Министерство образования и науки Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Долгих Т. Ф., Ширяева Е. В.

Язык Python 3 для научных исследований

(для направлений подготовки 01.03.01 «Математика», 01.03.02 «Прикладная математика и информатика», 01.03.03 «Механика и математическое моделирование», 02.03.02 «Фундаментальная информатика и информационные технологии»)

Ростов–на–Дону
2017

Содержание

Введение **3**

1 Краткое введение в Python на примерах **4**

1.1 Ввод и вывод данных 9
1.2 Оператор условного перехода 12
1.3 Операторы циклов 15
1.4 Функции 19
1.5 Одномерные списки 37
1.6 Двумерные массивы 49
1.7 Множества 51

2 Пакет SymPy **58**

2.1 Функции в SymPy 65
2.2 Степенные ряды в SymPy 65
2.3 Печать математических формул в LaTeX 66
2.4 Работа с выражениями с комплексными значениями . 66
2.5 Алгебраические уравнения 67
2.6 Суммирование рядов 68
2.7 Вычисление пределов 69
2.8 Дифференцирование 69
2.9 Интегрирование 70
2.10 Дифференциальные уравнения 70
2.11 Матрицы 72
2.12 Построение графиков функций и поверхностей 73

3 Пакет NymPy **76**

4 Библиотека Matplotlib **78**

5 Python и LaTeX **80**

5.1 Вызов интерпретатора Python из системы LaTeX 80
5.2 Использование модуля sympy 81
5.3 Рисование графиков в LaTeX при помощи Python . . . 84
5.4 Оформление скриптов при помощи LaTeX 85

Приложение. Установка дополнительных библиотек **88**

Введение

Язык Python является одним из самых простых и эффективных языков программирования на сегодняшний день. Он может использоваться как для начальных шагов в программировании, так и для решения серьезных научных задач.

В учебном пособии рассмотрено использование нескольких основных пакетов языка Python для решения различных научных задач. А качестве приложения приведены краткие теоретические сведения по синтаксису языка Python, а также большое количество примеров простых решения задач.

Работа поддержана Базовой частью государственного задания Министерства образования и науки РФ № 1.5169.2017/8.9.

1 Краткое введение в Python на примерах

Таблица приоритетов (порядке понижения приоритета)

	Описание
‘выражения‘	преобразование к строке
{кл:зн, ...}	словарь пар ключ–значение
[...]	список или списковое включение
(...)	скобки или кортеж
f(аргумент,...)	вызов функции
x[от:до]	выделение среза (от и до)
x[индекс]	взятие элемента по индексу
x.атрибут	ссылка на атрибут
**	возведение в степень
~	побитовое «НЕ»
+, -	унарные плюс и минус (смена знака)
*, /, %, //	умножение, деление, остаток, целочисленное деление
+, -	сложение и вычитание
>>, <<	побитовые сдвиги (вправо и влево)
&	побитовое «И»
^, |	побитовые «Исключающее ИЛИ» и «ИЛИ»
<=, <, >, >=, ==, !=	сравнения
=, %=, /=, //=, -=, +=, *=, **=	операторы присваивания
is, is not	тождественные операторы
in, not in	операторы идентичности
not	логическое отрицание
and	логическое умножение
or	логическое сложение
lambda	лямбда-выражение

Запись целых чисел

В Python числа можно записывать в системах счисления по основаниям 2, 8, 16 и 10, при этом перед числами указываются префиксы, отвечающие за разные системы счисления (см. таблицу ниже). Результат вычислений записывается в десятичной системе счисления. Десятичный результат можно представить в виде строки, изображающей число в соответствующей системе счисления. Для этого используются функции преобразования `bin()`, `oct()` и `hex()`.

Работа с двоичными числами

```
>> 0b10 + 0b10      # сложение двоичных чисел
4
>> bin(0b10 + 0b10) # использование функции преобразования
'0b100'
```

Целочисленный литерал	Префикс
десятичные числа	не используется
двоичные числа	«0b» или «0B»
восьмеричные числа	«0o» или «0O»
16-ричные числа	«0x» или «0X»

Комплексные числа

В Python кроме целых и вещественных чисел также и встроены комплексные числа. Для получения комплексного числа необходимо воспользоваться встроенной функцией преобразования типа

`complex([real[, imag]])`.

Операции с комплексными числами, доступные в языке Python:

1. Нахождение комплексно сопряжённого числа;
2. Определение вещественной и мнимой частей комплексного числа;
3. Сумма, разность, умножение и деление двух комплексных чисел;
4. Модуль комплексного числа;
5. Возведение в степень (в т. ч. и в вещественную) комплексного числа;
6. Сравнение двух комплексных чисел.

Примеры работы с комплексными числами

```
>> x = complex(1, 6) # Определение комплексного числа
>> print('x =', x)
x = (1+6j)
>> print('x* =', x.conjugate()) # Сопряжённое число
x* = (1-6j)
>> a, b = -3, 4
>> y = complex(a, b)
>> print('y =', y)
y = (-3+4j)
>> z = x + y # Сумма двух комплексных чисел
>> print('z = x + y =', z)
z = x + y = (-2+10j)
>> # Вещественная и мнимая части компл. числа
>> print('Re(z) =', z.real, ', Im(z) =', z.imag)
Re(z) = -2.0 , Im(z) = 10.0
>> print('|y| =', abs(y)) # Модуль комплексного числа
|y| = 5.0
>> print('x^2 =', pow(x, 2)) # Возведение в степень
x^2 = (-35+12j)
>> print('x == y? ...', x == y) # Проверка на равенство
x == y? ... False
```

Мнимая единица в стандартном пакете Python обозначается как `j`.

Некоторые встроенные математические функции

Имя функции	Возвращает
`abs(x)`	$\|x\|$, $x \in \mathbb{Z}, \mathbb{R}, \mathbb{C}$
`min(arg1, arg2, ...)`	наименьший из двух или более аргументов.
`max(arg1, arg2, ...)`	наибольший из двух или более аргументов.
`pow(x, n)`	x^n; эквивалент оператора степени: `x**n`.

Подключение модулей

Модули подключаются с помощью инструкции `import`.

Подключение библиотеки

```
import имя модуля
# или
# from имя модуля import *
# или
# from имя модуля import имя метода
```

Список методов из любой библиотеки можно напечатать командой

`dir(имя библиотеки)`,

а вызвать документацию по отдельному методу позволяет команда

`имя библиотеки.имя метода.__doc__`

Например,

Вызов списка методов и констант библиотеки math

```
import math
print(dir(math))
```

Вызов документации по отдельному методу

```
import math
print(math.exp.__doc__)
```

Модуль math

Подключение библиотеки

```
import math
a = math.sqrt(abs(-9))
```

Чтобы не писать math

```
from math import *
a = sqrt(abs(-9))
```

Некоторые константы из модуля math

Имя	Возвращает
`pi`	значение числа $\pi \approx 3{,}1415926$ (с имеющейся точностью)
`e`	основание натуральных логарифмов $e \approx 2{,}718281$ (с имеющейся точностью)
`inf`	$+\infty$ (значение с плавающей точкой)
`nan`	значение в плавающей точкой «not a number» (NaN) value

Некоторые степенные и логарифмические функции из модуля math

Имя функции	Возвращает
`sqrt(x)`	$\sqrt{x}$, где $x \geqslant 0$
`log(x)`	$\ln x$, где $x > 0$
`log10(x)`	$\lg x$, где $x > 0$
`log(x, b)`	$\log_b x$, где $x, b > 0,\ \ b \neq 1$
`exp(x)`	e^x

Некоторые тригонометрические функции из модуля math

Имя функции	Возвращает
`sin(x)`	$\sin x$ (аргумент — радианная мера угла)
`cos(x)`	$\cos x$ (аргумент — радианная мера угла)
`tan(x)`	$\operatorname{tg} x$ (аргумент — радианная мера угла)
`asin(x)`	$\arcsin x$ (результат в радианах)
`acos(x)`	$\arccos x$ (результат в радианах)
`atan(x)`	$\operatorname{arctg} x$ — угол в радианах, удовлетворяющий условию $x = \operatorname{tg} y$, где $-\frac{\pi}{2} < y < \frac{\pi}{2}$
`degrees(x)`	преобразует угол, заданный в радианах, в градусы
`radians(x)`	преобразует угол, заданный в градусах, в радианы

1.1 Ввод и вывод данных

Пример 1.1 (ввод и вывод строк). Продемонстрируем ввод строк и чисел с клавиатуры и вывод их на печать.

Ввод и вывод строк

```
print('Ваше имя?')
# однострочный комментарий
name = input()
print('Здравствуйте, ' + name + '!')
```

Ввод и преобразование строки

```
from math import sqrt
print('Введите целое число')
x = input()  # ввод данных с клавиатуры; x - строка
x = int(x)   # преобразование строки к целому типу
a = sqrt(abs(x))
print('a =', a)  # вывод результата на печать
```

Ввод и преобразование строки в одной инструкции

```
from math import sqrt
x = int(input('Введите целое число '))
a = sqrt(abs(x))
print('a =', a)
```

Пример 1.2 (управление выводом). Использование параметров `sep` и `end` оператора печати `print()` и управляющей последовательности для табуляции (см. также таблицу ниже).

Параметры sep, end и табуляция

```
a, b = 10, -4   # в Python'е можно делать и так
print(a, b)
print(a, b, sep = '\t')      # разделитель - гор. таб-ция
print('a =', a, end = '\t') # в конце строки - гор. таб.
print('b =', b)
```

Управляющие последовательности

Последовательность	Назначение
`\\`	символ обратного слеша (остается один символ \)
`\'`	апостроф (остается один символ ')
`\"`	кавычка (остается один символ ")
`\n`	новая строка (перевод строки)
`\a`	звонок
`\b`	забой
`\f`	перевод страницы
`\r`	возврат каретки
`\t`	горизонтальная табуляция
`\v`	вертикальная табуляция
`\другое`	не является экранированной последовательностью (символ обратного слеша сохраняется)

Использование форматированного вывода

Общий вид форматированного вывода

```
print('[текст] %[-][кол-во позиций]тип данных' %значение)
```

Параметры, указанные в квадратных скобках, могут отсутствовать.

`тип данных` — указывается один из возможных типов данных: `s` для строк, `d` для целых чисел и `g`, `f`, `e`, `E` для вещественных чисел;

`значение` — выводимое значение. Если выводимое значение — выражение, то оно заключается в круглые скобки. Если `значений` для вывода несколько, то они указываются в круглых скобках через запятую.

`[текст]` — поясняющий текст;

`[-]` — число выравнивается по левому краю поля вывода, отведённого под запись числа (иначе, число выравнивается по правому краю);

`[кол-во позиций]` — количество позиций, выделяемых под вывод числа. Если параметр отсутствует, то поле вывода будет минимально необходимой ширины.

Примеры форматированного вывода

«-» — выравнивание по левому краю

```
k = pow(2, 1/2)
print('%9.2e'  %k)  # _1.41e+00
print('%-9.2e' %k)  # 1.41e+00_
```

Здесь _ — пробел.

Вывод вещественного числа с фиксированной точкой

```
k = pow(2, 1/2)
print('%9.7f' %k)  # 1.4142136
print('%.2f'  %k)  # 1.41
```

Вывод вещественного числа с плавающей точкой

```
k = pow(2, 1/2)
print('%14.7e' %k)      #  1.4142136e+00
print('%14.7e' %(-k))   # -1.4142136e+00
print('%.2e'  %k)       # 1.41e+00
```

Вывод нескольких значений

```
k = pow(2, 1/2)
print('%9.2e; %.4e; %d' %(k, k, k))
```

Результат

```
 1.41e+00; 1.4142e+00; 1
```

1.2 Оператор условного перехода

Синтаксис условного оператора с одной ветвью

```
if условие:
    блок инструкций
```

Здесь `условие` — некоторое логическое выражение.

Если `условие` истинно, то выполняется `блок инструкций`; если же `условие` ложно, то `блок инструкций` не будет выполнен и управление передается следующему оператору программы.

Синтаксис условного оператора с двумя ветвями

```
if условие:
    блок инструкций 1
else:
    блок инструкций 2
```

Если `условие` истинно, то выполняется `блок инструкций 1`, иначе выполняется `блок инструкций 2`.

В языке Python для выделения блоков инструкций используются отступы. Все инструкции, которые относятся к одному блоку, должны иметь равную величину отступа слева (4 пробела; символ табуляции не рекомендуется).

Пример 1.3 (использование условных операторов)**.** Продемонстрируем использование условных операторов на примерах поиска $y = |x|$, $\max(x, y)$, $\min(x, y)$. Заметим, что все эти выражения в языке Python могут быть вычислены и с помощью встроенных функций.

Приведен также пример использования условного оператора в случае многоальтернативного решения.

Вычисление $y = |x|$ без использования функции abs()

```
1 x = int(input('Введите целое число '))
2 y = x
3 if x < 0: y = -x
4 print('|', x, '| = ', y, sep='')
```

Поиск max(x,y) и min(x,y)

```
print('Введите 2 числа')
x = int(input())
y = int(input())
if x > y:
    Max, Min = x, y
else:
    Max, Min = y, x
```

Многоальтернативное решение

```
if x == 1:
    if y > x:
        print('x=1, y>x')
    else:
        print('x=1, y<=x')
else:
    print('x<>1')
```

Тернарная условная операция

Простейшую условную инструкцию

```
if X:
    A = Y
else:
    A = Z
```

можно сократить до одной строки, используя следующую конструкцию:

```
A = Y if X else Z
```

В результате работы инструкции, выполнится выражение Y, если значение X истинно, иначе выполнится выражение Z.

Пример 1.4 (поиск наибольшего из двух чисел; if-else)**.** Найти $\max(x, y)$.

Поиск max(x,y); if-else

```
if x > y:
    Max = x
else:
    Min = y
```

Поиск max(x,y); условная операция

```
Max = x if x > y else y
```

Каскадные условные инструкции

В каскадной условной инструкции условия `if`, ..., `elif` проверяются по очереди. При истинности какого-то из условий выполняется соответствующий блок. Если все проверяемые условия ложны, то выполняется блок `else`, если он присутствует.

Пример 1.5. Приведем пример использования каскадных условных инструкций.

Какой четверти принадлежит точка (x,y)?

```
x = float(input('x = '))
y = float(input('y = '))
if x > 0 and y > 0:
    print('I четверть')
elif x < 0 and y > 0:
    print('II четверть')
elif x < 0 and y < 0:
    print('III четверть')
elif x > 0 and y < 0:
    print('IV четверть')
else:
    print('Точка лежит на координатной оси')
    print('или является началом координат.')
```

1.3 Операторы циклов

Оператор цикла с условием

Синтаксис оператора цикла с условием (`while`; с предусловием):

```
while условие: оператор
```

При выполнении данного оператора проверяется `условие` (условие повторения цикла), и если оно соблюдается, то выполняется `оператор` (оператор цикла). Как только на очередном шаге окажется, что `условие` не соблюдается, то выполнение оператора цикла прекращается.

Оператор цикла с предусловием может ни разу не выполниться. Тело цикла не выполняется, если условие повторения цикла при первой проверке оказалось ложным.

Оператор цикла с параметром

Синтаксис оператора цикла с параметром (`for-to`)

```
for ПЦ in итерируемый объект: блок инструкций
```

Здесь

- `ПЦ` — параметр цикла;
- `итерируемый объект` — строка, список, словарь, файл,...
- `блок инструкций` представляет собой тело цикла (любой оператор, в том числе составной).

Синтаксис заголовка оператора for i in range(n)

```
for i in range(количество шагов цикла):
```

Синтаксис заголовка оператора for i in range(n, m, [step])

```
for i in range(нач. знач., кон. знач.+1, [шаг]):
```

Здесь в [...] отмечена необязательная часть.

Синтаксис заголовка оператора for i in [элементы списка]:

```
for i in [элементы списка]:
```

Пример 1.6 (сумма чётных цифр целого числа; while). Найти сумму чётных цифр в записи натурального числа.

Сумма чётных цифр в числе

```
m = int(input())   # исходное число
S = 0              # инициализация суммы
while m > 0:
    d = m % 10     # цифра числа
    if d % 2 == 0: # если цифра чётная
        S += d     # суммирование
    m = m // 10
print(S)
```

Пример 1.7 (нахождение $\sum_{i=1}^{10} i$). Вычислить сумму первых десяти натуральных чисел $\sum_{i=1}^{10} i$.

for i in range(10):

```
S = 0
for i in range(10):
    S = S + (i+1)
print('S =', S)
```

(for i in range(1,11):

```
S = 0
for i in range(1,11):
    S = S + i
print('S =', S)
```

for i in range(10,0,-1):

```
S = 0
for i in range(10,0,-1):
    S = S + i
print('S =', S)
```

while i <= 10:

```
S = 0
i = 1
while i <= 10:
    S = S + i
    i += 1
print('S =', S)
```

Пример 1.8 (цикл с параметром). Демонстрация разных заголовков цикла с параметром.

Функция range(n)
```
for i in range(4):
    print(i)
```

Функция range(n, m)
```
for i in range(0, 4):
    print(i)
```

Функция range(n, m, step)
```
for i in range(0, 4, 1):
    print(i)
```

Функция range(n, m, -step)
```
for i in range(3, -1, -1):
    print(3-i)
```

Элементы списка
```
for i in [0, 1, 2, 3]:
    print(i)
```

Элементы кортежа
```
for i in (0, 1, 2, 3):
    print(i)
```

Вывод букв слова
```
for i in 'котик':
    print(i)
```

Счёт
```
for i in 'один', 'два', 'три':
    print(i)
```

Список значений разного типа для параметра цикла
```
for i in 1, 2, 3, 'раз', 'два', True:
    print(i)
```

Пример 1.9 (использование `break` и `else` в циклах). Ищем в списке слово raccoon.

Использование break и else

```
s = 'raccoon'
for i in ['cat', 'dog', 'mouse']:
    if i == 'mise':
        print('Слово', s, 'в списке есть')
        break
else: # значит вышли не по break
    print('Слова', s, 'в списке нет')
```

Результат работы для списка ('cat', 'dog', 'mouse')

```
Слова raccoon в списке нет
```

Результат работы для списка ('cat', 'dog', 'raccoon')

```
Слово raccoon в списке есть
```

1.4 Функции

Описание функции

```
def имя_функции(список параметров):
   тело функции
   [return значение_функции]
```

Здесь в [...] отмечена необязательная часть.

Выйти из функции можно в любой момент, используя инструкцию `return`. Если инструкция `return` используется без аргументов или она вообще не используется, то функция будет возвращать значение None.

Для того чтобы функция вернула не одно значение, а более, можно после инструкции return через запятую записать несколько переменных:

```
return a, b
```

Функция вернет кортеж из этих переменных. Результат вызова такой функции можно будет использовать во множественном присваивании:

```
n, m = имя_функции(список параметров)
```

Также можно собрать результат в любую структуру и вернуть ее.

Функция должна быть записана выше своего использования в любом месте программы (обычно выше текста основной программы). Также описания функций можно выносить в отдельные файлы, которые называются модулями.

Синтаксис вызова функции:

```
[переменная =] имя_функции(список параметров)
```

Пример 1.10 (функция без параметров). Функции без параметров и выходных значений обычно используются для отображения каких-либо статичных текстов, например, меню:

Функция без параметров и значения

```
def print_menu():
    print('   Меню')
    print('1: Метод 1')
    print('2: Метод 2')
    print('3: Выход')
print_menu()  # вызов функции без параметров и значения
```

Пример 1.11 (функции с параметрами). Описание и вызов функции для нахождения $x + y$.

Функция для нахождения $x + y$

```
def sum2(x, y):
    return x + y

S = sum2(5, 3)
```

Пример 1.12 (более одного результата). Для возвращения функций более одного значения используется список

Два результата у функции

```
def MinMax(a, b):
    if a > b:
        return [b, a]
    else:
        return [a, b]
```

Тогда результат вызова функции можно будет использовать либо так

Примеры вызовов функций

```
minab = MinMax(56, 17)[0]
maxab = MinMax(56, 17)[1]
print('min =', minab)
print('max =', maxab)
```

либо во множественном присваивании:

Примеры вызовов функций

```
minab, maxab = MinMax(56, 17)
print('min =', minab)
print('max =', maxab)
```

Пример 1.13 (функция для вычисления площади треугольника). Вычислить площадь треугольника по формуле Герона (параметры a, b, c — здесь стороны треугольника)

Использование функции с параметрами

```
import math
def geron(a, b, c):
    p = (a + b + c) / 2
    return math.sqrt(p * (p - a) * (p - b) * (p - c))

print('Площадь =', geron(3, 4, 5))
```

При вызове функции первому аргументу `a` передается первое значение `3`, второму аргументу `b` – 4, и третьему `c` — 5. Такой способ передачи аргументов, когда данные передаются в порядке их перечисления, называется позиционным.

Другой способ передачи параметров — по ключу (или по имени). Используя его можно было бы вызвать предыдущую функцию так:

```
print('Площадь =', geron(a = 3, b = 4, c = 5))
```

Причем перечислять параметры можно в любом порядке:

```
print('Площадь =', geron(c = 5, b = 4, a = 3))
```

Можно комбинировать два способа вызова (позиционный и по имени) в одном вызове:

```
print('Площадь =', geron(3, 4, c = 5))
```

Тогда при вызове позиционные параметры обязательно должны идти перед параметрами передаваемыми по имени:

Сначала позиционные!

```
print('Площадь =', geron(c = 5, 3, 4) ) # ошибка,
# сначала должны идти позиционные,
# а потом передаваемые по имени параметры
```

Значения параметров по умолчанию

При определении функции для ее параметров можно задавать значения по умолчанию, для этого они задаются в виде последовательности пар:

`идентификатор = значение.`

Пример 1.14 (значения по умолчанию; задание). Приведем заголовки функций со значениями по умолчанию.

Заголовки функций

```
def geron2(a=3, b=4, c=5):
def point_xy(x=0, y=0):
def set_pixel(color, x=0, y=0):
def set_pixel_no(x=0, y=0, color): # ошибка (см. ниже)
```

Причем, если в функции используются как параметры без начальных значений, так и с начальными значениями (как в функции `set_pixel`), то обычные параметры всегда должны идти перед параметрами с начальными значениями.

Пример 1.15 (значения по умолчанию; вызов). При вызове функции можно вообще не передавать параметры со значениями по умолчанию.

Значения по умолчанию; вызов

```
geron2()          # вызывается с параметрами по умолчанию
point_xy(10)      # параметры x=10 и y=0
set_pixel('red')  # параметры color='red', x=0, y=0
geron2(7, b = 8)  # параметры:
# a = 7 - передается по позиции,
# b = 8 - передается по имени,
# c = 5 - используется значение по умолчанию.
```

Значения по умолчанию задаются один раз при определении функции, а не при каждом вызове!

Пример 1.16 (сравнение двух выражений с заданной точностью)**.** Вещественные числа по умолчанию сравниваются с машинной точностью. На практике такая точность бывает нужна редко. Напишем функцию для сравнения двух выражений с заданной точностью. Точность по умолчанию возьмем $\varepsilon = 0.000001$.

Сравнение с заданной точностью

```
def Compare(x, y, eps = 1e-6):
    return abs(x - y) <= eps

print(Compare(1.55, 1.6));       # вернет False
print(Compare(1.55, 1.6, 0.1));  # вернет True
```

Аргументы произвольной длины

В случае, когда заранее неизвестно количество передаваемых функции аргументов, используются аргументы произвольной длины. Перед именем переменной ставится звездочка (*).

Аргументы произвольной длины

```
def sumN(*args):
    s = 0
    for x in args: s = s + x
    return s

k = sumN(3, 4)
print(k)

k = sumN(3, 4, 5)
print(k)
```

Все переданные в функцию `sumN` параметры соберутся в кортеж с именем `args`.

Покажем возможное описание стандартной функции Python `max`, которая тоже принимает произвольное число аргументов.

Произвольное число аргументов. Поиск максимума

```
def max(*args):
    m = args[0]
    for x in args[1:]:
        if x > m:
            m = x
    return m

print(max(3, 5, 4))
```

Параметры-функции

Пример 1.17. Вычислить среднее арифметическое значение для функции $f(x)$ на заданном интервале. В качестве функций выбрать

$$f_1(x) = x^2 + 3x, \quad \text{или} \quad f_2(x) = \sin x.$$

Демонстрация П/П с параметрами-функциями

```
import math

def F1(x):
    return  x*x + 3*x

def F2(x): # тестовая функция
    return  x

def Mean(a, b, N, F):
    h = (b - a) / N
    S = 0
    for i in range(0, N+1):
```

```
        x = a + h * i
        S = S + F(x)
    return S / N

a, b = -math.pi, math.pi

print(Mean(a, b, 10, F1))
print(Mean(a, b, 10, math.sin)) # исп. станд. функции
print(Mean(-1, 1, 2, F2))       # тест
```

Функции `F1` и `F2` можно задать, используя анонимную функцию `lambda`.

lambda-функции

```
import math

F1 = lambda x: x*x + 3*x
F2 = lambda x: x
...
print(Mean(a, b, 10, F1))
print(Mean(-1, 1, 2, F2))
```

Заданные выше функции можно использовать и для рисования графиков с использованием одного из пакетов (см. п.2). Например,

Пакет sympy. Три графика на одной плоскости

```
from sympy import*

q = plot((F1(x), (x, a, b)),  sin(x), (x, a, b)), \
         (F2(x), (x, a, b)), show=False)
q[0].line_color = 'blue'
q[1].line_color = 'green'
q[2].line_color = 'red'
q.show()
```

Результат работы программы приведен рис. 1.

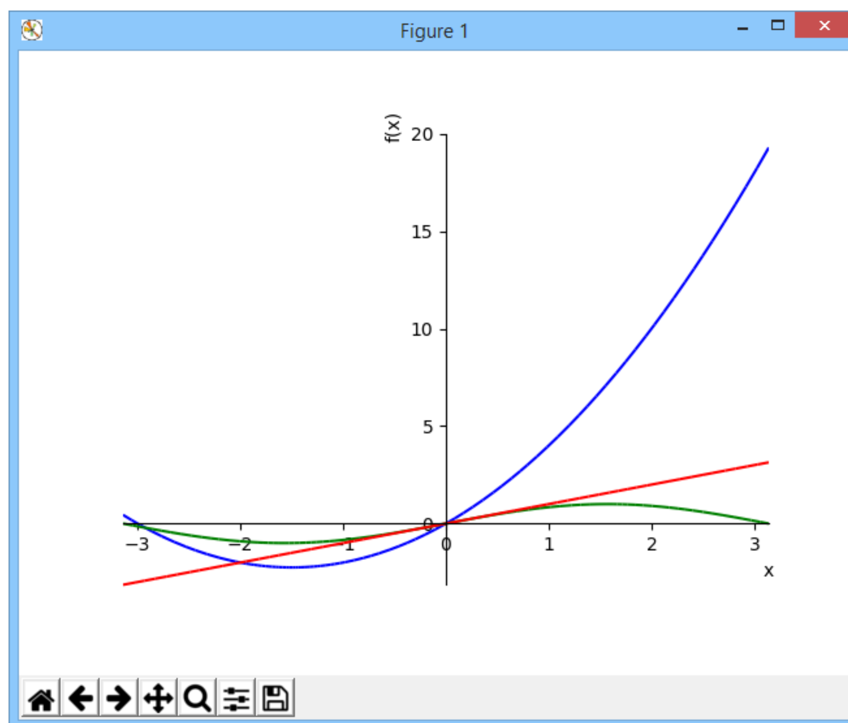

Рис. 1: Построение графиков функций

Пример 1.18 (численное интегрирование)**.** С геометрической точки зрения интеграл $\int\limits_a^b f(x)\,dx$ — это площадь криволинейной фигуры, ограниченной линиями: осью абсцисс, графиком функции $y = f(x)$, прямыми $x = a$, $x = b$.

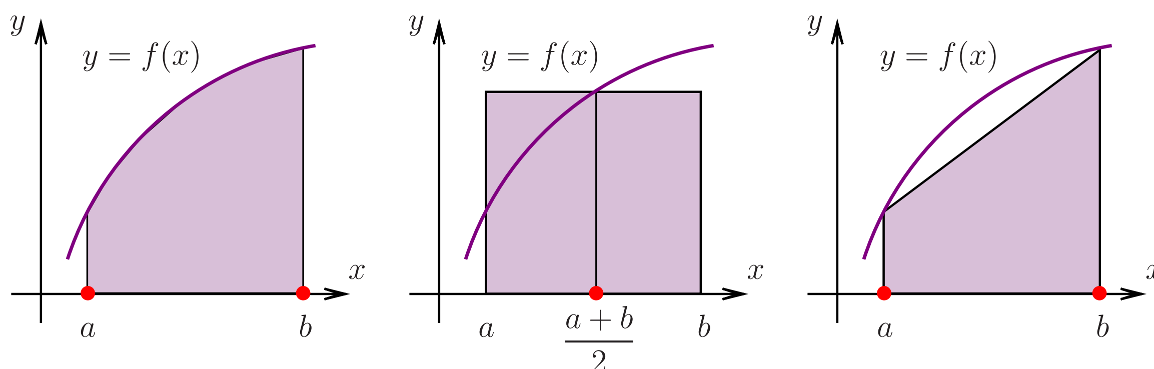

Рис. 2: Геометрическая интерпретация интеграла и методов численного интегрирования

На практике фигуру сложной формы заменяют более простой, например, прямоугольником или трапецией (см. рис. 2).

При реальных вычислениях используются составные формулы численного интегрирования:

Формула прямоугольников: $I = h\sum_{k=0}^{n-1} f\left(\frac{x_k + x_{k+1}}{2}\right).$

Формула трапеций: $I = \frac{h}{2}\left(f(x_0) + 2\sum_{k=1}^{n-1} f(x_k) + f(x_n)\right).$

Отрезок интегрирования $[a, b]$ разбивается на n частей с шагом

$$h = \frac{b-a}{n},$$

т. е. расстояние между двумя соседними узлами равно h

$$x_{k+1} - x_k = h, \quad \forall\, k.$$

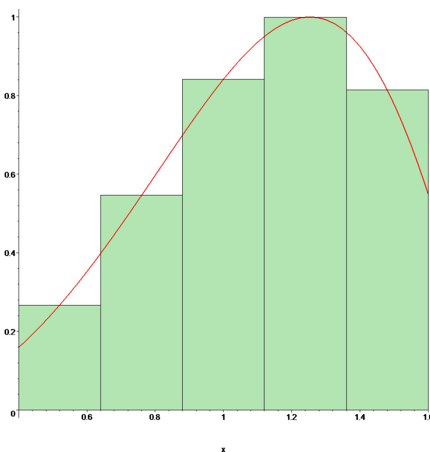

Рис. 3: Метод средних прямоугольников

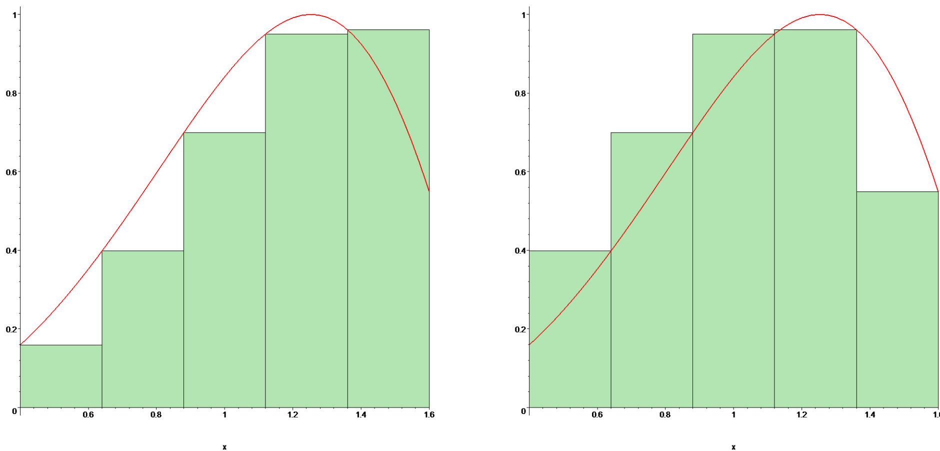

Рис. 4: Методы левых и правых прямоугольников

Реализация численного интегрирования на языке Python

Для демонстрации возможностей языка Python используем не составные формулы.

Численное интегрирование

```
def IntRect(a, b, FF):
    return (b - a) * FF((a + b)/2)
def F1(x):
    return  math.sin(x*x)
def F0(x):
    return  x


a, b = 0.4, 0.6
print('Test =%8.5f' %IntRect(a, b, F0))
print('Int(sin x^2) =%8.5f' %IntRect(a, b, F1))
print('Int(sin x) =%8.5f' %IntRect(a, b, math.sin))
```

$\int\limits_{0.4}^{0.6} f(x)\,dx$. Библиотека sympy

```
from sympy import *
x = Symbol('x')
```

```
Int0 = integrate(x, (x, 0.4, 0.6)).doit().evalf()
Int1 = integrate(sin(x*x), (x, 0.4, 0.6)).doit().evalf()
Int2 = integrate(sin(x), (x, 0.4, 0.6)).doit().evalf()
print(Int0) #
print(Int1) #
print(Int2) #
```

Результат (sympy)

```
0.100000000000000
0.0500418723959274
0.0957253790932068
```

Результат (наш алгоритм)

```
Test = 0.10000
Int(sin x^2) = 0.04948
Int(sin x) = 0.09589
```

Функция как результат другой функции

Функция может возвращать любые объекты: списки, функции,...

Результат 300

```
def F1(x):
    def F11(y):
        return x + y
    return F11

funcF11 = F1(100)  # funcF11 - это функция
print(funcF11(200))
```

Строковый тип данных

Тип данных `str` предназначен для хранения последовательности символов с произвольным доступом.

Символы в строке нумеруются с нуля. Функция `len()` служит для определения длины строки — количества элементов в строке. Длина пустой строки `len('')` равна нулю.

При выполнении программы переменная типа `str` вводится с клавиатуры без апострофов.

В языке Python строковый тип — неизменяемый тип, т. е. изменить символ в строке следующей инструкцией нельзя: `S[1] = 'I'`

Пример 1.19. Пусть определена переменная Date, в которой содержится строка с датой формата:

`ДД МММ, ГГГГ`;

где ДД — число месяца (позиции 1–2), МММ — месяц (4–6) и ГГГГ — год (9–12).

Извлечение названия месяца из исходной строки:

`Month = Date[3:6]`

Поиск месяца

```
Date = '22 OCT, 2017'
Month = Date[3:6]
print(Month)
```

Функции и методы строк

Методы поиска в строке

Метод `count()` подсчитывает количество вхождений строки Образец в Строку

`Строка.count(Образец[, Старт][, Финиш])`

Указание необязательных параметров `Старт`, `Финиш` позволяет выполнять подсчет числа вхождений строки `Образец` в срезе строки

`[Старт:Финиш]`.

Подсчитываются только непересекающиеся вхождения, например:

Поиск подстроки в строке

```
print(('7' * 10).count('77'))   # вернёт 5
```

Поиск подстроки в строке. Использование срезов

```
# Поиск в строке с третьего по 10 символ
print(('enotik begemotik').count('e',3,10)) # вернёт 1

# Поиск в строке с третьего символа по конец строки
print(('enotik begemotik').count('e',3))    # вернёт 2
```

Метод `find()` ищет в Строке подстроку Образец

`Строка.find(Образец[, countS])`

Поиск подстроки в строке происходит слева направо. Для поиска справа налево существует метод `rfind()`.

Если подстрока найдена, то функция возвращает индекс первого вхождения искомой подстроки, иначе — возвращает значение -1.

Пример 1.20. Пусть определена переменная Date, в которой содержится строка с датой формата:

`Число Месяц, Год`

Извлечь название месяца

Поиск месяца

```
Date = '2 October 2017'
Month = Date[Date.find(' ')+1:Date.find(',')]
print(Month)
```

Метод replace

Метод `replace()` заменяет в строке все вхождения подстроки `oldS` на подстроку `newS`

```
Строка.replace(oldS, newS[, countS])
```

Необязательный параметр `countS` указывает, что заменены будут только первые `countS` из найденных строк.

Замена подстроки в строке

```
print('mmcs. 1 course'.replace('s', 'S'))
# вернет 'mmcS. 1 courSe'
```

Замена подстроки в строке. Третий параметр

```
print('mmcs. 1 course'.replace('s', 'S', 1))
# вернет 'mmcS. 1 course'
```

Пример 1.21. Заменить все вхождения подстроки `del` на `Ins`. Метод `replace()` не использовать.

Замена

```
# k  { позиция вхождения искомой подстроки }
S = 'Del и del. Del или del'
print(S)

while S.find('del') != -1:
    k = S.find('del')
    D = len('del')
    S = S[:k] + 'Ins' + S[k+D:]
print(S)
```

Пример 1.22 (перестановка слов). Дана строка, состоящая из нескольких слов (два и более), разделенных пробелом. Переставить первое и последнее слова местами. Результат записать в строку и вывести на экран.

Указание. При решении этой задачи не использовать инструкцию `if`.

Перестановка слов

```
1 S = input()
2 S1 = S[:S.find(' ')]
3 S2 = S[S.rfind(' ')+1:]
4 S0 = S[S.find(' '):S.rfind(' ')+1]
5 SNew = S2 + S0 + S1
6 print(SNew)
```

Программа будет работать правильно только, если в начале и в конце строки нет пробелов. Поэтому перед использованием среза требуется привести строку в соответствие нашим правилам — «в начале и в конце строки пробелы отсутствуют». Для удаления пробелов в начале и в конце строки используется метод `strip()`.

Методы split и join

Метод `split()`

`строка.split([вид разделителя])`

преобразует строку в список подстрок, разделенных по умолчанию пробелами.

Метод `join()`

`вид разделителя.join(строка)`

позволяет создавать строки из списка строк.

Создание списка строк из строки

```
s = input()              # ввод 1 2 3 (Enter = конец ввода)
a = s.split()            # результат ['1', '2', '3']
```

Создание списка чисел из строки

```
a = input().split()     # строка 12 30 8 678 23
for i in range(len(a)):
    a[i] = int(a[i])    # преобразование элементов строки
print(a)                # [12, 30, 8, 678, 23]
```

Создание списка чисел из строки. Использование генератора

```
a = [int(s) for s in input().split()]
print(a)
```

Разбиение строки

```
s = 'red yellow green'
print(s.split())       # разделитель по умолчанию пробел
                       # ['red', 'yellow', 'green']

s = 'red, yellow, green'
print(s.split(','))    # разделитель <<,>>
                       # ['red', ' yellow', ' green']

s = 'red, yellow, green'
print(s.split(', '))   # разделитель <<,>> + пробел
                       # ['red', 'yellow', 'green']
```

Использование метода join()

```
1 a = ['red', 'green', 'blue'] # список строк
2 print(' '.join(a))           # red green blue
3 print(''.join(a))            # redgreenblue
4 print('***'.join(a))         # red***green***blue
5 s = '-'.join(a)
6 print(s + ' new string')     # red-green-blue new string
```

Основные строковые методы

Метод	Назначение
`ord(c)`	перевод символа (односимвольной строки) в его код ASCII
`chr(i)`	перевод код ASCII в символ (односимвольную строку)
`S.find(str,[start],[end])`	поиск подстроки в строке, возвращает номер первого вхождения или -1
`S.rfind(str,[start],[end])`	поиск подстроки в строке, возвращает номер последнего вхождения или -1
`S.replace(old,new)`	замена всех вхождений подстроки `old` на подстроку `new`
`S.split(slist)`	разбиение строки по разделителю и создание списка строк `slist`
`S.isdigit()`	состоит ли строка из цифр
`S.isalpha()`	состоит ли строка из букв
`S.isalnum()`	состоит ли строка из цифр или букв
`S.islower()`	состоит ли строка из символов в нижнем регистре
`S.isupper()`	состоит ли строка из символов в верхнем регистре
`S.isspace()`	состоит ли строка из неотображаемых символов
`S.istitle()`	начинаются ли слова в строке с заглавной буквы
`S.upper()`	преобразование строки к верхнему регистру
`S.lower()`	преобразование строки к нижнему регистру

Основные строковые методы (окончание)

Метод	Назначение
`S.startswith(str)`	начинается ли строка S с шаблона str
`S.endswith(str)`	заканчивается ли строка S шаблоном str
`S.capitalize()`	преобразование первого символа строки в верхний регистр, а всех остальных в нижний
`S.count(str,[start],[end])`	возвращает количество непересекающихся вхождений подстроки в диапазоне [start,end] (0 и длина строки по умолчанию)
`S.lstrip([chars])`	удаление пробельных символов в начале строки
`S.rstrip([chars])`	удаление пробельных символов в конце строки
`S.strip([chars])`	удаление пробельных символов в начале и в конце строки
`S.swapcase()`	преобразование символов нижнего регистра в верхний, а верхнего — в нижний
`S.title()`	преобразование первой буквы каждого слова в верхний регистр, а всех остальных в нижний
`S.format(*args, **kwargs)`	форматирование строки

1.5 Одномерные списки

Пример 1.23 (простейшие операции со списками)**.** Сложить два полинома одинаковой степени. Коэффициенты полинома целые числа.

Сложение полиномов

```
n = 5
f = [3, 2, 3, 0, 0, 0]  # инициализация списка f
g = []
h = []

# печать уже заданного списка f
print('Заданы коэффициенты полинома F')
for i in range(n+1):
    print('F[', i,'] = ', f[i], sep='', end=' ')

# ввод элементов списка g и процесс сложения полиномов
print('\nВв. целые коэф. полинома g степени не выше', n)
for i in range(n+1):
    # ввод элементов списка
    g.append(int(input('G['+ str(i) +'] = ')))
    # сложение полиномов
    h.append(f[i] + g[i])

print('--------------------')

for i in range(n, -1, -1):
    if (f[i] != 0 or g[i] != 0) and h[i] != 0:
        res = str(h[i]) +'x^' + str(i)
        print('%+6s' % res)
```

Использование списков в качестве параметров подпрограмм

Пример 1.24 (использование параметра-списка). Написать и использовать четыре подпрограммы:

`input_vector`	получение списка случайным образом
`print_vector`	печать списка
`equal_objects`	сравнение двух элементов списка
`create_vector`	формирование нового списка из чётных элементов исходного списка

Описания функций

```
import random

def print_vector(vector):
    for element in vector: print(element, end=' ')
    print()

def input_vector(count_elements, vector):
    for i in range(count_elements):
        x = random.randint(1, 5)
        vector.append(x)

def create_vector(vector_source):
    vector_result = []
    for element in vector_source:
        if element % 2 == 0: vector_result.append(element)
    return vector_result

def equal_objects(object_1, object_2):
    return object_1 == object_2
```

Основная часть программы

```
random.seed() # Настройка генератора случайных чисел
a = []
input_vector(5, a)
print_vector(a)
b = create_vector(a)
print_vector(b)
print(equal_objects(a[0], a[1]))
print(equal_objects(a, b))
```

Функция `create_vector` в качестве результата своей работы возвращает список. Эту функцию можно было записать короче, если использовать генератор списка.

Использование генератора списка

```
def create_vector(vector_source):
  return [element for element in vector_source
          if element % 2 == 0]
```

Функция `equal_objects` проверяет на равенство два любых объекта. Это могут быть не только списки или их элементы, но и просто символы, строки или другие объекты.

Примеры вызова функции equal_objects

```
print(equal_objects(a[0], a[1]))
print(equal_objects(a, b))
print(equal_objects('w', 'w'))
print(equal_objects(print_vector(a), print_vector(b)))
```

Приведенная в примере функция `input_vector` возвращает список через параметр функции. Ниже представлена функция, возвращающая список с помощью `return`.

Функция заполнения списка-2

```
import random

def input_vector(count_elements):
    vector = []
    for i in range(count_elements):
        x = random.randint(1, 5)
        vector.append(x)
    return vector

random.seed() # Настройка генератора случайных чисел
a = input_vector(5)
...
```

Пример 1.25 (поиск максимального элемента списка и его номера). Найти максимальный элемент в списке и значение индекса этого элемента.

Поиск максимального элемента и его индекса

```
import random
count_elements = 5

def idx_max_el(vector):  # ищем только номер элемента
    if not vector: return -1
    imax = 0
    for i in range(1, len(vector)):
        if vector[i] > vector[imax]: imax = i
    return imax

random.seed()
a = [random.randint(1, 5) for i in range(count_elements)]
print(a)
print('Max = a[%d] = %d' %(idx_max_el(a), a[idx_max_el(a)]))
```

```
# Tests
a = [5, 7, 12, 4, -1] # Test 1
assert idx_max_el(a) == 2, "Error in Test 1"
print('Test 1: OK')

a = []                # Test 2
assert idx_max_el(a) == -1, "Error in Test 2"
print('Test 2: OK')
```

Функция assert (утверждать) позволяет прервать выполнение программы, если логическое условие ложно. При этом выводится сообщение, которое указано после запятой. Сообщение может и отсутствовать.

Пример 1.26 (использование в качестве параметра элемента списка). Пусть имеются все приведенные выше описания. Напишем функцию для сравнения двух элементов списка. Результат вычисления — булев. Операция сравнения двух элементов списка ничем не отличается от операции сравнения двух чисел, т. е. вид функции может быть, например, такой

Сравнение двух элементов списка–1

```
def sravn1(x, y):
    if x == y:
        return True
    else:
        return False
```

Наиболее классический вид функции использует условный оператор.

Сравнение двух элементов списка–2

```
def sravn2(x, y):
    return x == y
```

Более компактный вид записи функции. Он является наиболее предпочтительным.

Вывод результата работы функции может быть помещен в тело программы, например:

Пример-1 вызова подпрограммы

```
print(sravn3(a[0], a[1]))
```

Сравниваются два первых элемента

Пример-2 вызова подпрограммы

```
print(sravn3(a[0], a[len(a)-1]))
```

Сравниваются первый и последний элементы

Заметим, что ошибочным при описании функции `sravn` мог быть следующий заголовок подпрограммы

Ошибочный заголовок функции

```
def sravn(a[i], a[j]):
```

Замена, удаление и вставка элементов

Пример 1.27 (обмен значениями двух списков). Произвести обмен значениями между двумя списками.

Обмен между двумя списками

```
a = [1, 2, 3]
b = [4, 5]
a, b = b, a
```

Обмен значениями с использованием кортежного присваивания.

Пример 1.28 (замена элементов списка по значению). Заменить все отрицательные элементы списка нулями.

Замена элементов списка по значению-1

```
a = [5, -2, 16, 8, -3]
for i in range(len(a)):
    if a[i] < 0: a[i] = 0
print(a)
```

Пример классической замены.

Результат:

```
[5, 0, 16, 8, 0]
```

Замена элементов списка по значению-2

```
def func(x):
    if x < 0: return 0
    return x

a = [5, -2, 16, 8, -3]
a = list(map(func, a))
print(a)
```

функция `map()` позволяет применить функцию `func()` ко всем элементам списка `a`.

Замена элементов списка по значению-3; lambda-функция

```
a = [5, -2, 16, 8, -3]
a = list(map(lambda x: 0 if x < 0 else x, a))
print(a)
```

Пример 1.29 (удаление элемента списка по номеру). Удалить элемент списка с заданным номером.

Удаление элемента с использованием метода списка pop. В качестве параметра указывается индекс удаляемого элемента. Метод `pop()` возвращает значение удаляемого элемента списка.

Удаление элемента списка по номеру

```
b = [8, 0, 3, 2, 5, 1, 4]
nom = 3
b.pop(nom) # удаление элемента с номером nom
print(b)
```

Пример 1.30 (вставка числа в список). Вставить заданное число в список на место с заданным номером.

В примерах ниже попутно продемонстрированы разные способы инициализации списка случайным образом.

Способ 1. Классический сдвиг элементов вправо.

Вставка числа в список-1

```
import random
'''
Настраиваем генератор случайных чисел
на новую последовательность.
По умолчанию используется системное время.
'''
random.seed()
a = [] # пустой список
n = 10 # количество элементов списка
a_min = 0 # минимальное значение элемента списка
a_max = 9 # минимальное значение элемента списка
for i in range(n):
    # генерация чисел из [a_min; a_max]
    a.append(random.randint(a_min, a_max))
print(a)
# ------------- вставка числа в список ------------------
x = int(input('Введите число для вставки: '))
k = int(input('На какую позицию будем вставлять? '))
# расширяем список на один элемента, добавив 0 в конец
a.append(0)
# увеличиваем количество элементов на один
n += 1
for i in range(n-1, k, -1):
    a[i] = a[i-1]   # сдвигаем вправо
a[k] = x
print(a)
```

Способ 2. Использование метода списка `insert()`.

Вставка числа в список-2

```
import random

MIN_EL = 0
MAX_EL = 9
COUNT_EL = 10

random.seed()
a = [random.randint(MIN_EL, MAX_EL) for i in range(COUNT_EL)]
print(a)
# ------------- вставка числа в список ------------------
x = int(input('Введите число для вставки: '))
k = int(input('На какую позицию будем вставлять? '))
a.insert(k, x)
print(a)
```

Способ 3. Использование среза списка.

Вставка числа в список-3

```
import random

COUNT_EL = 10

random.seed()
a = [random.choice(range(10)) for i in range(COUNT_EL)]
print(a)
# ------------- вставка числа в список ------------------
x = int(input('Введите число для вставки: '))
k = int(input('На какую позицию будем вставлять? '))
a[k:k] = [x]
print(a)
```

Методы сортировки в Python

`A.sort([key=функция])`	Сортирует список на основе функции

Пример 1.31 (сортировка средствами языка Pythom). Ниже приведены несколько примеров сортировок списков встроенными средствами языка Python.

Сортировка средствами Python

```
A = [13, 7, 8, 17, 13, 15, 16, 8, 21, 3, 37, 39, 40, 4, 13]
A.sort()
print(A)
A.sort(reverse=True)
print(A)
```

Результат работы программы

```
[3, 4, 7, 8, 8, 13, 13, 13, 15, 16, 17, 21, 37, 39, 40]
[40, 39, 37, 21, 17, 16, 15, 13, 13, 13, 8, 8, 7, 4, 3]
```

Python (сортировка по длине строки)

```
A = ['a', 'kotik' , 'cccc', 'bbb']
A.sort(key=len)
print(A)
```

Результат работы программы

```
['a', 'bbb', 'cccc', 'kotik']
```

Средства Python (сортировка по алфавиту I символа)

```
A = ['ar', 'car', 'as', 'bar', 'abt', 'abg']
print('Исх. список: ', A)

# Функция для сортировки списка в алф. порядке по I символу:
def sortByAlphabet1(inputStr):
   return inputStr[0] # Ключ - I символ в каждой строке

A.sort(key = sortByAlphabet1) #
print('По I букве:  ', A)
```

Результат работы программы

```
Исх. список:  ['ar', 'car', 'as', 'bar', 'abt', 'abg']
По I букве:   ['ar', 'as', 'abt', 'abg', 'bar', 'car']
```

Средства Python (сортировки по алфавиту)

```
A = ['ar', 'car', 'as', 'bar', 'abt', 'abg']
print('Исх. список:  ', A)

# Функция для сортировки списка в алф. порядке:
def sortByAlphabet(inputStr):
    return inputStr # Ключ - вся строка

A.sort(key = sortByAlphabet)
print('Алф. порядок: ', A)
```

Результат работы программы

```
Исх. список:   ['ar', 'car', 'as', 'bar', 'abt', 'abg']
Алф. порядок:  ['abg', 'abt', 'ar', 'as', 'bar', 'car']
```

Пример 1.32 (случайная сортировка)**.** Часто требуется элементы исходного списка перемешать, т. е. установить случайный порядок элементов.

В этом случае требуется использовать в качестве ключа функцию `random()`, которая выдает числа в случайном порядке от 0 до 1.

Средства Python (случайная сортировка)

```
from random import random

def randomOrder_key(elem):
    return random()
```

Примеры сортировки в случайном порядке

```
a = [1, 2, 3, 4, 5, 6]
print('Исх. список:  ', a)

b = sorted(a, key = randomOrder_key)
print(b)  # [5, 3, 2, 4, 6, 1]

c = sorted(a, key = randomOrder_key)
print(c)  # [2, 1, 5, 4, 3, 6]
```

1.6 Двумерные массивы (списки списков)

Ниже приведены фрагменты кода заполнения и печати массивов.

Ввод элементов массива с клавиатуры

```
N = int(input())
a = [0] * N
for i in range(N):
    a[i] = [0] * N
    for j in range(N):
        s = 'a[' + str(i) + '][' + str(j) + '] = '
        a[i][j] = int(input(s))
```

Печать двумерного массива в виде таблицы

```
a = [[1, 2, 3], [4, 5, 6]]
for i in range(len(a)):
    for j in range(len(a[i])):
        print(a[i][j], end=' ') # печать по столбцам
    print()                     # переход на новую строчку
```

Описание и вызов функции печати 2d-массива. Вариант 1

```
def printArr(a):
    for i in range(len(a[0])):
        for j in range(len(a)):
            print(a[i][j], end='\t')
        print()

A = [[1, 2, 3], [4, 5, 6]]
printArr(A)
```

Переменная цикла for в Python может перебирать любые элементы любой последовательности (в частности, списки). Используем это свойство для печати массива.

Описание и вызов функции печати 2d-массива. Вариант 2

```
def printArr2(a):
    for row in a:
        for elem in row:
            print(elem, end='\t')
        print()

A = [[1, 20, 3, 4], [55, 6], [7, 8, 9]]
printArr2(A)
```

Пример 1.33 (генераторы списков)**.** Более простой пример инициализации списка списков с помощью генератора: массив 3×4 заполняется нулями.

Генерация списков

```
N = 3
M = 4
a = [[0] * M for i in range(N)]
```

1.7 Множества

Множество состоит из различных элементов, порядок которых в множестве не определен.

Элементы множества: данные неизменяемых типов (числа, строки, кортежи).

Пример 1.34 (задание непустого множества)**.** Множество задается перечислением всех его элементов в фигурных скобках:

`A = {1, 2, 3}`.

Каждый элемент может входить в множество только один раз. Если задать множество следующим образом

`A = {3, 1, 2, 3, 3, 2, 2}`,

то будет создано множество `{1, 2, 3}`.

Пример 1.35 (задание пустого множества и функция `set()`)**.** Функция `set()` используется для создания множества.

Пустое множество задается следующим образом

`A = set()`.

Создание множества	Пример	Результат
из строки	`A = set('racoon')`	множество из 5 элементов
из списка	`A = set([1, 6, 7, 2, 1])`	`{1, 2, 6, 7}`

Пример 1.36 (использование генератора для создания множества)**.** Создать множество целых чисел от 0 до $n-1$ можно с помощью генератора

```
d = {i for i in range(n)}
```

Основные приемы работы с данными множественного типа

Количество элементов в множестве определяется с помощью функции `len()`.

Для проверки принадлежности элемента множеству используется зарезервированное слово `in`:

Демонстрация len() и in

```
b = {1, 3, 3, 5, 9}
print('кол-во элементов: ', len(b))                  # рез-тат 4
print('наличие элемента 4 в b: ', 4 in b)            # False
print('отсутствие элемента 4 в b: ', not 4 in b)  # True
print('отсутствие элемента 3 в b: ', 3 not in b)  # False
```

Множества легко вывести на экран (порядок вывода элементов в множестве может быть непредсказуем)

Печать множеств

```
a = set()
b = {1, 3, 3, 5, 9}
c = set('hello world!')
print(a, b, c, sep='\n')
```

Результат-1

```
set()
{1, 3, 5, 9}
{'!', 'h', 'o', 'r', 'l', 'w', ' ', 'd', 'e'}
```

Результат-2

```
set()
{1, 3, 5, 9}
{'d', ' ', 'l', 'h', 'o', 'e', 'r', '!', 'w'}
```

Пример 1.37 (поэлементная печать множества). Для печати элементов множества можно использовать цикл `for`:

Печать множеств

```
word = set('programming')
for c in word:
    print(c, end='')
```

Примеры результатов

```
armpgnio  # первый результат
prgonmai  # второй результат
```

Пример 1.38 (преобразование множеств). Множества можно преобразовывать в строку, список или кортеж, используя соответствующие методы.

Преобразование множества к другому типу

```
Set = set('programming')  # множество
List = list(Set)          # список
Tuple = tuple(Set)        # кортеж
Str = str(Set)            # строка
```

Методы добавления, удаления элементов из множества

Метод	Описание
`add(x)`	добавление элемента x в множество
`discard(x)`	удаление элемента x из множества
`remove(x)`	удаление элемента x из множества. Если удаляемый элемент отсутствует в множестве, то генерируется исключение `KeyError`
`pop()`	удаляет из множества первый элемент и возвращает его значение. Если множество пусто, то генерируется исключение `KeyError`. Примечание: так как множество не упорядочено, какой элемент будет первым, неизвестно
`clear()`	очистка множества

Пример 1.39 (добавление элемента в список и очистка списка). Метод `add()` добавляет элемент в множество.

Добавление элемента в список и очистка списка

```
s = {1, 2, 10}
print(s)  # {1, 2, 10}
s.add(5)
print(s)  # {1, 2, 10, 5}
s.add(7)
print(s)  # {1, 2, 5, 7, 10}
s.clear()
print(s)  # set()
```

Пример 1.40 (удаление случайного элемента). Использование метода `pop()` позволяет не только удалить элемент из множества, но и запомнить значение удаленного элемента. Если применить метод `pop()` к пустому списку, то генерируется исключение `KeyError` — несуществующий ключ (во множестве, в данном случае). Для обработки исключений используется конструкция `try-except`.

]sf Удаление случайного элемента

```
s1 = {-1, 2, 10}
print(s1)    # {2, 10, -1}

s1.pop()
print(s1)    # {10, -1}

d = s1.pop()
print(s1, 'удалено число', d)    # {-1} удалено число 10

# обработка исключения,
# если происходит удаление из пустого списка
s1 = {}
try:
    # инструкция, которая может породить исключение
    s1.pop()
except Exception:
    print('Error')  # перехват исключения
print(s1)    # {}
```

Пример 1.41 (сравнение методов удаления элемента из списка). Методы `discard(x)` и `remove(x)` удаляют элемент x из множества. Метод `remove(x)` генерирует исключение `KeyError`, при попытке удалить отсутствующий элемент.

Удаление элемента из списка

```
s = {1, 2, 10}
s.discard(2)
print(s)     # {1, 10}

# попытка удалить несущ-щий элемент ошибку не вызывает
s.discard(5)
print(s)     # {1, 10}

#-----------------------------------------
s = {1, 2, 10}
s.remove(2)
print(s)     # {1, 10}

'''
# попытка удалить несущ-щий элемент вызывает ошибку
s.remove(5)
print(s)
'''
```

Функции с логическим результатом

`b.isdisjoint(c)`	проверяет нет ли пересечения множеств (True, если нет)
`b.issubset(c)` `b <= c`	$b \subset c$ (True, если все элементы b принадлежат c)
`b.issuperset(c)` `b >= c`	$b \supset c$ (True, если все элементы c принадлежат b)

Пример 1.42 (демонстрация методов `isdisjoint()`...). Продемонстрируем работу методов `isdisjoint()`, `issubset()`, `issuperset()`.

Демонстрация методов isdisjoint(), issubset(), issuperset()

```
A = {1, 2, 10}
B = {1, 12, 100}
C = {11,14}
print(A.isdisjoint(B))  # False. Нет ли пересечения?
print(A.isdisjoint(C))  # True

#------------------------------------------
B = {1, 12, 100}
C = {1, 100}
print(B.issubset(C))    # False. Все эл-ты b принадлежат c
print(B.issuperset(C))  # True. Все эл-ты c принадлежат b
```

Объединение, пересечение, разность множеств

`a.update(b)`	добавление элементов множества b в a
`F = b.union(c, d)` `F = b | c | d)`	объединение множеств ($F = b \cup c \cup d$)
`F = b.intersection(c, d)` `F = b & c & d`	пересечение ($F = b \cap c \cap d$).
`F = A.difference(B)` `F = A - B`	разность множеств A и B (элементы, входящие в A, но не входящие в B)

Пример 1.43 (метод `update()`). Метод `update()` служит для добавления элементов одного множества в другое.

Демонстрация метода update()

```
A = {'a', 'c'}
B = {1, 12, 100}
A.update(B)  # добавление элементов множества B в A
print(A)     # {1, 'c', 100, 'a', 12}
```

Пример 1.44 (объединение, пересечение и разность множеств). Продемонстрируем работу методов для объединения, пересечения нескольких множеств, а также разности множеств.

Объединение множеств

```
A = {'a', 'c'}
B = {1, 12, 100}
C = {1, 100}
D = {-5, -7}
S1 = B.union(C, D)
print(S1)             # {1, 100, -7, -5, 12}
S11 = set.union(B, C, D)
print(S11)            # {1, 100, -7, -5, 12}
```

Пересечение множеств и разность множеств

```
A = {12, 5, 100, -3}
B = {1, 12, 100}
C = {1, 100}
D = {-5, -7, 1}
P = B.intersection(C, D) # пересечение (или P = B & C & D)
print(P) # {1}
#------------------------------------------
R = A.difference(B) # разность (или R = A - B)
print(R) # {5, -3}
```

2 Пакет SymPy

SymPy — это библиотека для символьных вычислений в Python. В документации пакета очень кратко описаны используемые классы, зато приведено большое количество примеров применения методов пакета, благодаря чему можно довольно легко освоить данный пакет для решения поставленных научных задач.

Основные операции, реализованные в пакете SymPy:

1. упрощение выражений, раскрытие скобок;
2. разложение в ряд функций;
3. вычисление сумм рядов (последовательностей);
4. вычисление пределов;
5. вычисление производных функций;
6. вычисление интегралов;
7. работа с матрицами (модуль линейной алгебры);
8. вычисление площадей фигур, образованных при пересечении прямых, отрезков и лучей (модуль геометрии);
9. получение случайных величин с заданной функцией распределения плотности вероятности (модуль статистики);
10. построение графиков функций и трёхмерных поверхностей, заданных в виде уравнений с символьными переменными;
11. возможность красивой печати символьных выражений в различных форматах.

На сайте `http://www.sympy.org/ru/index.html` можно более подробно ознакомится с пакетом **SymPy**.

Пример 1. Использование **SymPy** в качестве калькулятора

```
>> from sympy import*
>> a = Rational(1, 2)
>> b = Rational(1, 3)
>> print(a, b, sep = ', ')
1/2, 1/3
>> c = a + b
>> print(c)
5/6
>> print(c**2)
25/36
```

Функция `Rational(int1, int2)` представляет собой обыкновенную дробь, которая задаётся с помощью двух целых чисел: числителя и знаменателя. Деление двух целых чисел с помощью оператора «/» возвращает вещественное значение — десятичную дробь. Для работы с обычными дробями нужно использовать функцию `Rational`.

В пакете **SymPy** имеются особые константы, такие как `e` и `pi`, которые ведут себя как переменные. Для использования этих констант при численных вычислениях нужно использовать функцию `evalf()`. Если в качестве параметра функции `evalf([int])` указать целое число, то вернется вещественная дробь с указанным количеством значащих цифр.

Кроме этого, есть возможность работы с математической бесконечностью, которая в пакете **SymPy** обозначается как `oo`.

Пример 2. Константы в пакете **SymPy**

```
>> from sympy import*
>> print(pi + 1)
1 + pi
>> print(E.evalf(), E.evalf(5), sep = ', ')
2.71828182845905, 2.7183
>> print(oo * 3)
oo
```

Для работы с символьными переменными необходимо описывать их в явном виде. Сделать это можно при помощи функций `Symbol()` (для описания одной переменной) и `symbol()`, `var()` (для описания нескольких переменных).

При описании нескольких переменных допустимо перечислении как через запятую, так и с указанием диапазона (см. пример 3). Отличие функций `symbol()` и `var()` состоит в том, что `var()` добавляет созданные переменные в текущее пространство имен.

Описанные символьные переменные взаимодействуют друг с другом. Таким образом, можно конструировать различные алгебраические выражения. При этом для более наглядного вывода символьных выражений в модуле **SymPy** предусмотрена специальная функция

`pprint(выражение)` (Pretty-print).

Для раскрытия скобок в символьных выражениях используется функция

`expand(выражение)`,

а для замены переменных на другие переменные, числовые значения и выражения используется функция

`subs(old, new)`.

Пример 3. Основные операции с символьными переменными

```
>> from sympy import*
>> x = Symbol('x')
>> y, z = symbols('y z') # или symbols('y, z')
>> a, b, c = symbols('a:c')
>> print(a, b, c, x, y, z)
a b c x y z
>> print(var('d:f k p:t u w'))
(d, e, f, k, p, q, r, s, t, u, w)
>> print(var('f:6'))
(f0, f1, f2, f3, f4, f5)
>> C = a * a - b - c + 2 * b
>> print(C)
```

```
a**2 + b - c
>> pprint(C)
 2
a  + b - c
>> f1 = (x + y) * z
>> print(f1)
z*(x + y)
>> print(f1.expand())
x*z + y*z
>> f2 = f1.subs(z, 4)
>> print(f2)
4*x + 4*y
>> f3 = f2.subs(x, y - Rational(1, 4))
>> print(f3)
8*y - 1
```

Стоит обратить особое внимание на некоторые особенности, в частности, на вычисление значений с рациональными степенями. Рассмотрим пример.

Пример 4. Вычисление значения кубического корня

```
>> import sympy.abc
>> var('x f')
>> f = x**(1/3)
>> print(f.subs(x, 27))
3.00000000000000
>> print(f.subs(x, -27))
1.5 + 2.59807621135332*I
```

На первый взгляд может показаться, что вычисления некорректные. На самом деле, это не совсем так. Поясним полученные результаты. При подстановке в функцию `f` какого-либо значения первое, что происходит — это вычисление значения степени, т. е. обычная дробь $1/3$ представляется

в виде десятичной дроби 0,333333. Именно поэтому первое полученное значение — вещественное число, а не целое. При подстановке отрицательного значения в корень кубический ожидался ответ -3. Однако на вывод получено комплексное число (о работе с комплексными числами в **SymPy** написано в пункте 2.4).

Для того чтобы исправить результаты, полученные в примере 4, нужно использовать специальную функцию

`real_root()`,

описанную в пакете **SymPy**.

Пример 5. Вычисление значения кубического корня–2

```
>> import sympy.abc
>> var('x f')
>> f = real_root(x, 3)
>> print(f.subs(x, 27))
3
>> print(f.subs(x, -27))
-3
```

Теперь результаты вычислений соответствуют нашим ожиданиям.

Но стоит всё-таки разобраться с тем комплексным числом, которое получалось при вычислении $\sqrt[3]{-27}$ в примере 4. Дело в том, что заданная функция `f` имеет три корня: один вещественный и два комплексно сопряжённых. Функция `root()` (родственная функции `real_root()`) может иметь третий параметр, который будет соответствовать номеру корня (нумерация начинается с нуля).

Пример 6. Вычисление значения кубического корня–3

```
>> print(root(-27, 3, 0).evalf())
1.5 + 2.59807621135332*I
>> print(root(-27, 3, 1).evalf())
-3.00000000000000
>> print(root(-27, 3, 2).evalf())
1.5 - 2.59807621135332*I
```

Таким образом, прямое вычисление корня кубического в примере 4 даёт нам самый первый корень.

Также стоим отметить, что в пакете **SymPy** имеются предопределённые символы (прописные и строчные латинские и строчные греческие буквы плюс несколько строк), которые можно импортировать из пакета sympy.abc. Таким образом, подключив пакет sympy.abc можно без описания использовать прописанные там символьные переменные.

Пример 7. Предопределённые символы

```
>> import sympy.abc
>> # Список предопределённых символов
>> print(print(dir(sympy.abc)))
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__spec__', '_clash', '_clash1', '_clash2', 'a', 'alpha',
'b', 'beta', 'c', 'chi', 'd', 'delta', 'division', 'e',
'epsilon', 'eta', 'exec_', 'f', 'g', 'gamma', 'greeks',
'h', 'i', 'iota', 'j', 'k', 'kappa', 'l', 'lamda', 'm',
'mu', 'n', 'nu', 'o', 'omega', 'omicron', 'p', 'phi', 'pi',
'print_function', 'psi', 'q', 'r', 'rho', 's', 'sigma',
'string', 'symbols', 't', 'tau', 'theta', 'u', 'upsilon',
'v', 'w', 'x', 'xi', 'y', 'z', 'zeta']
```

При работе с дробно-рациональными выражениями часто необходимо приводить дроби к общему знаменателю или, наоборот, разложить выражение на простые дроби. В Python для этого в пакете **SymPy** реализованы функции

`together(exp)` и `apart(exp)`

соответственно.

Пример 8. Работа с дробными выражениями

```
>> from sympy import*
>> from sympy.abc import*
>> pprint((x**2 - 3) / (x + 1))
 2
x  - 3
------
x + 1
>> pprint(apart((x**2 - 3) / (x + 1)))
          2
x - 1 - ------
        x + 1
>> f = 1/x - 1/y + 1/z
>> pprint(f)
1   1   1
- - - + -
z   y   x
>> pprint(together(f))
x*y - x*z + y*z
---------------
     x*y*z
```

Если установлен шрифт с юникодом, он будет использовать Pretty-print с юникодом по умолчанию. Эту настройку можно отключить, используя параметр

`use_unicode = False`.

И наоборот, для включения юникода следует добавить в `pprint()` параметр `use_unicode = True`.

2.1 Функции в SymPy

В пакете **SymPy** имеется возможность работы с различными функциями, такими как: логарифмические, тригонометрические, экспоненциальные, сферические, факториал, многочлены (Чебышёва, Лежандра и т. д.), гамма-функции и др.

Пример 9. Использование функций в **SymPy**

```
>> from sympy import*
>> from sympy.abc import*
>> print(cos(x)/sin(x))
cos(x)/sin(x)
>> print(exp(x).subs(x, 2), '~', exp(x).subs(x, 2).evalf(6))
exp(2) ~ 7.38906
>> print('5! =', factorial(5))
5! = 120
```

2.2 Степенные ряды в SymPy

Для разложения функции в степенной ряд используется метод

series(var, point, order).

Пример 10. Разложение функции в степенной ряд

```
>> from sympy import*
>> from sympy.abc import*
>> print(sin(x).series(x, 0, 7))
x - x**3/6 + x**5/120 + O(x**7)
>> pprint(sin(x).series(x, 0, 7), use_unicode = False)
     3     5
    x     x     / 7\
x - -- + --- + O\x /
    6    120
>> pprint(sin(x).series(x, 0, 7), use_unicode = True)
     3     5
```

```
    x     x    ( 7)
x - - +  -- + O(x )
    6    120
```

2.3 Печать математических формул в LaTeX

Библиотека **SymPy** предоставляет возможность выводить на печать математические формулы в формате LaTeX. Для этого необходимо воспользоваться функцией `latex(exp)`. А указав нужное значение параметра `mode` в этой функции, можно получить внутристрочную или выделенную в отдельную строку (нумерованную или ненумерованную) формулы (см. также п. 5).

Пример 11. Печать формул в формате LaTeX

```
>> from sympy import*
>> from sympy.abc import*
>> print(latex(1 / x**2))
\frac{1}{x^{2}}
>> print(latex(1 / x**2, mode = 'inline'))
$\frac{1}{x^{2}}$
>> print(latex(1 / x**2, mode = 'equation'))
\begin{equation}\frac{1}{x^{2}}\end{equation}
>> print(latex(1 / x**2, mode = 'equation*'))
\begin{equation*}\frac{1}{x^{2}}\end{equation*}
```

2.4 Работа с выражениями с комплексными значениями

При работе с комплексными числами помимо мнимой единицы `I` в пакете **SymPy** имеются специальные атрибуты (`real`, `positive`, `complex` и т.д), которые определяют поведение этих символов при вычислении символьных выражений.

Пример 12. Комплексные числа

```
>> from sympy import*
>> x = Symbol('x')
```

```
>> print(exp(I*2*x).expand())
exp(2*I*x)
>> print(exp(I*2*x).expand(complex = True))
I*exp(-2*im(x))*sin(2*re(x)) + exp(-2*im(x))*cos(2*re(x))
>> x = Symbol('x', real = True)
>> print(exp(I*2*x).expand(complex = True))
I*sin(2*x) + cos(2*x)
```

Стоит обратить внимание, что при работе с комплексными значениями, не импортируется пакет sympy.abc, т. к. I не будет восприниматься компилятором как мнимая единица, а будет обрабатываться как символ.

2.5 Алгебраические уравнения

Библиотека **SymPy** даёт возможность решать линейные алгебраические уравнения. Для этого используется функция

`solve(func, var).`

Пример 13. Решение линейных алгебраических уравнений

```
>> from sympy import*
>> from sympy.abc import*
>> print(solve(x**2 + 2*x - 15, x))
[-5, 3]
>> print(solve(x**4 - 1, x))
[-1, 1, -I, I]
>> print(solve([-3*x + 5*y + 5, -3*x - 8*y + 31], [x, y]))
{x: 5, y: 2}
```

2.6 Суммирование рядов

Для вычисления суммы $\sum_{i=a}^{b} f(x)$ функции $f(x)$ по заданным значениям переменной $x \in [a,b]$ используется функция

summation(f, (i, a, b)).

При этом верхний предел суммирования может быть и бесконечным. Если функция summation() не может вычислить значение суммы, то будет выведена соответствующая формула суммирования. Так же имеется возможность вычисления кратных сумм, путём введения дополнительных пределов (при этом используется единственная функция суммирования).

Пример 14. Суммирование

```
from sympy import*
from sympy.abc import*
print(summation(i, (i, 1, 10)))
print(summation(1 / 2**i, (i, 1, oo)))
print(summation((-1)**i * x**(2*i) / factorial(2*i),
                (i, 0, oo)))
print(summation(k**2, (k, 1, m), (m, 1, n)))
pprint(summation(log(x), (x, 1, n)))
```

Результат работы программы в окне вывода Python Shell приведен ниже на рис. 5.

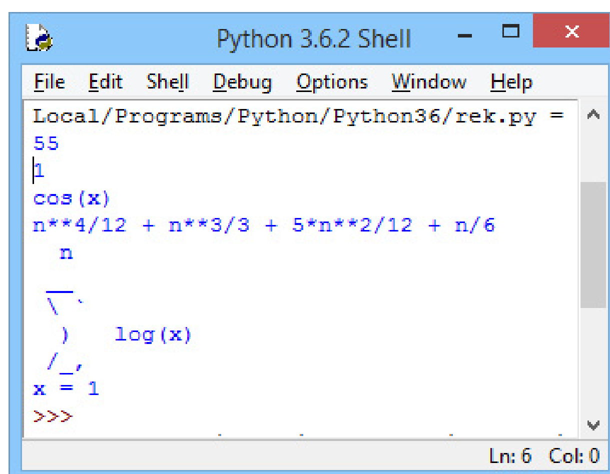

Рис. 5: Результат работы программы из примера 14

2.7 Вычисление пределов

Чтобы вычислить предел функции в **SymPy**, используется функция

`limit(func, var, point).`

Значение параметра `point` может быть как конечным, так и бесконечным.

Пример 15. Пределы

```
>> from sympy import*
>> from sympy.abc import*
>> print(limit((2*x**2 - 3*x - 5) / (x + 1), x, -1))
-7
>> print(limit(1 - x, x, oo))
-oo
>> print(limit(x**x, x, 0))
1
```

2.8 Дифференцирование

Продифференцировать любое выражение можно, используя метод

`diff(func, var, [int]).`

Параметр `int` определяет порядок производной по указанной переменной и не является обязательным (по умолчанию он равен единице, т. е. определяется производного первого порядка).

Пример 16. Дифференцирование

```
>> from sympy import*
>> from sympy.abc import*
>> print(diff(cos(x), x))
-sin(x)
>> print(diff((x**3 - 1)/x, x, 2))
2*(x**3 - 1)/x**3
>> print(diff(sin(x*y), y))
x*cos(x*y)
```

2.9 Интегрирование

SymPy поддерживает вычисление интегралов с помощью функции

`integrate()`.

Для неопределённых интегралов синтаксис функции имеет вид

`integrate(func, var)`,

а для определённых интегралов добавляется указание пределов интегрирования

`integrate(func, (var, a, b))`.

Можно вычислять интегралы трансцендентных, простых и специальных (см. п. 2.1) функций. Возможно также вычислять несобственные интегралы.

Пример 17. Интегрирование

```
>> from sympy import*
>> from sympy.abc import*
>> print(integrate(exp(-x), x))
-exp(-x)
>> print(integrate(exp(-x), (x, 0, 2)))
-exp(-2) + 1
>> print(integrate(exp(-x), (x, 0, 2)).evalf())
0.864664716763387
>> print(integrate(exp(-x), (x, 0, oo)))
1
```

2.10 Дифференциальные уравнения

В пакете **SymPy** кроме определения символьных переменных имеется возможность описывать функции. Для этого используется команда `Function()`. Переменные описанные как функции могут использоваться с указанием переменной, от которой эта функция зависит.

Функция `diff()` может использоваться не только для вычисления производных функции, но и для записи дифференциальных уравнений, которые могут быть решены в Python при помощи функции `dsolve()`.

Пример 18. Дифференциальные уравнения

```
from sympy import*
from sympy.abc import*
f = Function('f')
pprint(f(x).diff(x, x) + f(x).diff(x) - 2*(x+1))
pprint(dsolve(f(x).diff(x, x) + f(x).diff(x)-2*(x+1), f(x)))
```

Результат работы программы приведен на рис. 6.

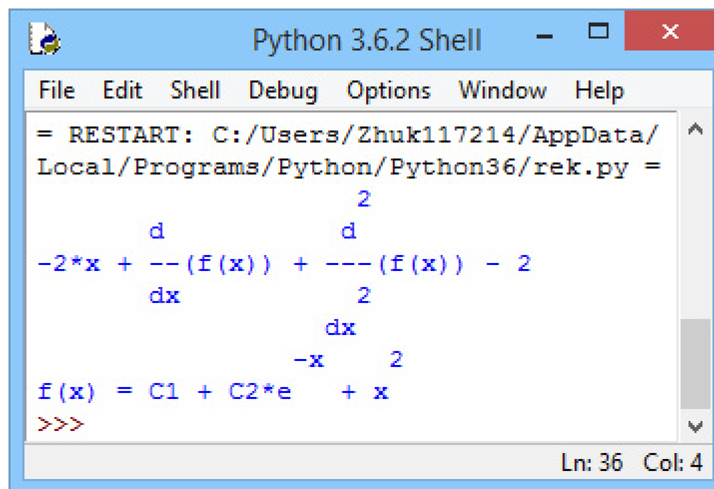

Рис. 6: Результат работы программы из примера 18

Если опустить описание переменной `f` как функции, то для Python это будет просто символьная переменная из библиотеки `sympy.abc`. Поэтому запись `f(x).diff(x)` будет некорректной и приведет к ошибке.

2.11 Матрицы

Для работы с векторами и матрицами в **SymPy** добавлен конструктор `Matrix()`.

Пример 19. Матрицы

```
>> from sympy import*
>> from sympy.abc import*
>> pprint(Matrix([1, 0, 0]))
[1]
[ ]
[0]
[ ]
[0]
>> pprint(Matrix([[1, 0, 0], [0, 0, 1]]))
[1  0  0]
[       ]
[0  0  1]
>> A = Matrix([[1, a], [b, 2]])
>> pprint(A**2)
[a*b + 1    3*a  ]
[                ]
[  3*b    a*b + 4]
```

2.12 Построение графиков функций и поверхностей

В **Sympy** имеется возможность построения двухмерных и трёхмерных образов при помощи функций `plot()` и `plot3d()` из пакета sympy.plotting.

Пример 20. Графика

```
from sympy.abc import*
from sympy.plotting import*
plot(x**2, (x, -5, 5))
plot(x, x**2, x**3, (x, -5, 5))
plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
```

После выполнения каждой команды `plot()` или `plot3d()` открывается новое окно выводла (см. рис. 7, 8). В этих окнах имеется возможность задания масштаба, положения и размера рисунка. Кроме этого, можно сохранить полученное изображение в удобном для дальнейшего использования формата (например, для вставки рисунка в tex-документ).

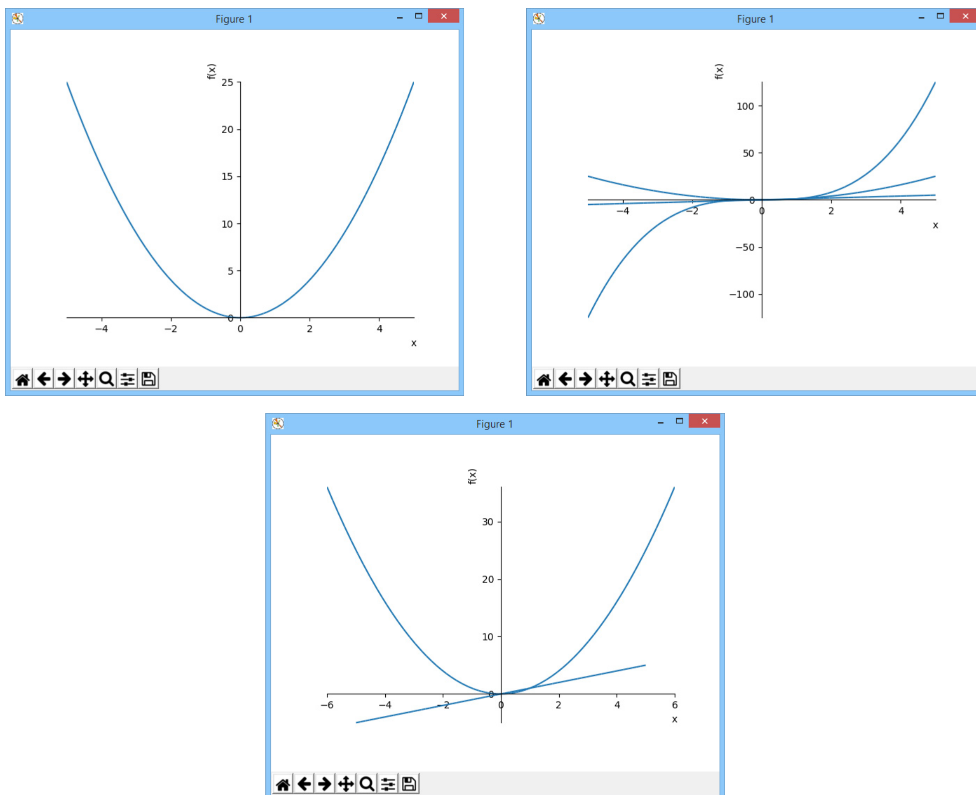

Рис. 7: Результат работы программы из примера 20

Пример 21. Графика

```
from sympy import *
from sympy.abc import*
from sympy.plotting import*
import math

F1 = lambda x: 0.5*x*x + 3*x
F2 = lambda x: 4*sin(2*x)
a, b = -math.pi, math.pi

q = plot((F1(x), (x,a,b)), (F2(x), (x,a,b)), show = False)
# установка цвета для графиков
q[0].line_color = 'blue'
q[1].line_color = 'red'
q.show()
```

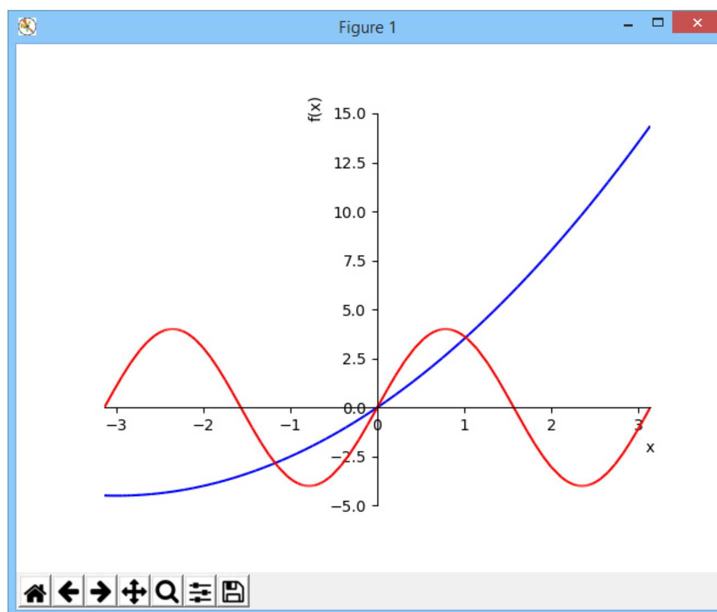

Рис. 8: Результат работы программы из примера 21

Пример 22. Графика

```
from sympy import*
from sympy.abc import*
from sympy.plotting import*
f = lambda x, y: x**2 + y**2
g = lambda x, y: x * y
plot3d(f(x,y))
plot3d(g(x,y), (x, -5, 5), (y, -5, 5))
plot3d((2*f(x,y), (x, -5, 5), (y, -5, 5)),
       (g(x,y), (x, -3, 3), (y, -3, 3)))
plot3d((log(f(x,y)), (x, -5, 5), (y, -5, 5)))
```

После выполнения каждой команды `plot3d()` открывается новое окно с трехмерной поверхностью, которую можно вращать (см. рис. 9.

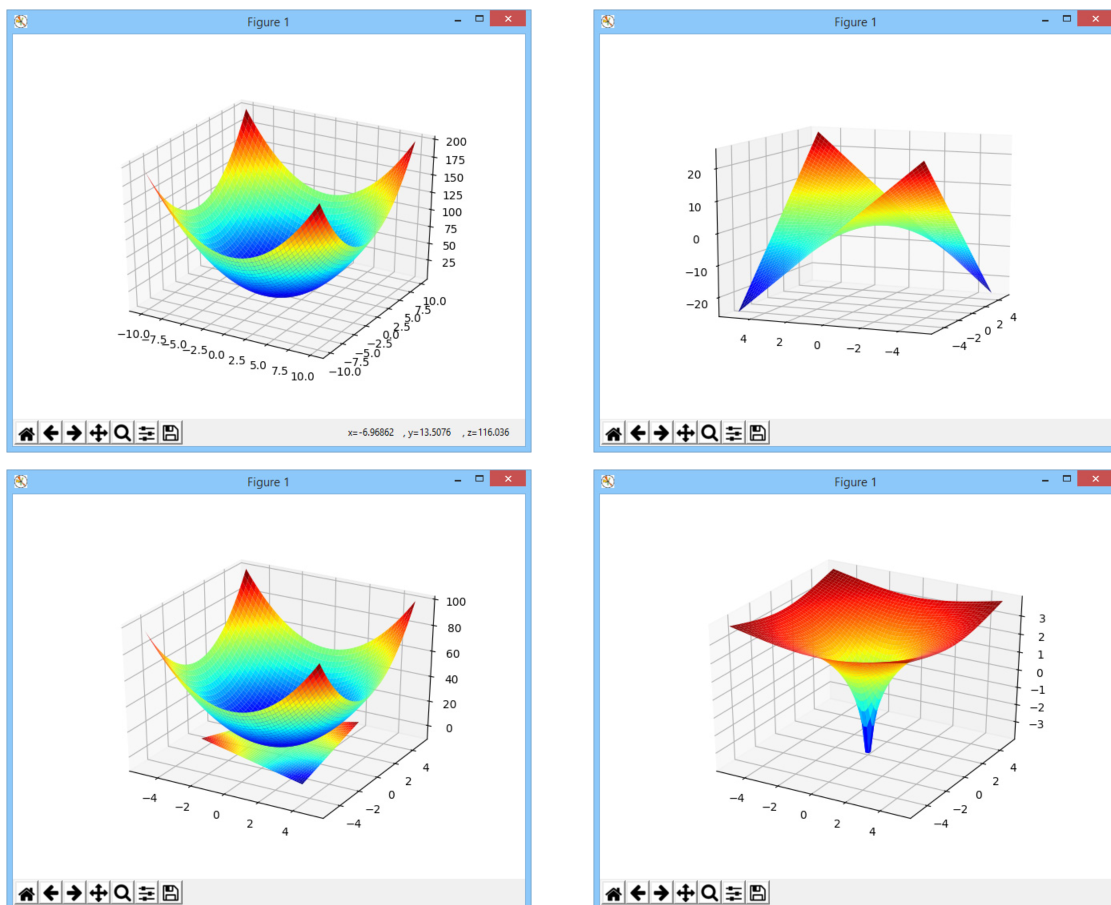

Рис. 9: Результат работы программы из примера 22

3 Пакет NymPy

Пакет **NumPy** является одним из необходимых при организации научных вычислений в Python. Самыми востребованными возможностями этой библиотеки являются работа с массивами и полиномами. К недостаткам данного пакета можно отнести то, что нет бесплатного полного руководства пользователя. Но некоторые часто встречающиеся функции можно посмотреть на сайте `http://www.numpy.org/`

Задаются массивы при помощи функций `array()` и/или `matrix()`, в которых кроме размера и значений элементов массива можно указывать и тип элементов (например, параметр `'f'` преобразует элементы в вещественные).

Доступны также функции для линейной алгебры: вычисление определителя матрицы, вычисление обратных матриц, решение системы линейных уравнений и т. п.

Отметим, что в пакете **NumPy** имеется две команды для перемножения матриц: поэлементное перемножение (оператор `'*'`) и обычное умножение матриц (функция `dot()`).

Пример 23. Работа с массивами в NumPy

```
>> from numpy import*
>> a = array([[1, 2, 3], [4, 5, 6]])
>> print(a)
[[1 2 3]
 [4 5 6]]
>> b = array([[0, -3, 6], [-6, -1, 2]], 'f')
>> print(b)
[[ 0. -3.  6.]
 [-6. -1.  2.]]
>> c = a + b
>> print(c)
[[ 1. -1.  9.]
 [-2.  4.  8.]]
>> C = a * b # поэлементное перемножение
>> print(C)
```

```
[[  0.  -6.  18.]
 [-24.  -5.  12.]]
>> b = b.T   # определение транспонированной матрицы
>> print(b)
[[ 0. -6.]
 [-3. -1.]
 [ 6.  2.]]
>> C2 = dot(a, b) #умножение двух матриц
>> print(C2)
[[ 12.  -2.]
 [ 21. -17.]]
```

4 Библиотека Matplotlib

Matplotlib — библиотека для построения графиков и визуализации данных. Имеются проблемы с отображением на графиках русских букв, но зато можно выводить формулы в виде LaTeX. Графики, нарисованные с помощью **Matplotlib** можно масштабировать, причём как с использованием специальных команд, так и через интерфейс с помощью мыши.

Для ознакомления с **Matplotlib** рекомендуем посетить

`http://matplotlib.org/`,

где кроме списка основных функций можно найти множество различных примеров построений различных графиков, трёхмерных поверхностей, гистограмм и пр.

Пример 24. Графика в Matplotlib

```
from math import*
# pylab - пакет из библ. matplotlib со спец. возможностями
from pylab import*
from matplotlib import*
def func (x):
    if x == 0: return 1.0
    return sin (x) / x
# Интервал изменения переменной по оси X и шаг
xmin, xmax = -10.0, 10.0
dx = 0.05                    # Шаг между точками
# Создаем список координат по оси X
# на отрезке [-xmin; xmax], включая концы
xlist = frange (xmin, xmax, dx)
# Вычисляем значение функции в заданных точках
ylist = [func (x) for x in xlist]
# Рисуем одномерный график
plot(xlist, ylist)
# Окно с нарисованным графиком
show()
```

Пример 25. Графика в Matplotlib

```
from math import*
from pylab import*
from matplotlib import*

func = lambda x: x*cos(x)

xmin, xmax = -10.0, 10.0
dx = 0.05
xlist = frange(xmin, xmax, dx)
ylist = [func(x) for x in xlist]
ylist2 = [func(0.4*x) for x in xlist]
plot(xlist, ylist)
plot(xlist, ylist2)
show()
```

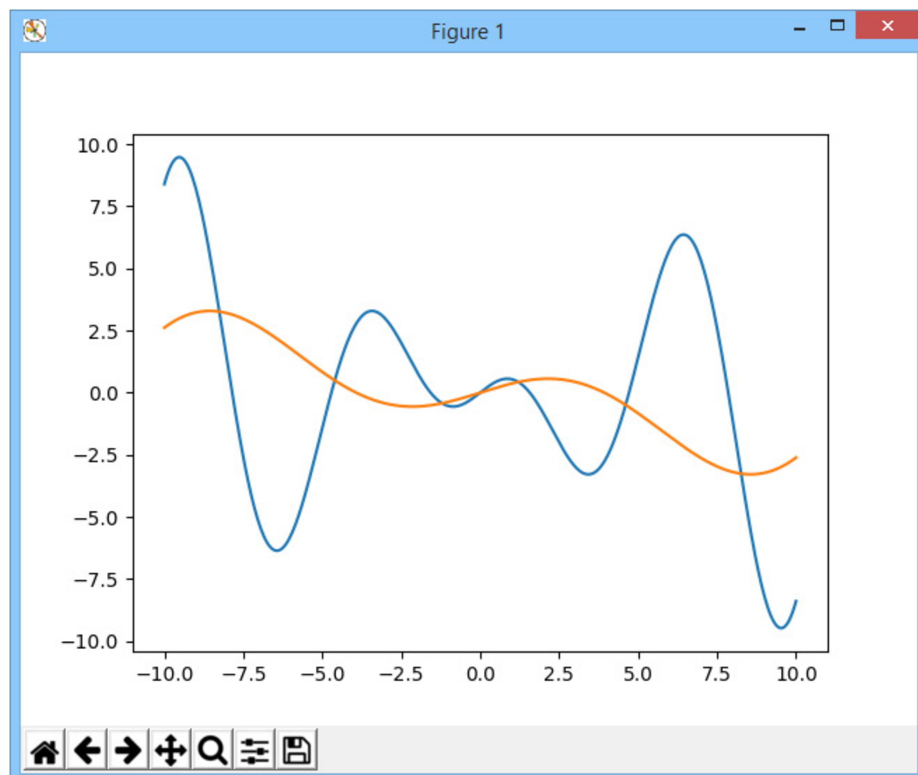

Рис. 10: Результат работы программы из примера 25

5 Python и LaTeX

Система компьютерной верстки LaTeX, помимо оформления описаний скриптов Python, может эффективно использоваться и для вызова интерпретатора Python непосредственно из LaTeX.

5.1 Вызов интерпретатора Python из системы LaTeX

Для выполнения скриптов языка Python в LaTeX требуется вызов пакета

```
12 \usepackage{python}
```

который предоставляет окружение

```
16 \begin{python}
```

```
41 \end{python}
```

Внутри этого окружения можно записывать любой скрипт на языке Python.

Далее для определенности считаем, что на компьютере установлен MikTex 2.9, текстовый редактор WinEdt версии не ниже 6.0 и используется PDFLaTex.

Замечания.

1. MikTex 2.9 имеет в своем составе стилевой файл python.sty.

Однако, если MikTex 2.9 был установлен достаточно давно, то такой стилевой файл может быть устаревшим и его следует обновить на последнюю версию — отыскать python.sty в интернете и просто заменить им имеющийся ранее стилевой файл.

2. Компилятор PDFLaTex следует вызывать с ключом -shell-escape.

Для этого в \WinEdt следует задать указанный ключ

Option - > Execution Modes...

в окне Accessories выбрать PDFLaTeX и в поле Switches: записать строку --shell-escape

3. Скрипт Python обязательно должен содержать первую строку пустой и начинающейся с символа # (см. ниже строку 17).

4. Скрипт Python не должен содержать русские буквы (решение проблем с кодировкой см. в интернете).

5.2 Использование модуля sympy

Приведем пример LaTeX-файла, который осуществляет подключение модуля sympy.

```
1  % & -shell-escape
2  \documentclass{article}
3  \usepackage{amssymb}
4  \usepackage{amsmath}
5  \usepackage{amsthm}
6  \usepackage{cmap}
7
8  \usepackage[cp1251]{inputenc}
9  \usepackage[russian]{babel}
10
11 \usepackage{graphicx}
12 \usepackage{python}
13
14 \begin{document}
15
16 \begin{python}
17 #
18
19 from sympy import *
20
21 x = Symbol('x')
22 y = sin(2*x)
23 dy_dx = diff(y,x)
24
25 print(r'Function')
26 print(r'$')
27 print(r'y(x)=')
28 print(latex(y))
29 print(r'$')
30
31 print(r'and derivative of the function')
32
33 print(r'\begin{equation}')
```

```
34 print(r'y(x) = ')
35 print(latex(y))
36 print(r', \quad')
37 print(r'\frac{dy}{dx} = ')
38 print(latex(dy_dx))
39 print(r'\end{equation}')
40
41 \end{python}
42
43 \end{document}
```

Фактически в окружении \begin(python)... \end(python) (строки 16–41) записан текст PDFLaTex при помощи функции print и некоторые команды Python.

Строки 21–23 содержат операторы языка Python и методы из модуля sympy, позволяющие производить символьное дифференцирование (подробнее см., например, sympy.org). Переменная y, dy_dx (строки 22, 23), то есть функция и результат ее дифференцирования, преобразуется в строки LaTeX при помощи функции latex (строки 28, 38).

После компилирования текста при помощи PDFLaTeX будет получен pdf файл, фрагмент которого показан на рис. 11

Function $y(x) = \sin(2x)$ and derivative of the function

$$y(x) = \sin(2x), \quad \frac{dy}{dx} = 2\cos(2x) \tag{1}$$

Рис. 11: Результат компиляции LaTeX файла

Помимо этого, в результате компиляции будут изготовлены два файла — файл name.py.out, в котором будут записан текст в формате LaTeX, сгенерированный в результате работы скрипта Python, файл name.py, в котором содержится скрипт Python.

Содержание файл `name.py.out`

```
Function
$
y(x)=
\sin{\left (2 x \right )}
$
and derivative of the function
\begin{equation}
y(x) =
\sin{\left (2 x \right )}
, \quad
\frac{dy}{dx} =
2 \cos{\left (2 x \right )}
\end{equation}
```

Содержание файл `name.py`

```
##

from sympy import *

x = Symbol('x')
y = sin(2*x)
dy_dx = diff(y,x)

print(r'Function')
print(r'$')
print(r'y(x)=')
print(latex(y))
print(r'$')

print(r'and derivative of the function')

print(r'\begin{equation}')
print(r'y(x) = ')
print(latex(y))
print(r', \quad')
print(r'\frac{dy}{dx} = ')
```

```
22 print(latex(dy_dx))
23 print(r'\end{equation}')
```

5.3 Рисование графиков в LaTeX при помощи Python

Рассмотрим пример рисования графиков в LaTeX при помощи Python.

```
1 %& -shell-escape
2 \documentclass{article}
3 \usepackage{graphicx}
4 \usepackage{python}
5 \begin{document}
6
7 \begin{figure}
8 \centering
9 \begin{python}
10 #
11 import matplotlib.pyplot as plt
12
13 import numpy as np
14 a = 0.0
15 b = 12.0 * np.pi
16 N = 100
17 x_points = np.linspace(a, b, num=N)
18
19 f=plt.figure()
20 plt.plot(x_points, np.sin(x_points)/x_points)
21 plt.grid(True)
22
23 plt.show()
24
25 f.savefig("function.pdf")
26 print(r'\includegraphics{function}')
27 \end{python}
28 \caption{$y(x)=\frac{\sin(x)}{x}$}
29 \end{figure}
30
```

```
31 \end{document}
```

Результат показан на рис. 12

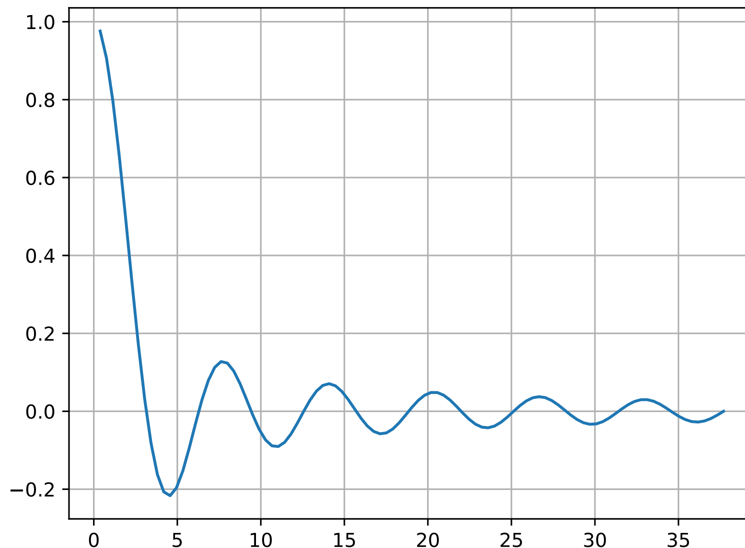

Figure 1: $y(x) = \frac{\sin(x)}{x}$

Рис. 12: Результат работы — фрагмент pdf-файла

5.4 Оформление скриптов при помощи LaTeX

В системе компьютерной верстки LaTeX имеется пакет `listing`, который позволяет оформлять скрипты Python, заключая их в рамки, делая нумерации строк, обеспечивая выделение цветом ключевых слов. Собственно говоря, данный текст был оформлен именно при помощи этого пакета. Подробное описание возможностей имеется в руководстве к пакету. Здесь ограничимся лишь указанием фрагмента преамбулы и одной из команд пакета, действие которой достаточно очевидно (строки 69–73).

```
1 \documentclass[a4paper,12pt]{article}
2
3 \usepackage[cp1251]{inputenc}
4 \usepackage[russian]{babel}
5
6 \usepackage[left=2.2cm, right=2.8cm, %
7             top=2cm, bottom=3.1cm,   %
```

```
8              includehead]{geometry}    %
9
10 \jot=3mm
11 \flushbottom
12
13 \usepackage{graphicx}
14 \usepackage{color}
15 \usepackage{float}
16
17
18 % Default fixed font does not support bold face
19 % for bold
20 \DeclareFixedFont{\ttb}{T1}{txtt}{bx}{n}{12}
21 % for normal
22 \DeclareFixedFont{\ttm}{T1}{txtt}{m}{n}{12}
23
24 % Custom colors
25 \usepackage{color}
26 \definecolor{deepblue}{rgb}{0,0,0.5}
27 \definecolor{deepred}{rgb}{0.6,0,0}
28 \definecolor{deepgreen}{rgb}{0,0.5,0}
29
30 \usepackage{listings}
31
32 % Python style for highlighting
33 \newcommand\pythonstyle{\lstset{
34 language=Python,
35 basicstyle=\ttm,
36 bold black keywords
37 commentstyle=\ttfamily\color{blue}, % white comments
38 moredelim=*[s][\ttfamily\color{deepgreen}]{\'}{\'},
39 moredelim=*[s][\ttfamily\color{blue}]{\'\'\'}{\'\'\'},
40 otherkeywords={self},             % Add keywords here
41 keywordstyle=\ttb\color{deepblue},
42 emph={MyClass,__init__},          % Highlighting
43 emphstyle=\ttb\color{deepred},    % Highlighting style
44 stringstyle=\color{deepgreen},    %
```

```
45 numbers=left,                        %
46 numberstyle=\small,                  %
47 stepnumber=1,                        %
48 numbersep=5pt,
49 xleftmargin=15mm,                    %
50 xrightmargin=5mm,                    %
51 frame=tb,                            % Any extra options
52 showstringspaces=false,              %
53 framexleftmargin=0mm,                %
54 framexrightmargin=-5mm,              %
55 frame=shadowbox,                     %
56 rulesepcolor=\color{blue}
57 }}
58
59 % Python for external files
60 \newcommand\pythonexternal[2][]{{
61 \pythonstyle
62 \lstinputlisting[#1]{#2}}}
63
64
65
66 \begin{document}
67 % ...
68
69 \pythonexternal[numbers=left,
70                 firstnumber=126,
71                 firstline=126,
72                 lastline=148]{Name.py}
73 \pythonexternal[numbers=left, firstnumber=1]{name.py}
74
75 % ...
76
77 \end{document}
```

Приложение
Установка дополнительных библиотек

Отсутствующие библиотеки можно инсталлировать, вызывая командные строки, например, для numpy

```
pip install numpy
```

Замечание. Предполагается, что на компьютере имеется лишь одна версия Python и установлены соответствующие переменные окружения (Envirenment). В противном случае необходимо указывать путь к расположению `pip`.

При работе в оболочке PyCharm инсталляция дополнительных модулей может быть сделана следующим способом:

```
File -> Settings -> Project: -> Project Interpreter
```

В окне программы будет приведен список всех установленных модулей. Для инсталляции недостающего модуля следует нажать значок «+» (в правом верхнем углу) и в строке поиска указать имя отсутствующего модуля. Далее следует выбрать модуль, указать номер версии `Specify version`, и нажать кнопку Install Package. В случае удачной (или неудачной) установки появится соответствующее сообщение.

Литература

1. Бахвалов Н. С. Численные методы. М.: Наука, 1973.
2. http://www.sympy.org/ru/index.html
3. http://matplotlib.org/
4. Долгих Т. Ф., Мелехов А. П., Полякова Н. М., Романов М. Н., Ширяева Е. В. Основы программирования. Python 3: электронное учебное пособие. Ростов-на-Дону. 2017. Электронный ресурс.

Список иллюстраций

1 Построение графиков функций 26
2 Геометрическая интерпретация интеграла и методов численного интегрирования 26
3 Метод средних прямоугольников 27
4 Методы левых и правых прямоугольников 28
5 Результат работы программы из примера 14 68
6 Результат работы программы из примера 18 71
7 Результат работы программы из примера 20 73
8 Результат работы программы из примера 21 74
9 Результат работы программы из примера 22 75
10 Результат работы программы из примера 25 79
11 Результат компиляции LaTeX файла 82
12 Результат работы — фрагмент pdf-файла 85