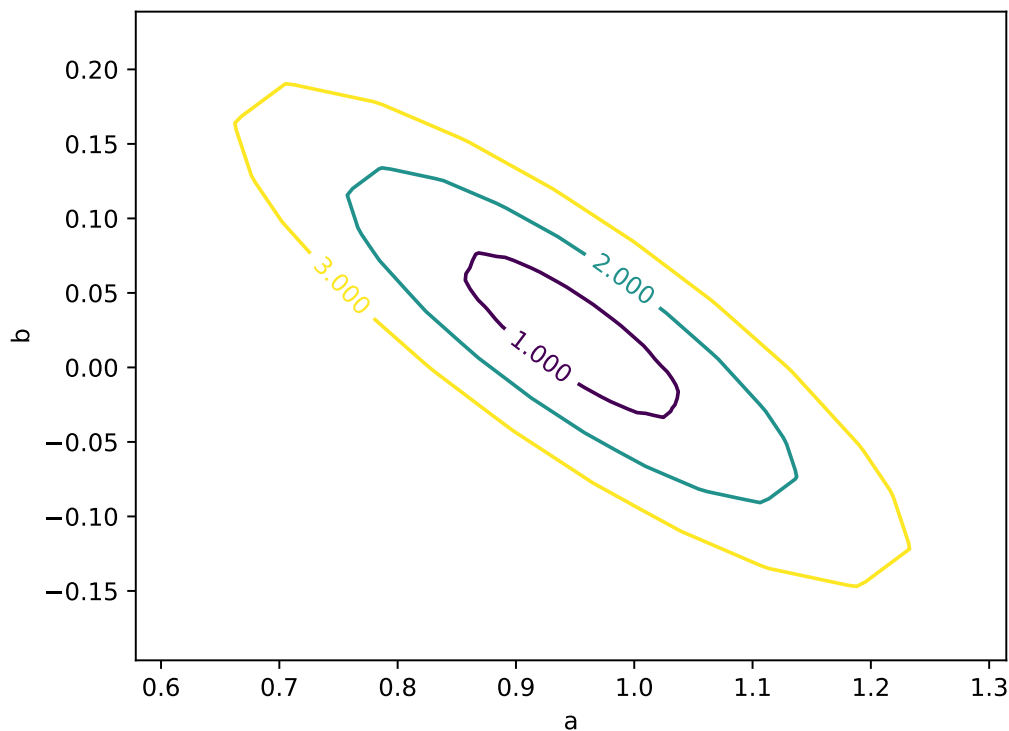
```
Out[14]: 6.64280758124837
```

```
In [20]: m.matrix()
```

```
Out[15]: ((0.009090908928531831, -0.004545454457026877),
    (-0.004545454457026877, 0.0031818181326472316))
```

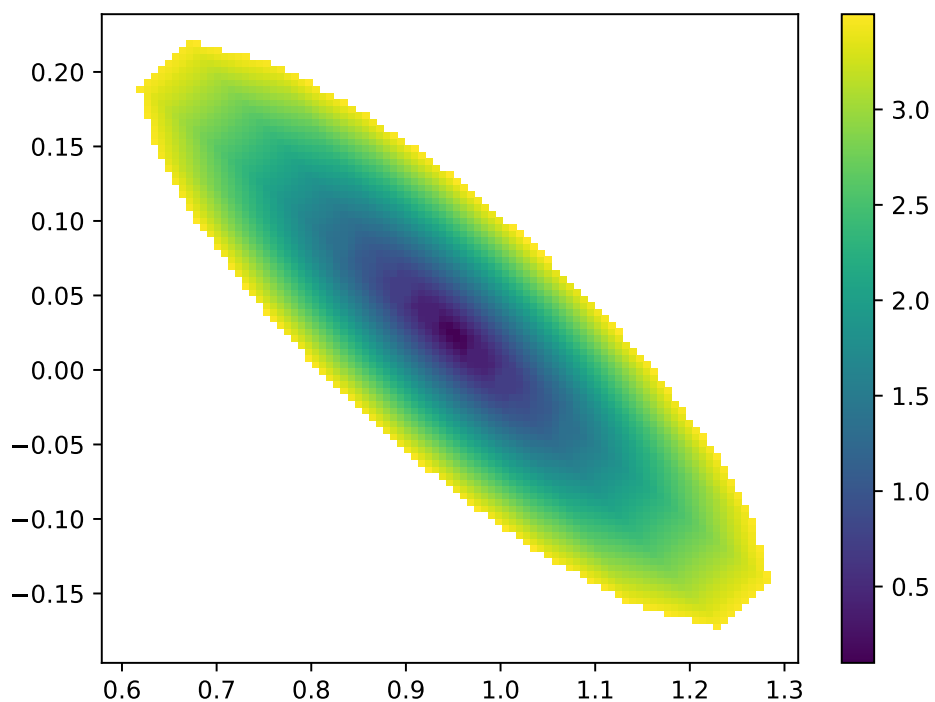
```
In [21]: m.draw_mncontour('a','b',nsigma=3)
```

```
Out[21]: <matplotlib.contour.QuadContourSet at 0x7fb4cc0ab860>
```



```
In [22]: a,b,g,r=m.mncontour_grid('a','b',nsigma=3)
          pcolormesh(a,b,g)
          colorbar()
```

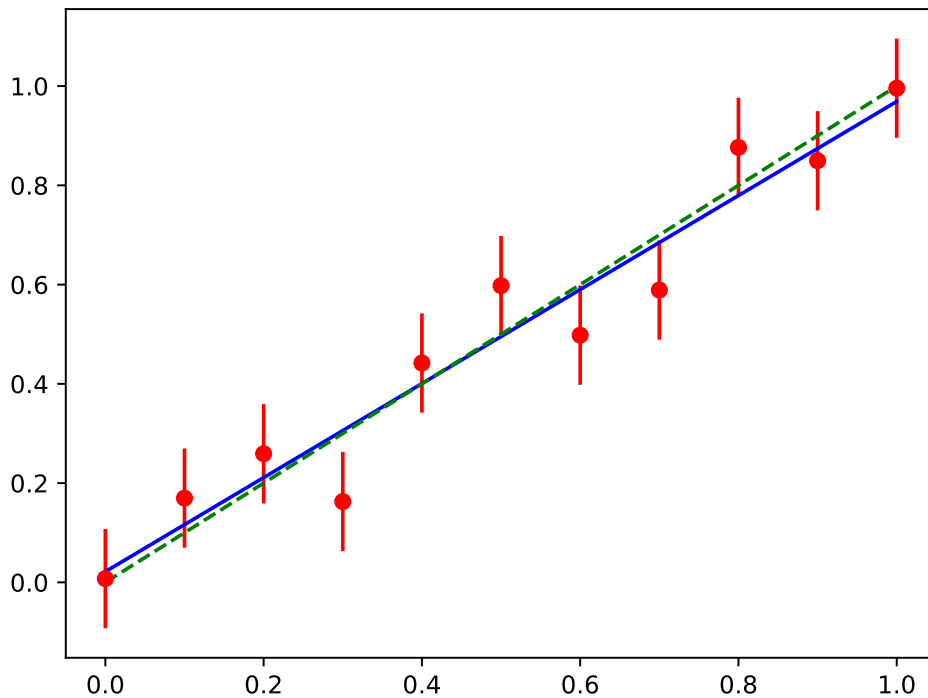
```
Out[22]: <matplotlib.colorbar.Colorbar at 0x7fb4cc127da0>
```



Нарисуем на одном графике экспериментальные точки, наш фит (сплошная линия) и истинную теоретическую кривую (пунктир).

```
In [23]: errorbar(x,y,dy,fmt='ro')
          xt=linspace(0,1,101)
          plot(xt,fit(m.values['a'],m.values['b'],xt),'b-')
          plot(xt,fit(1,0,xt),'g--')
```

```
Out[23]: [<matplotlib.lines.Line2D at 0x7fb4cbf076a0>]
```



Когда фитирующая функция есть линейная комбинация каких-то фиксированных функций с неизвестными коэффициентами, минимизация χ^2 сводится к решению системы линейных уравнений. Нет надобности использовать Minuit.

2.6.3 Резонанс без фона

Пусть теперь наша фитирующая функция — Брейт-Вигнеровский резонанс (без фона), с двумя параметрами — положением и шириной (лучше бы ввести третий — высоту, но я не стал этого делать для простоты). Теперь χ^2 — сложная нелинейная функция параметров.

```
In [24]: def fit(x0,Gamma,x):
         return 1/((x-x0)**2+Gamma**2)
```

Вот наши экспериментальные данные (с ошибками 0.1).

```
In [25]: x=linspace(-3,3,21)
         dy=0.1*ones(21)
         y=fit(0,1,x)+dy*normal(size=21)
```

Минимизируем χ^2 .

```
In [26]: def chi2(x0,Gamma):
         global x,y,dy
         return (((y-fit(x0,Gamma,x))/dy)**2).sum()
```

```
In [27]: m=Minuit(chi2,x0=0,error_x0=1,Gamma=1,error_Gamma=1)
```

```
/usr/lib64/python3.6/site-packages/ipykernel_launcher.py:1: InitialParamWarning: errordef is not
    """Entry point for launching an IPython kernel.
```

```
In [28]: m.migrad()
```

```
Out [23]: ({'fval': 25.075929198875393, 'edm': 8.303934975885262e-07, 'nfcn': 31, 'up': 1.0, 'is_
    [{'number': 0, 'name': 'x0', 'value': -0.0934806358702056, 'error': 0.0666333682582053
    {'number': 1, 'name': 'Gamma', 'value': 0.9965125441764586, 'error': 0.02701915613581
```

```
In [29]: m.values
```

```
Out [24]: {'Gamma': 0.9965125441764586, 'x0': -0.0934806358702056}
```

```
In [30]: m.fval
```

```
Out [25]: 25.075929198875393
```

```
In [31]: m.errors
```

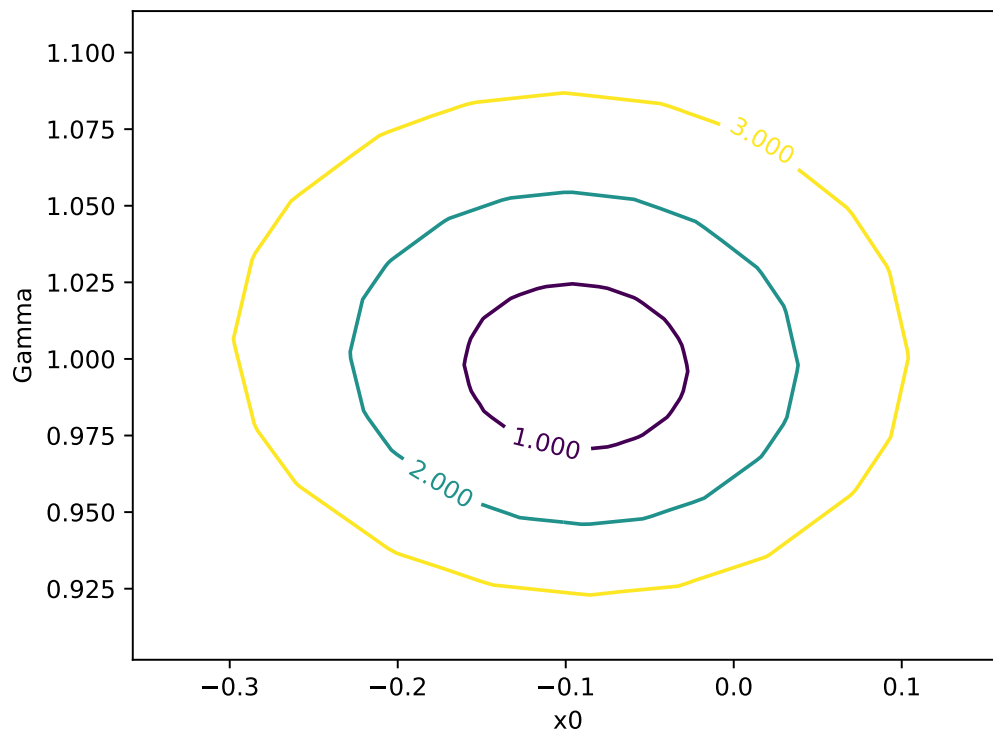
```
Out [26]: {'Gamma': 0.02701915613581351, 'x0': 0.06663336825820534}
```

```
In [32]: m.matrix()
```

```
Out [27]: ((0.0044400057654336075, -6.554599873966115e-05),
    (-6.554599873966115e-05, 0.0007300347982914688))
```

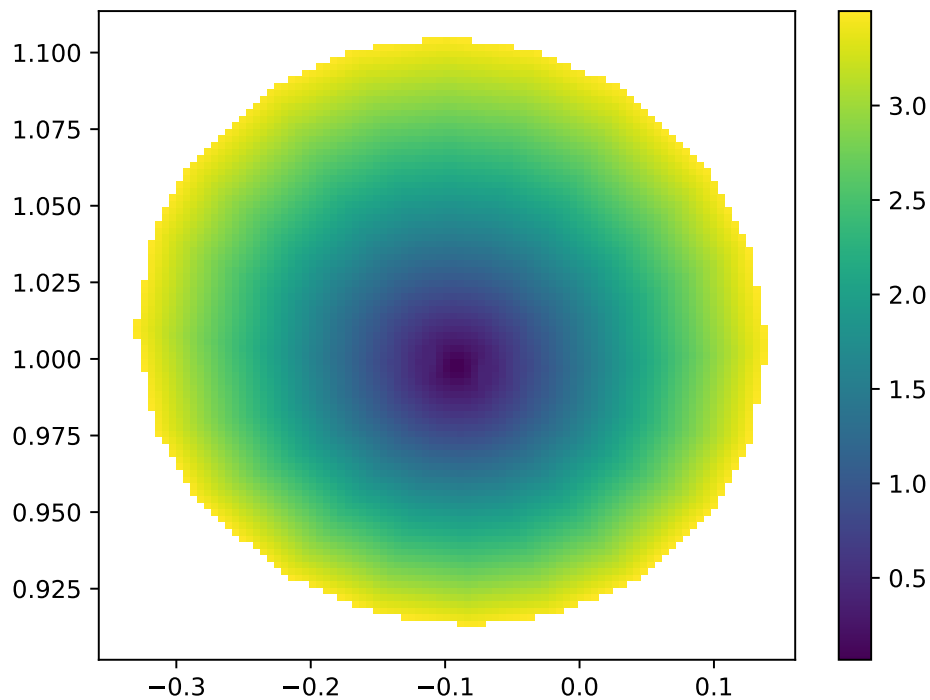
```
In [33]: m.draw_mncontour('x0', 'Gamma', nsigma=3)
```

```
Out [33]: <matplotlib.contour.QuadContourSet at 0x7fb4cbe2b4e0>
```



```
In [34]: x0,Gamma,g,r=m.mncontour_grid('x0','Gamma',nsigma=3)
         pcolormesh(x0,Gamma,g)
         colorbar()
```

```
Out [34]: <matplotlib.colorbar.Colorbar at 0x7fb4cbf0da90>
```

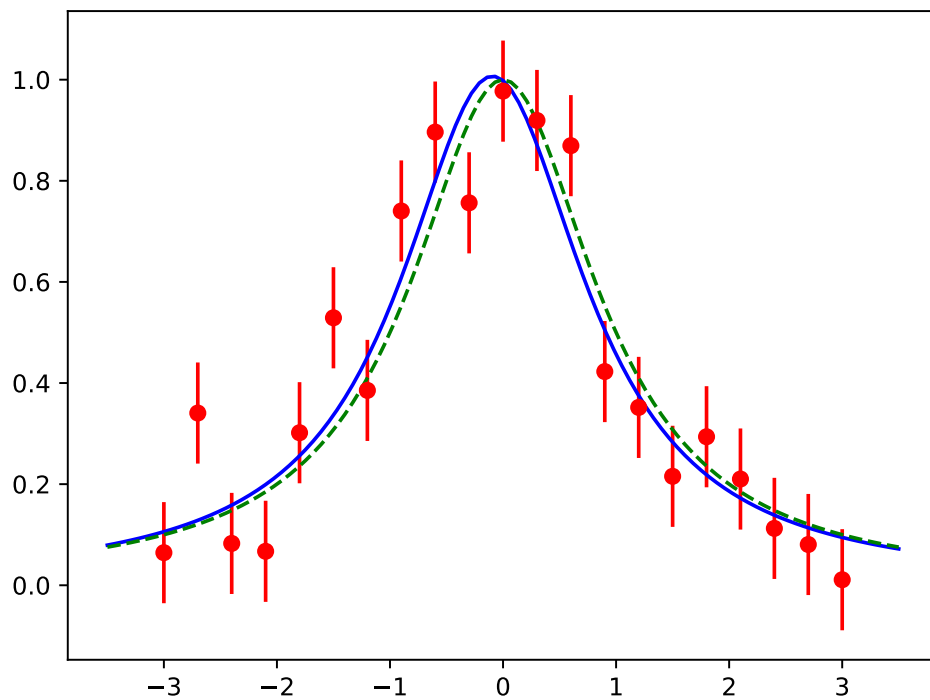



Теперь контуры постоянной высоты χ^2 — уже не симметричные эллипсы с центром в оптимальной точке, а какие-то сложные кривые. Ошибки положения и ширины резонанса довольно-таки независимы.

Нарисуем на одном графике экспериментальные точки, наш фит (сплошная линия) и истинную теоретическую кривую (пунктир).

```
In [35]: errorbar(x,y,dy,fmt='ro')
         xt=linspace(-3.5,3.5,101)
         plot(xt,fit(m.values['x0'],m.values['Gamma'],xt),'b-')
         plot(xt,fit(0,1,xt),'g--')
```

```
Out [35]: [<matplotlib.lines.Line2D at 0x7fb4cbc7b828>]
```



2.7 sh, или как писать shell-скрипты на питоне

Очень простой и удобный пакет для запуска внешних программ из питона.

```
In [1]: import sh
```

Вызовем команду `ls`. Это не значит, что в модуле `sh` есть 100500 функций, соответствующих всем командам; это было бы невозможно. Модуль `sh` производит тёмную магию, переопределяющую вызов `sh.something` в вызов внешней программы.

```
In [2]: sh.ls()
```

```
Out[2]: C1.pyx          foo.c          poster          python8.ipynb
        C2.pyx          foo.o          python.png      root.ipynb
        C3.pyx          foo.pxd        python0.png     rpyc.html
        Untitled.ipynb   foo.pyx        python1.html    rpyc.ipynb
        Untitled1.ipynb  foo.so         python1.ipynb   rpyc_old.ipynb
        Untitled2.ipynb  google-python-exercises  python2.html    sh.ipynb
        Untitled3.ipynb  ind.gle        python2.ipynb   sympy.html
        Zskim.root       ind.png        python3.html    sympy.ipynb
        __pycache__      iterators_generators.ipynb  python3.ipynb   tasks
        cfib.c           minuit.html     python4.html    text.txt
        cfib.h           minuit.ipynb    python4.ipynb   text2.txt
        cfib.o           mpmath.html     python5.html    wrap.c
        cfoo.c           mpmath.ipynb    python5.ipynb   wrap.o
        cfoo.h           newtext.txt     python6.html    wrap.pyx
        cfoo.o           osc.ipynb       python6.ipynb   wrap.so
        d1               p1             python7.html
        du               pandas.html     python7.ipynb
        fac.py           pandas.ipynb    python8.html
```

Результат можно присвоить строковой переменной.

```
In [3]: s=sh.ls()
        s.split()
```

```
Out[3]: ['C1.pyx',
        'foo.c',
        'poster',
        'python8.ipynb',
        'C2.pyx',
        'foo.o',
        'python.png',
        'root.ipynb',
        'C3.pyx',
        'foo.pxd',
        'python0.png',
        'rpyc.html',
        'Untitled.ipynb',
```

```
'foo.pyx',
'python1.html',
'rpyc.ipynb',
'Untitled1.ipynb',
'foo.so',
'python1.ipynb',
'rpyc_old.ipynb',
'Untitled2.ipynb',
'google-python-exercises',
'python2.html',
'sh.ipynb',
'Untitled3.ipynb',
'ind.gle',
'python2.ipynb',
'sympy.html',
'Zskim.root',
'ind.png',
'python3.html',
'sympy.ipynb',
'__pycache__',
'iterators_generators.ipynb',
'python3.ipynb',
'tasks',
'cfib.c',
'minuit.html',
'python4.html',
'text.txt',
'cfib.h',
'minuit.ipynb',
'python4.ipynb',
'text2.txt',
'cfib.o',
'mpmath.html',
'python5.html',
'wrap.c',
'cfoo.c',
'mpmath.ipynb',
'python5.ipynb',
'wrap.o',
'cfoo.h',
'newtext.txt',
'python6.html',
'wrap.pyx',
'cfoo.o',
'osc.ipynb',
'python6.ipynb',
'wrap.so',
'd1',
```

```
'p1',
'python7.html',
'du',
'pandas.html',
'python7.ipynb',
'fac.py',
'pandas.ipynb',
'python8.html']
```

Командам можно передавать параметры.

```
In [4]: sh.ls('d1')
```

```
Out[4]: __pycache__  d2  m1.py
```

Допустим, мы хотим вызывать команду `ls` много раз. `sh` также производит тёмную магию, переопределяющую `from sh import something`, так что после этого импорта `ls` становится полноправной питонской функцией (вызывающей внешнюю программу).

```
In [5]: from sh import ls
```

```
In [6]: s=ls('d1')
        s.split()
```

```
Out[6]: ['__pycache__', 'd2', 'm1.py']
```

```
In [7]: ls('-l', 'd1')
```

```
Out[7]: итого 12
        drwxr-xr-x 2 grozin grozin 4096 ноя  4  2015 __pycache__
        drwxr-xr-x 3 grozin grozin 4096 ноя  4  2015 d2
        -rw-r--r-- 1 grozin grozin   23 ноя  4  2015 m1.py
```

Разумеется, таким же образом Вы можете вызвать Вашу программу (написанную на любом языке).

2.8 plumbum

Другой пакет для написания shell-скриптов.

2.8.1 Работа на локальной машине

```
In [1]: from plumbum import local
```

Из объекта `local` можно получать питонские объекты, представляющие внешние программы.

```
In [2]: ls=local["ls"]  
        print(ls)
```

```
/bin/ls
```

Их можно вызывать.

```
In [3]: print(ls())
```

```
C1.pyx  
C2.pyx  
C3.pyx  
Untitled.ipynb  
Untitled1.ipynb  
Untitled2.ipynb  
Untitled3.ipynb  
Zskim.root  
__pycache__  
cfib.c  
cfib.h  
cfib.o  
cfoo.c  
cfoo.h  
cfoo.o  
d1  
du  
fac.py  
foo.c  
foo.o  
foo.pxd  
foo.pyx  
foo.so  
google-python-exercises  
ind.gle  
ind.png  
iterators_generators.ipynb  
minuit.html
```

```
minuit.ipynb
mpmath.html
mpmath.ipynb
newtext.txt
osc.ipynb
p1
pandas.html
pandas.ipynb
poster
python.png
python0.png
python1.html
python1.ipynb
python2.html
python2.ipynb
python3.html
python3.ipynb
python4.html
python4.ipynb
python5.html
python5.ipynb
python6.html
python6.ipynb
python7.html
python7.ipynb
python8.html
python8.ipynb
root.ipynb
rpyc.html
rpyc.ipynb
rpyc_old.ipynb
sh.ipynb
sympy.html
sympy.ipynb
tasks
text.txt
text2.txt
wrap.c
wrap.o
wrap.pyx
wrap.so
```

Они возвращают строки, которые можна присваивать переменным или ещё как-то использовать.

```
In [4]: s=ls()
        s.split()
```

```
Out[4]: ['C1.pyx',
         'C2.pyx',
         'C3.pyx',
         'Untitled.ipynb',
         'Untitled1.ipynb',
         'Untitled2.ipynb',
         'Untitled3.ipynb',
         'Zskim.root',
         '__pycache__',
         'cfib.c',
         'cfib.h',
         'cfib.o',
         'cfoo.c',
         'cfoo.h',
         'cfoo.o',
         'd1',
         'du',
         'fac.py',
         'foo.c',
         'foo.o',
         'foo.pxd',
         'foo.pyx',
         'foo.so',
         'google-python-exercises',
         'ind.gle',
         'ind.png',
         'iterators_generators.ipynb',
         'minuit.html',
         'minuit.ipynb',
         'mpmath.html',
         'mpmath.ipynb',
         'newtext.txt',
         'osc.ipynb',
         'p1',
         'pandas.html',
         'pandas.ipynb',
         'poster',
         'python.png',
         'python0.png',
         'python1.html',
         'python1.ipynb',
         'python2.html',
         'python2.ipynb',
         'python3.html',
         'python3.ipynb',
         'python4.html',
         'python4.ipynb',
         'python5.html',
```



```
'python5.ipynb',
'python6.html',
'python6.ipynb',
'python7.html',
'python7.ipynb',
'python8.html',
'python8.ipynb',
'root.ipynb',
'rpyc.html',
'rpyc.ipynb',
'rpyc_old.ipynb',
'sh.ipynb',
'sympy.html',
'sympy.ipynb',
'tasks',
'text.txt',
'text2.txt',
'wrap.c',
'wrap.o',
'wrap.pyx',
'wrap.so']
```

Можно вызывать их с аргументами.

```
In [5]: print(ls('-l', 'd1'))
```

```
итого 12
drwxr-xr-x 2 grozin grozin 4096 ноя  4  2015 __pycache__
drwxr-xr-x 3 grozin grozin 4096 ноя  4  2015 d2
-rw-r--r-- 1 grozin grozin  23 ноя  4  2015 m1.py
```

Следующая строчка означает в точности то же самое, что

```
cat=local['cat']
grep=local['grep']
```

(модуль `plumbum.cmd` использует чёрную магию для переопределения импорта из него).

```
In [6]: from plumbum.cmd import cat,grep
```

Это объект, представляющий внешнюю программу с привязанными аргументами.

```
In [7]: ll=ls['-l']
        print(ll)
```

```
/bin/ls -l
```

```
In [9]: print(ll())
```

```
итого 17856
```

```
-rw-r--r-- 1 grozin grozin      290 дек 10  2015 C1.pyx
-rw-r--r-- 1 grozin grozin      208 дек 10  2015 C2.pyx
-rw-r--r-- 1 grozin grozin      310 дек 10  2015 C3.pyx
-rw-r--r-- 1 grozin grozin  18997 ноя 12  2015 Untitled.ipynb
-rw-r--r-- 1 grozin grozin   1588 ноя 29  2015 Untitled1.ipynb
-rw-r--r-- 1 grozin grozin    841 дек 13  2015 Untitled2.ipynb
-rw-r--r-- 1 grozin grozin  16752 дек 28  2015 Untitled3.ipynb
-rw-r--r-- 1 grozin grozin 9231391 дек 13  2015 Zskim.root
drwxr-xr-x 2 grozin grozin   4096 ноя  4  2015 __pycache__
-rw-r--r-- 1 grozin grozin    157 дек  6  2015 cfib.c
-rw-r--r-- 1 grozin grozin     19 дек  6  2015 cfib.h
-rw-r--r-- 1 grozin grozin   1312 ноя 16 19:08 cfib.o
-rw-r--r-- 1 grozin grozin    243 дек 10  2015 cfoo.c
-rw-r--r-- 1 grozin grozin    130 дек 10  2015 cfoo.h
-rw-r--r-- 1 grozin grozin   1752 ноя 16 19:45 cfoo.o
drwxr-xr-x 4 grozin grozin   4096 ноя  4  2015 d1
drwxr-xr-x 3 grozin grozin   4096 окт  9 21:13 du
-rwxr-xr-x 1 grozin grozin    401 ноя  4  2015 fac.py
-rw-r--r-- 1 grozin grozin  85006 ноя 16 19:45 foo.c
-rw-r--r-- 1 grozin grozin  21512 ноя 16 19:45 foo.o
-rw-r--r-- 1 grozin grozin    164 дек 10  2015 foo.pxd
-rw-r--r-- 1 grozin grozin    282 дек 10  2015 foo.pyx
-rwxr-xr-x 1 grozin grozin  24200 ноя 16 19:45 foo.so
drwxr-xr-x 6 grozin grozin   4096 сен 28 16:50 google-python-exercises
-rw-r--r-- 1 grozin grozin    630 окт 11  2015 ind.gle
-rw-r--r-- 1 grozin grozin   3945 окт 12  2015 ind.png
-rw-r--r-- 1 grozin grozin  11154 ноя  2 16:25 iterators_generators.ipynb
-rw-r--r-- 1 grozin grozin 421451 дек 25  2015 minuit.html
-rw-r--r-- 1 grozin grozin 294985 ноя  9 17:37 minuit.ipynb
-rw-r--r-- 1 grozin grozin 543403 ноя  2 20:51 mpmath.html
-rw-r--r-- 1 grozin grozin 266855 ноя  2 19:39 mpmath.ipynb
-rw-r--r-- 1 grozin grozin     15 ноя  8  2015 newtext.txt
-rw-r--r-- 1 grozin grozin  10665 дек  3  2015 osc.ipynb
drwxr-xr-x 4 grozin grozin   4096 ноя  8  2015 p1
-rw-r--r-- 1 grozin grozin 299444 дек 25  2015 pandas.html
-rw-r--r-- 1 grozin grozin 130941 ноя  2 18:51 pandas.ipynb
drwxr-xr-x 2 grozin grozin   4096 авг 29 15:48 poster
-rw-r--r-- 1 grozin grozin   67230 окт 11  2015 python.png
-rw-r--r-- 1 grozin grozin  146956 окт 10  2015 python0.png
-rw-r--r-- 1 grozin grozin  371477 сен 17 23:16 python1.html
-rw-r--r-- 1 grozin grozin   81013 сен 21 19:23 python1.ipynb
-rw-r--r-- 1 grozin grozin  373780 сен 28 16:20 python2.html
-rw-r--r-- 1 grozin grozin   68480 сен 28 22:18 python2.ipynb
-rw-r--r-- 1 grozin grozin 295939 ноя  5  2015 python3.html
-rw-r--r-- 1 grozin grozin   47236 окт 22 10:51 python3.ipynb
```

```

-rw-r--r-- 1 grozin grozin 291364 ноя 5 2015 python4.html
-rw-r--r-- 1 grozin grozin 58657 окт 12 19:46 python4.ipynb
-rw-r--r-- 1 grozin grozin 349899 дек 5 2015 python5.html
-rw-r--r-- 1 grozin grozin 101188 окт 19 19:47 python5.ipynb
-rw-r--r-- 1 grozin grozin 919499 дек 5 2015 python6.html
-rw-r--r-- 1 grozin grozin 1018131 окт 26 19:37 python6.ipynb
-rw-r--r-- 1 grozin grozin 390422 дек 19 2015 python7.html
-rw-r--r-- 1 grozin grozin 730517 ноя 16 18:48 python7.ipynb
-rw-r--r-- 1 grozin grozin 240728 дек 24 2015 python8.html
-rw-r--r-- 1 grozin grozin 30472 ноя 16 19:49 python8.ipynb
-rw-r--r-- 1 grozin grozin 16206 дек 14 2015 root.ipynb
-rw-r--r-- 1 grozin grozin 243066 дек 28 2015 rpyc.html
-rw-r--r-- 1 grozin grozin 26991 ноя 23 18:36 rpyc.ipynb
-rw-r--r-- 1 grozin grozin 23252 дек 28 2015 rpyc_old.ipynb
-rw-r--r-- 1 grozin grozin 7492 ноя 23 18:31 sh.ipynb
-rw-r--r-- 1 grozin grozin 378705 дек 25 2015 sympy.html
-rw-r--r-- 1 grozin grozin 332000 дек 25 2015 sympy.ipynb
drwxr-xr-x 3 grozin grozin 4096 дек 3 2015 tasks
-rw-r--r-- 1 grozin grozin 15 окт 11 2015 text.txt
-rw-r--r-- 1 grozin grozin 5 ноя 23 15:09 text2.txt
-rw-r--r-- 1 grozin grozin 73021 ноя 16 19:11 wrap.c
-rw-r--r-- 1 grozin grozin 16216 ноя 16 19:11 wrap.o
-rw-r--r-- 1 grozin grozin 86 дек 6 2015 wrap.pyx
-rwxr-xr-x 1 grozin grozin 19048 ноя 16 19:11 wrap.so

```

Из таких объектов можно строить цепочки.

```

In [10]: chain=ll | grep['ipynb']
         print(chain)

```

```

/bin/ls -l | /bin/grep ipynb

```

Цепочки можно вызывать.

```

In [11]: print(chain())

```

```

-rw-r--r-- 1 grozin grozin 18997 ноя 12 2015 Untitled.ipynb
-rw-r--r-- 1 grozin grozin 1588 ноя 29 2015 Untitled1.ipynb
-rw-r--r-- 1 grozin grozin 841 дек 13 2015 Untitled2.ipynb
-rw-r--r-- 1 grozin grozin 16752 дек 28 2015 Untitled3.ipynb
-rw-r--r-- 1 grozin grozin 11154 ноя 2 16:25 iterators_generators.ipynb
-rw-r--r-- 1 grozin grozin 294985 ноя 9 17:37 minuit.ipynb
-rw-r--r-- 1 grozin grozin 266855 ноя 2 19:39 mpmath.ipynb
-rw-r--r-- 1 grozin grozin 10665 дек 3 2015 osc.ipynb
-rw-r--r-- 1 grozin grozin 130941 ноя 2 18:51 pandas.ipynb
-rw-r--r-- 1 grozin grozin 81013 сен 21 19:23 python1.ipynb

```

```
-rw-r--r-- 1 grozin grozin 68480 сен 28 22:18 python2.ipynb
-rw-r--r-- 1 grozin grozin 47236 окт 22 10:51 python3.ipynb
-rw-r--r-- 1 grozin grozin 58657 окт 12 19:46 python4.ipynb
-rw-r--r-- 1 grozin grozin 101188 окт 19 19:47 python5.ipynb
-rw-r--r-- 1 grozin grozin 1018131 окт 26 19:37 python6.ipynb
-rw-r--r-- 1 grozin grozin 730517 ноя 16 18:48 python7.ipynb
-rw-r--r-- 1 grozin grozin 30472 ноя 16 19:49 python8.ipynb
-rw-r--r-- 1 grozin grozin 16206 дек 14 2015 root.ipynb
-rw-r--r-- 1 grozin grozin 32413 ноя 23 18:38 rpyc.ipynb
-rw-r--r-- 1 grozin grozin 23252 дек 28 2015 rpyc_old.ipynb
-rw-r--r-- 1 grozin grozin 7492 ноя 23 18:31 sh.ipynb
-rw-r--r-- 1 grozin grozin 332000 дек 25 2015 sympy.ipynb
```

Можно использовать перенаправления ввода-вывода.

```
In [12]: !cat newtext.txt
```

```
abcd
efgh
ijkl
```

```
In [13]: chain(grep['ab'] < 'newtext.txt') > 'text2.txt'
          print(chain)
```

```
/bin/grep ab < newtext.txt > text2.txt
```

(скобки здесь обязательны)

```
In [14]: chain()
          print(cat('text2.txt'))
```

```
abcd
```

Если нужно послать текст в `stdin` внешней программы, используется оператор `<<`.

```
In [15]: print((grep['ab'] << 'xxx\nabc\nyyy\n')())
```

```
abc
```

2.8.2 Работа на удалённой машине

Допустим, в локальной сети есть машина еееpc (192.168.0.105), доступная по `ssh`.

```
In [16]: ip='192.168.43.95'
```

```
In [17]: from plumbum import SshMachine
         eeepc=SshMachine(ip)
```

Теперь мы можем выполнять на ней команды.

```
In [18]: eeepc_ls=eeepc['ls']
         print(eeepc_ls)
```

```
/bin/ls
```

```
In [19]: print(eeepc_ls('rpyc'))
```

```
__pycache__
mymodule.py
rpyc.txt
```

Можно строить цепочки из локальных и удалённых команд.

```
In [20]: chain=eeepc_ls['rpyc'] | grep[r'\.py']
         print(chain)
```

```
/bin/ls rpyc | /bin/grep '\.py'
```

```
In [22]: chain
```

```
Out[22]: Pipeline(BoundCommand(RemoteCommand(<SshMachine ssh://192.168.43.95>, <RemotePath /bin/
```

```
In [21]: print(chain())
```

```
mymodule.py
```

```
In [23]: eeepc_grep=eeepc['grep']
         print((eeepc_grep['ab'] < 'newtext.txt')())
```

```
abcd
```

```
In [24]: print((eeepc_grep['ab'] << 'xxx\nabc\nyyy\n')())
```

```
abc
```

```
In [25]: print((cat['newtext.txt'] | eeepc_grep['ab']))
```

```
abcd
```

Теперь закроем связь с `ееерс`.

```
In [26]: eeepc.close()
```

Сеанс работы с `ееерс` удобно записать в виде

```
with SshMachine('192.168.0.105') as eeeps:
    eeepc['ls']()
    # и так далее
```

2.9 RPyC

Remote Python Call <http://rpyc.sourceforge.net/> — очень простой пакет для организации распределённых вычислений. Он работает на любой платформе, где есть питон, так что Вы можете объединить в вычислительный кластер всё, что подвернётся под руку - Linux, Windows, Mac, хоть свой телефон.

На всех компьютерах, которые мы хотим использовать, запускаются `rpyc` серверы. Клиент обращается к ним и поручает выполнить какую-нибудь работу. Есть два типа серверов — классический (или `slave`-сервер) и современный. Классический сервер может (по поручению клиента) делать всё, что может делать интерпретатор питон (от имени того пользователя, который запустил сервер). Он не производит аутентификацию клиента. Поэтому его можно запускать только в защищённой локальной сети (возможно виртуальной), или он должен принимать соединения только с `localhost` (а другие машины получают к нему доступ через `ssh` туннели). В отличие от классического сервера (который поставляется в пакете `RPyC`), современный сервер должен быть написан пользователем под конкретную задачу. Он предоставляет клиентам некоторый набор сервисов; клиенты могут вызывать их. Если эти сервисы безопасны, такой сервер может работать и в открытом интернете.

Есть более простой способ использования `RPyC`, который даже не требует, чтобы этот пакет был установлен на всех машинах — достаточно иметь его на клиентской (локальной) машине. Установим `ssh` связь с удалённой машиной.

```
In [27]: eeepc=SshMachine(ip)
```

Функция `DeployedServer` передаёт на `ееерс` нужные исходные тексты, запускает там классический `rpyc` сервер, принимающий соединения только с локальной машины, и создаёт `ssh` туннель на `ееерс`.

```
In [28]: import rpyc
          from rpyc.utils.zerodeploy import DeployedServer
          server=DeployedServer(eeepc)
```

Теперь мы можем установить связь с этим сервером.

```
In [29]: eee=server.classic_connect()
```

Теперь я могу выполнять различные действия на `ееерс`:

```
In [30]: eee.execute('n=2')
```

```
In [31]: eee.eval('n+1')
```

```
Out[31]: 3
```

Я могу использовать встроенные функции питона.

```
In [32]: eee_file=eee.builtins.open('/home/grozin/rpyc/mymodule.py')
```

В отличие от простых неизменяемых объектов (чисел, строк и т.д.), которые передаются между машинами по значению, изменяемые объекты передаются по ссылке. То есть на `ееерс` создан файл-объект; на локальной машине создана сетевая ссылка на него — прокси-объект `eee_file`. Мы можем производить над ним любые действия, доступные для файлового объекта. Все они переадресуются объекту на `ееерс`.

```
In [33]: print(eee_file.read())
```

```
#!/usr/bin/env python
from time import sleep
```

```
class MyClass:
```

```
    def __init__(self,t):
        self.t=t
```

```
    def f(self,n):
        sleep(self.t)
        return n+1
```

```
In [34]: eee_file.close()
```

Этот прокси-объект можно подставить в любую программу, ожидающую иметь файлового объект. По принципу утиной типизации этот объект — файл.

Я могу использовать функции и прочие объекты из библиотечных модулей питона на `ееерс`:

```
In [35]: eee_path=eee.modules.sys.path
         print(eee_path)
```

```
['/home/grozin/tmp.hK717huHzN', '/home/grozin/tmp.hK717huHzN', '/usr/lib/python35.zip', '/usr/li
```

```
In [36]: eee_path.append('/home/grozin/rpyc')
```

`eee_path` — это прокси-объект для `sys.path` на `ееерс`; любые изменения этого объекта сразу передаются туда. Теперь, расширив `path`, я могу использовать файл из этой директории на `ееерс`:

```
In [37]: eee_object=eee.modules.mymodule.MyClass(0)
```

```
In [38]: eee_object.f(3)
```

```
Out[38]: 4
```

`eee_object` — это прокси-объект (сетевая ссылка) для объекта класса `MyClass` на машине `ееерс`. Его метод `f` прибавляет 1 к аргументу; чтобы у нас была возможность моделировать длительные вычисления, он это делает за `t` секунд, где `t` — атрибут этого объекта.

```
In [39]: eee_object.t=2
         eee_object.f(4)
```

```
Out[39]: 5
```

Теперь нам пришлось ждать 2 секунды.

Можно передавать удалённым функциям в качестве параметров любые объекты, в частности, локальные функции. Определим

```
In [40]: def loc(n):
         print('loc',n)
         return n+1
```

Тогда

```
In [41]: list(eee.builtins.map(loc,[1,2,3]))
```

```
loc 1
loc 2
loc 3
```

```
Out[41]: [2, 3, 4]
```

То есть функция `map` на `ееерс` на каждом шаге вызывает функцию `loc` на локальной машине (callback).

Всё, что мы до сих пор обсуждали, несомненно, красиво — разные объекты могут жить на разных машинах, и единая программа работает с ними, не замечая этого. Но все эти операции синхронные — одна машина просит другую что-то сделать и ждёт, когда та вернёт ей результат. Для организации распределённых вычислений нужны асинхронные операции:

```
In [42]: eee_object.t=10
         async_f=rpyc.async(eee_object.f)
         res=async_f(1)
         res.ready
```



```
Out [42]: False
```

```
In [43]: res.ready
```

```
Out [43]: False
```

```
In [45]: res.ready
```

```
Out [45]: True
```

```
In [46]: res.value
```

```
Out [46]: 2
```

Это уже лучше. Клиент может время от времени спрашивать, готов ли результат, и когда он будет готов, забрать его. Если запросить `res.value` когда результат ещё не готов, то клиент блокируется до момента, когда он будет готов:

```
In [47]: res=async_f(2)
         res.value
```

```
Out [47]: 3
```

(после `res.value` 10 секунд ожидания, потом появляется ответ).

Но ещё лучше определить callback-функцию, которая будет вызвана на локальной машине, когда результат будет готов:

```
In [48]: def callback(res):
         print(res.value)
```

Эта функция может быть вызвана только в отдельном thread-e:

```
In [49]: thr=rpc.BgServingThread(eee)
         res=async_f(3)
         res.add_callback(callback)
```

```
In [50]: n=1
```

```
In [51]: n
```

```
Out [51]: 1
```

```
In [52]: n+1
```

```
Out [52]: 2
```

```
4
```

Это печать из функции callback из другого thread-a. Теперь это thread можно и остановить.

```
In [53]: thr.stop()
```

Например, на клиентской машине может работать графический пользовательский интерфейс (на питоне легко написать такой интерфейс, причём он будет работать на любой платформе — Linux, Windows, Mac — без малейших изменений в программе). Эта клиентская программа обращается к нескольким мощным серверам для проведения длинных вычислений, и регистрирует callback функции, которые, например, добавляют очередную точку на график.

Наконец, закроем связь с машиной `ееерс`:

```
In [54]: еее.close()
```

Сеанс связи с `гpus` сервером удобно записывать как

```
with server.classic_connection() as еее:  
    еее.execute('n=2')  
    # и так далее
```

Наконец, закроем `ssh` связь с `ееерс`.

```
In [55]: ееерс.close()
```

2.10 cython

`cython` позволяет писать программы, выглядящие почти как питонские, но с добавлением статических деклараций типов. Эти программы (`foo.pyx`) транслируются в исходные тексты на C (`foo.c`) и затем компилируются. Определённые в них функции могут использоваться из программ на чистом питоне. Программа на `cython`-е может также вызывать функции из библиотек, написанных на C. `cython` не пытается автоматически сгенерировать интерфейсы к таким библиотекам, читая их `.h` файлы; для этого можно использовать `swig` или другие подобные системы.

В `ipython` можно писать `cython` фрагменты `inline`, если загрузить расширение `cython`.

```
In [1]: %load_ext cython
```

2.10.1 Функции

Это интерпретируемая функция на питоне.

```
In [2]: def fib(n):
        if n<=2:
            return 1
        a,b=1,1
        for i in range(n-2):
            a,b=b,a+b
        return b
```

```
In [3]: fib(90)
```

```
Out[3]: 2880067194370816120
```

```
In [4]: %timeit fib(90)
```

```
100000 loops, best of 3: 7.05 µs per loop
```

Это такая же функция на `cython`, типы переменных не объявлены — то есть все они обычные питонские объекты.

```
In [5]: %%cython
        def dyn_fib(n):
            if n<=2:
                return 1
            a,b=1,1
            for i in range(n-2):
                a,b=b,a+b
            return b
```

```
In [6]: dyn_fib(90)
```

```
Out[6]: 2880067194370816120
```

```
In [7]: %timeit dyn_fib(90)
```

100000 loops, best of 3: 5.52 μ s per loop

Получилось чуть быстрее. Скомпилированная программа выполняет всю ту же возню с типами и их преобразованиями, что и интерпретируемая.

Теперь типы декларированы статически.

```
In [8]: %%cython
def stat_fib(long n):
    cdef long i,a,b
    if n<=2:
        return 1
    a,b=1,1
    for i in range(n-2):
        a,b=b,a+b
    return b
```

```
In [9]: stat_fib(90)
```

```
Out[9]: 2880067194370816120
```

```
In [10]: %timeit stat_fib(90)
```

The slowest run took 6.68 times longer than the fastest. This could mean that an intermediate re
1000000 loops, best of 3: 630 ns per loop

Получилось на порядок быстрее.

`c_fib` — это фактически функция на C, только написанная в `cython`-ском синтаксисе. Её можно вызывать откуда угодно в той же программе на `cython`, но не из питонской программы. Поэтому напомним обёртку, которую можно вызывать из питона.

```
In [11]: %%cython
cdef long c_fib(long n):
    cdef long i,a,b
    if n<=2:
        return 1
    a,b=1,1
    for i in range(n-2):
        a,b=b,a+b
    return b
def wrap_fib(long n):
    return c_fib(n)
```

```
In [12]: wrap_fib(90)
```

```
Out[12]: 2880067194370816120
```

```
In [13]: %timeit wrap_fib(90)
```

The slowest run took 5.85 times longer than the fastest. This could mean that an intermediate re
1000000 loops, best of 3: 632 ns per loop

Время то же самое.

`cpdef` создаёт как С функцию, так и питонскую. Первая вызывается из `cython`, вторая из питона.

```
In [14]: %%cython
         cpdef long cp_fib(long n):
             cdef long i,a,b
             if n<=2:
                 return 1
             a,b=1,1
             for i in range(n-2):
                 a,b=b,a+b
             return b
```

```
In [15]: cp_fib(90)
```

```
Out[15]: 2880067194370816120
```

```
In [16]: %timeit cp_fib(90)
```

The slowest run took 5.04 times longer than the fastest. This could mean that an intermediate re
1000000 loops, best of 3: 637 ns per loop

Время то же самое.

2.10.2 Интерфейс к библиотеке на С

Пусть у нас есть файл на С.

```
In [17]: !cat cfib.c
```

```
long cfib(long n)
{
    long i,a,b,c;
    if(n<=2) return 1;
    {
        a=1; b=1;
        for(i=2;i<n;++i)
            { c=a+b; a=b; b=c; }
        return b;
    }
}
```

```
In [18]: !cat cfib.h
```

```
long cfib(long n);
```

Скомпилируем его.

```
In [19]: !gcc -fPIC -c cfib.c
```

Напишем обёртку на cython.

```
In [20]: !cat wrap.pyx
```

```
cdef extern from "cfib.h":
    long cfib(long n)

def fib(long n):
    return cfib(n)
```

Скомпилируем её и соберём в библиотеку.

```
In [21]: %%!
          cython -3 wrap.pyx
          CFLAGS=$(python-config --cflags)
          LDFLAGS=$(python-config --ldflags)
          gcc $CFLAGS -fPIC -c wrap.c
          gcc $LDFLAGS -shared wrap.o cfib.o -o wrap.so
```

```
Out[21]: []
```

Эту библиотеку можно импортировать в программу на питоне.

```
In [22]: from wrap import fib
```

```
In [23]: fib(90)
```

```
Out[23]: 2880067194370816120
```

```
In [24]: %timeit fib(90)
```

```
The slowest run took 4.65 times longer than the fastest. This could mean that an intermediate re
1000000 loops, best of 3: 574 ns per loop
```

Получилось чуть быстрее, чем функция на cython.

2.10.3 Структуры

Структуры можно описывать в cython с помощью `ctypedef struct`. Поля в них описываются фактически в синтаксисе C. Переменную, описываемую в `cdef`, можно, если хочется, сразу инициализировать. Имя типа-структуры можно использовать как функцию, аргументы которой — её поля (в порядке описания). `print` печатает структуру как словарь; на самом деле это не словарь, а структура языка C, не содержащая накладных расходов по памяти и времени, имеющих у словаря, но и не дающая гибкости словаря. Поля структуры обозначаются `z.re`; их можно менять.

В `cython` можно работать с указателями. Импортируем `malloc` и `free` из стандартной библиотеки. Результат `malloc` — адрес, его нужно привести к правильному типу, используя `<type>`. В С поля структуры, на которую ссылается `w`, обозначаются `w->re`; в `cython` — просто `w.re`. В С структура, на которую ссылается `w`, обозначается `*w`; в `cython` такой синтаксис не разрешён, вместо этого надо писать `w[0]` (в С это тоже законная форма записи, но чаще используется `*w`). При работе с указателями управление памятью производится вручную, а не автоматически, как в питоне, так что не забывайте `free`.

```
In [25]: %%cython
         ctypedef struct mycomplex:
             double re
             double im
         cdef mycomplex z=mycomplex(1.,2.)
         print(z)
         print(z.re)
         z.re=-1
         print(z)
         # pointers
         from libc.stdlib cimport malloc,free
         cdef mycomplex *w=<mycomplex*>malloc(sizeof(mycomplex))
         w.re,w.im=2.,1.
         print(w[0])
         free(w)

{'re': 1.0, 'im': 2.0}
1.0
{'re': -1.0, 'im': 2.0}
{'re': 2.0, 'im': 1.0}
```

2.10.4 cdef классы

`cython` позволяет определять классы, объекты которых являются фактически структурами языка С. Их атрибуты нужно статически описывать с помощью `cdef`; во время выполнения нельзя добавлять новые атрибуты (или уничтожать имеющиеся). Вот пример такого класса. Его основной метод `atol` вызывает функцию `atol` из стандартной библиотеки С, преобразующую строку в `long`.

```
In [26]: !cat C1.pyx

from libc.stdlib cimport atol

cdef class C1:

    cdef:
        char *s
        long n

    def __init__(self):
```

```

        self.s=NULL
        self.n=0

    def set_s(self, bytes s):
        self.s=s

    def get_n(self):
        return self.n

    def atol(self):
        self.n=atol(self.s)

```

Есть удобный способ импортировать `pyx` модуль в питон: `pyximport`, он автоматически произведёт преобразование в C, компиляцию и сборку.

```
In [27]: import pyximport
         pyximport.install()
```

```
Out[27]: (None, <pyximport.pyximport.PyxImporter at 0x7f2185779780>)
```

```
In [28]: from C1 import C1
```

```
In [29]: o=C1()
         s=b"12345"
         o.set_s(s)
         o.atol()
         print(o.get_n())
```

```
12345
```

Тип `char*` в C соответствует типу `bytes` в питоне. При совместном использовании питона с его автоматическим управлением памятью и C с указателями нужно соблюдать осторожность. Строка `b"12345"` доступна в питоне как значение переменной `s`, поэтому занимаемая ей память не будет освобождена, пока `s` не будет присвоено другое значение. Мы скопировали её адрес в атрибут `o.s` типа `char*`. Если бы мы не присвоили эту строку переменной `s`, а прямо подставили бы её в качестве аргумента метода `o.set_s`, то питон не знал бы, что её надо сохранять, и освободил бы занимаемую её память. Указатель `o.s` указывал бы после этого неведомо куда, с катастрофическими последствиями.

Усовершенствуем немного эту `cython` программу. По умолчанию `cdef` атрибуты недоступны ни из питона, ни из `cython` программы. Но можно описать их как `public` или `readonly`, тогда не нужны будут методы `get_foo` и `set_foo`. Метод `__init__` может быть и не будет вызван (например, другой класс унаследовал текущий, и его `__init__` не вызвал `__init__` родителя); если в структуре есть указатели, то они могут остаться неинициализированными. Поэтому лучше использовать `__cinit__`, который обязательно вызывается сразу после выделения память для объекта.

```
In [30]: !cat C2.pyx
```



```
from libc.stdlib cimport atol
```

```
cdef class C2:
```

```
    cdef public char *s
    cdef readonly long n
```

```
    def __cinit__(self):
        self.s=NULL
        self.n=0
```

```
    def atol(self):
        self.n=atol(self.s)
```

```
In [31]: from C2 import C2
```

```
In [32]: o=C2()
         o.s=s
         o.atol()
         print(o.n)
```

```
12345
```

`cdef` классы поддерживают наследование (только от одного класса, не множественное). Можно написать класс-потомок как `cdef` класс на `cython`. Можно и написать класс-потомок на питоне. Пусть мы хотим добавить к нашему классу метод преобразования строки в число с плавающей точкой, но нам лень использовать `atof` из стандартной библиотеки С. Сделаем это обычными средствами питона. Атрибут `x` добавляется к объектам класса `C3` динамически, описывать его не надо.

```
In [33]: class C3(C2):
```

```
    def atof(self):
        self.x=float(self.s)
```

```
In [34]: o=C3()
         s=b"12345.6789"
         o.s=s
         o.atof()
         print(o.x)
```

```
12345.6789
```

2.10.5 Интерфейс к библиотеке на С

Рассмотрим очень упрощённый пример того, как можно написать удобный питонский интерфейс к библиотеке на С, используя `cython`. Если бы мы хотели использовать эту библиотеку из программы на С, достаточно было бы включить `#include "foo.h"` в эту программу.

In [35]: !cat cfoo.h

```
typedef struct { long n; double x; } CFoo;
CFoo *Foo_new(long n,double x);
void Foo_del(CFoo *z);
double Foo_f(CFoo *z,double y);
```

Здесь описан тип-структура CFoo. Функция Foo_new создаёт и инициализирует такую структуру и возвращает указатель на неё. Функция Foo_del уничтожает эту структуру. Наконец, функция Foo_f делает какое-то вычисление со своим параметром y и данными из структуры. Подобным образом часто выглядят интерфейсы к генераторам случайных чисел: мы можем создать несколько структур с начальными данными и получить несколько независимых потоков случайных чисел.

А вот реализация на C.

In [36]: !cat cfoo.c

```
#include <stdlib.h>
#include "cfoo.h"

CFoo *Foo_new(long n,double x)
{   CFoo *r=(CFoo*)malloc(sizeof(CFoo));
    r->n=n;
    r->x=x;
    return r;
}

void Foo_del(CFoo *z)
{ free(z); }

double Foo_f(CFoo *z,double y)
{ return z->n*y+z->x; }
```

В первую очередь мы напомним файл определений cython. Он почти копирует foo.h с минимальными синтаксическими изменениями.

In [37]: !cat foo.pxd

```
cdef extern from "cfoo.h":

    ctypedef struct CFoo:
        pass

    CFoo *Foo_new(long n,double x)
    void Foo_del(CFoo *z)
    double Foo_f(CFoo *z,double y)
```

Теперь напомним удобную объектно-ориентированную обёртку. Файл определений импортируется при помощи `cimport` (мы уже использовали эту команду, когда импортировали `libc.stdlib`; `cython` содержит ряд стандартных `pxd` файлов, включая `stdlib.pxd`, `stdio.pxd` и т.д.). Теперь определим `cdef` класс `Foo`. Метод `__dealloc__` вызывается в последний момент перед уничтожением объекта (условие `if self.foo!=NULL`: написано из перестраховки, в законном объекте класса `Foo` этот атрибут всегда не `NULL`, т.к. он инициализируется в `__cinit__`).

```
In [38]: !cat foo.pyx
```

```
cimport foo
```

```
cdef class Foo:
```

```
    cdef foo.CFoo *foo
```

```
    def __cinit__(self, long n, double x):
        self.foo=foo.Foo_new(n,x)
```

```
    def __dealloc__(self):
        if self.foo!=NULL:
            foo.Foo_del(self.foo)
```

```
    def f(self, double y):
        return foo.Foo_f(self.foo,y)
```

Скомпилируем и соберём.

```
In [39]: %!
          gcc -fPIC -c cfoo.c
          cython -3 foo.pyx
          CFLAGS=$(python-config --cflags)
          LDFLAGS=$(python-config --ldflags)
          gcc $CFLAGS -fPIC -c foo.c
          gcc $LDLAGS -shared foo.o cfoo.o -o foo.so
```

```
Out[39]: []
```

```
In [40]: from foo import Foo
```

Теперь мы можем в питоне создавать объекты класса `Foo` и вызывать их метод `f`.

```
In [41]: o=Foo(2,0.)
          o.f(3.)
```

```
Out[41]: 6.0
```