# Тема 7. Модуль math, модуль fractions, модуль cmath, модуль struct, модуль shelve (*.ini)

## I. Модуль math

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

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

Пример применения некоторых функций:

```
import math

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

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

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

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

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

```
# перевод из радиан в градусы
print(math.degrees(3.14159))  # 180

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

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

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

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

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

# II. Модуль ractions - рациональные числа (обыкновенные дроби)

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

## Создание обыкновенных дробей

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

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

```
>>> from fractions import Fraction
>>>
```

```
>>> Fraction()    #  по умолчанию numerator=0,
denominator=1
Fraction(0, 1)
>>>
>>> Fraction(numerator=1, denominator=2)    #
равносильно Fraction(1, 2)
Fraction(1, 2)
>>>
>>> Fraction(1, 2)
Fraction(1, 2)
```

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

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

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

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

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

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

А вот комплексные числа приведут к ошибке:

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

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

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

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

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

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

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

## Математические операции

## над рациональными числами

Все арифметические операторы поддерживают вычисления с рациональными числами:

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

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

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

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

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

## Fraction — атрибуты и методы

**x.numerator**

возвращает числитель дроби:

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

**x.denominator**

возвращает знаменатель дроби:

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

**Fraction.from_float(flt)**

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

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

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

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

**Fraction.from_decimal(dec)**

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

```
>>> from decimal import Decimal
>>>
>>> Fraction.from_decimal(Decimal('0.7'))
```

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

**Fraction.limit_denominator(max_denominator=1000000)**

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

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

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

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

**x.__floor__()**

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

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

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

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

**x.__ceil__()**

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

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

**x.__round__(ndigits=None)**

Округляет до ближайшего четного числа.

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

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

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

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

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

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

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

**`fractions.gcd(a, b)`**

возвращает наибольший общий делитель чисел a и b.

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

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

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

**Пример:**

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

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

>>> 473/60  = 7+53/60
```

# III. Модуль cmath

Модуль cmath – предоставляет функции для работы с комплексными числами.

**cmath.phase**(x) - возвращает фазу комплексного числа (её ещё называют аргументом). Эквивалентно math.atan2(x.imag, x.real). Результат лежит в промежутке [-π, π].

Получить модуль комплексного числа можно с помощью встроенной функции abs().

**cmath.polar**(x) - преобразование к полярным координатам. Возвращает пару (r, phi).

**cmath.rect**(r, phi) - преобразование из полярных координат.

**cmath.exp**(x) - $e^x$.

**cmath.log**(x[, base]) - логарифм x по основанию base. Если base не указан, возвращается натуральный логарифм.

**cmath.log10**(x) - десятичный логарифм.

**cmath.sqrt**(x) - квадратный корень из x.

**cmath.acos**(x) - арккосинус x.

**cmath.asin**(x) - арксинус x.

**cmath.atan**(x) - арктангенс x.

**cmath.cos**(x) - косинус x.

**cmath.sin**(x) - синус x.

**cmath.tan**(x) - тангенс x.

**cmath.acosh**(x) - гиперболический арккосинус x.

**cmath.asinh**(x) - гиперболический арксинус x.

**cmath.atanh**(x) - гиперболический арктангенс x.

**cmath.cosh**(x) - гиперболический косинус x.

**cmath.sinh**(x) - гиперболический синус x.

**cmath.tanh**(x) - гиперболический тангенс x.

**cmath.isfinite**(x) - True, если действительная и мнимая части конечны.

**cmath.isinf**(x) - True, если либо действительная, либо мнимая часть бесконечна.

**cmath.isnan**(x) - True, если либо действительная, либо мнимая часть NaN.

**cmath.pi** - π.

**cmath.e** - e.

# IV. Модуль struct -упаковка данных в бинарный файл

В некоторых приложениях приходится иметь дело с упакованными двоичными данными, которые создаются, например, программами на языке C. Для их сохранения и восстановления можно воспользоваться модулем struct из стандартной библиотеки Python.

Пример записи в файл и чтения из файла:

```
import struct
```

```
myfile = open("data.bin", "wb")
# упаковываем данные
bytes = struct.pack(b'>i4sh', 7, b'spam', 8)
myfile.write(bytes)
myfile.close()

myfile = open("data.bin", "rb")
data = myfile.read()
# извлекаем данные
values = struct.unpack(b'>i4sh', data)
print(values)
```

## Методы:

**struct.unpack(format, data)**

Распаковывает данные из структуры

Примеры:

```
# два целых 4х байтовые числа прямого порядка
struct.unpack('>LL', b'\x00\x00\x00\x9a\x00\x00\x00\x8d')
# (154, 141)

# два целых 4х байтовые числа прямого порядка
struct.unpack('>2L', b'\x00\x00\x00\x9a\x00\x00\x00\x8d')
# (154, 141)

# два целых 4х байтовые числа прямого порядка, пропустив
1 и 2 байта по краям
struct.unpack('>1x2L2x',
     b'\x00\x00\x00\x00\x9a\x00\x00\x00\x8d\x00\x00')
# (154, 141)
```

**struct.pack(format, data)**

Пакует данные в структуру

```
struct.pack('>L', 154)
# b'\x00\x00\x00\x9a'

struct.pack('>L', 141)
# b'\x00\x00\x00\x8d'
```

## Спецификаторы формата

**>** - прямой порядок
**<** - обратный порядок
**x** - пропустить один байт

**b -** знаковый один байт
**B -** беззнаковый байт
**h -** знаковое короткое целое число, 2 байта
**H -** беззнаковое короткое целое число, 2 байта
**i -** знаковое целое число, 4 байта
**I -** беззнаковое целое число, 4 байта
**l -** знаковое длинное целое число, 4 байта
**L -** беззнаковое длинное целое число, 4 байта
**Q -** беззнаковое очень длинное целое число, 8 байт
**f -** число с плавающей точкой, 4 байта
**d -** число с плавающей точкой двойной точности, 8 байт
**p -** счетчик и символы, 1 + count байт
**s -** символы, count символов

## Пример записи и чтения вещественных данных в бинаный файл

```
import sys
import struct
myfile = open("data.bin", "wb")
# упаковываем данные
for i in range(100):
    x1=10.0*float(i+0)
    x2=10.0*float(i+1)
    x3=10.0*float(i+2)
    x4=10.0*float(i+3)
    x5=10.0*float(i+4)
    print(x1,x2,x3,x4,x5)
    b=struct.pack(b'>fffff', x1,x2,x3,x4,x5)
    myfile.write(b)
myfile.close()
#sys.exit(0)
myfile = open("data.bin", "rb")
while (True):
    data = myfile.read(4*5)
    if data==b'':
        break
    # извлекаем данные
    values = struct.unpack(b'>fffff', data)
    y1=values[0]
    y2=values[1]
    y3=values[2]
    y4=values[3]
    y5=values[4]
    print(y1, y2, y3, y4, y5 )
```

```
myfile.close()
```

## Пример записи файла на FORTRAN и чтения на PYTHON вещественных данных в бинаный файл

Формирование файла:

```
! компилятор ming gfortran
  implicit none
  integer :: j, i
  real(4), DIMENSION(5) :: buf
  character(40) :: fname='d:\float5.bin'
  open(unit=50, file=trim(fname), status='UNKNOWN',
form='UNFORMATTED',err=100)
!
!  запись в файле содержит в "начале и "в конце" записи
!  "дискриптор" ЧЕТЫРЕ байта - количество байт
!  в записи не считая дискрипторов
!
  do i=1,50
     buf(1)=float(i)
     buf(2)=float(i+1)
     buf(3)=float(i+2)
     buf(4)=float(i+3)
     buf(5)=float(i+4)
     write(50) (buf(j), j=1,5)
     write(*, '( 5(g12.5,1x) )' ) (buf(j), j=1,5)
  enddo
  close(50)
  stop 0
100 write(*,*) '* error open file ', trim(fname), '*'
    stop 16
    end
```

Чтение файла:

```
import sys
import struct
myfile = open(r"d:\float5.bin", "rb")
##kbyteB=myfile.read(4) # дискриптор записи
nzap=0
while (True):
    kbyteB=myfile.read(4) # дискриптор записи -
                          # кол-во байт в этой записи
    if kbyteB==b'':
        break
```

```
    nzap+=1
    kbyte = struct.unpack(b'<L', kbyteB)[0]
    print('запись № ',nzap,' содержит ',kbyte,' байт')
    data = myfile.read(4*5)  #чтение записи
    if data==b'':
        break
    # извлекаем данные
    values = struct.unpack(b'<fffff', data)
    y1=values[0]
    y2=values[1]
    y3=values[2]
    y4=values[3]
    y5=values[4]
    print(y1, y2, y3, y4, y5 ) #values)
    kbyteB=myfile.read(4) # дискриптор конца записи -
                          # кол-во байт в этой записи
myfile.close()
```

# V. Файлы CSV

Программисты часто сталкиваются с задачей обработки больших объемов структурированных данных.Python имеет встроенную библиотеку CSV, с помощью которой программист может работать со специальными CSV файлами. Это своего рода электронные таблицы.

Каждая строка в файле csv представляет отдельную запись или строку, которая состоит из отдельных столбцов, разделенных запятыми. Собственно поэтому формат и называется Comma Separated Values. Но хотя формат csv - это формат текстовых файлов, Python для упрощения работы с ним предоставляет специальный встроенный модуль csv

**Что такое файлы CSV**

Файл CSV – это особый вид файла, который позволяет структурировать большие объемы данных.

По сути, он является обычным текстовым файлом, однако каждый новый элемент отделен от предыдущего запятой или другим разделителем. Обычно каждая запись начинается с новой строки. Данные CSV можно легко экспортировать в электронные таблицы или базы данных. Программист может расширять CSV файл, добавляя новые строки.

**Пример CSV файла, где в качестве разделителя используется запятая:**

```
Имя,Профессия,Год рождения
Виктор,Токарь,1995
Сергей,Сварщик,1983
```

Как видно из примера, в первой строке обычно указывается, какая информация будет находиться в каждом столбце. Кроме того, после последнего элемента строки запятая не ставиться, интерпретатор определяет конец строки по символу переноса.

Вместо запятой можно использовать любой другой разделитель, поэтому при чтении CSV файла нужно заранее знать, какой символ используется.

Важно помнить, что CSV – это обычный текстовый файл, который не поддерживает символы в кодировках, отличающихся от ASCII или Unicode.

**Библиотека CSV**

Эта основная библиотека для работы с CSV файлами в Python.

Библиотека csv является встроенной, поэтому её не нужно скачивать, достаточно использовать обычный импорт:

```
import csv
```

**Чтение из файлов (парсинг)**

Для того чтобы прочитать данные из файла, программист должен создать объект **reader**:

```
reader_object = csv.reader(file, delimiter = ",")
```

**reader** имеет метод **__next__(),** то есть является итерируемым объектом, поэтому чтение из файла происходит следующим образом:

```
import csv
with open("classmates.csv", encoding='utf-8') as r_file:
    # Создаем объект reader, указываем символ-разделитель ","
    file_reader = csv.reader(r_file, delimiter = ",")
    # Счетчик для подсчета количества строк и вывода заголовков
столбцов
    count = 0
    # Считывание данных из CSV файла
    for row in file_reader:
        if count == 0:
            # Вывод строки, содержащей заголовки для столбцов
            print(f'Файл содержит столбцы: {", ".join(row)}')
        else:
            # Вывод строк
            print(f'{row[0]} - {row[1]} и он родился в {row[2]}
                                                    году.')
        count += 1
    print(f'Всего в файле {count} строк.')
```

Предположим, что у нас есть CSV файл, который содержит следующую информацию:

```
Имя,Успеваемость,Год рождения
Саша,отличник,200
Маша,хорошистка,1999
Петя,троечник,2000
```

Тогда, если открыть этот файл в нашей программе, то будут получены следующие результаты:

```
Файл содержит столбцы: Имя, Успеваемость, Год рождения
    Саша - отличник и он родился в 200 году.
    Маша - хорошистка и он родился в 1999 году.
    Петя - троечник и он родился в 2000 году.
Всего в файле 4 строк.
```

Использование конструкции **with…as** позволяет программисту быть уверенным, что файл будет закрыт, даже если при выполнении кода произойдет какая-то ошибка.

Обратите внимание, что при открытии нужно указать правильную кодировку, в которой сохранены данные. В данном случае **encoding=’utf-8′.** Если не указывать, то будет использоваться кодировка по умолчанию. Для Windows это **cp1251**.

Библиотека CSV позволяет работать с файлами, как со словарями, для этого нужно создать объект **DictReader**. Обращаться к элементам можно по имени столбцов, а не с помощью индексов. Для того, чтобы исходная программа делала аналогичный вывод, её следует изменить следующим образом:

```
import csv
with open("classmates.csv", encoding='utf-8') as r_file:
    #Создаем объект DictReader, указываем символ-разделитель
","
    file_reader = csv.DictReader(r_file, delimiter = ",")
    #  Счетчик для подсчета количества строк и вывода
                                    заголовков столбцов
    count = 0
    # Считывание данных из CSV файла
    for row in file_reader:
        if count == 0:
            # Вывод строки, содержащей заголовки для столбцов
            print(f'Файл содержит столбцы: {", ".join(row)}')
        # Вывод строк
        print(f' {row["Имя"]} - {row["Успеваемость"]}', end='')
        print(f' и он родился в {row["Год рождения"]} году.')
        count += 1
    print(f'Всего в файле {count + 1} строк.')
```

Обращаться к элементам по названию столбца более удобно, кроме того, это упрощает понимание кода.

Обратите внимание, что в цикл for при первой итерации будет записан в row не шапка таблицы, а первая её строка. Поэтому при выводе количества строк переменную count увеличили на 1.

**Дополнительные параметры объекта DictReader**

**DictReader** имеет параметры:

- `dialect` — Набор параметров для форматирования информации. Подробнее про них ниже.
- `line_num` — Устанавливает количество строк, которое может быть прочитано.
- `fieldnames` — Определяет заголовки для столбцов, если не определить атрибут, то в него запишутся элементы из первой прочитанной строки файла. Заголовки нужны для того, чтобы легко было понять, какая информация содержится или должна содержаться в столбце.

Например, если бы в classmates.csv не было бы первой строки с заголовками, то можно было бы его открыть следующим образом:

```
fieldnames = ['Имя', 'Успеваемость', 'Год рождения']
file_reader = csv.DictReader(r_file, fieldnames = fieldnames)
```

Также можно использовать метод __next__() для получения следующей строки. Этот метод делает объект reader итерируемым. То есть он вызывается при каждой итерации и возвращает следующую строку. Этот метод и используется при каждой итерации в цикле for для получения очередной строки.

**Запись в файлы**

Для записи информации в CSV файл необходимо создать объект writer:

```
file_writer = csv.writer(w_file, delimiter = "\t")
```

Для записи в файл данных используется метод writerow(), который имеет следующий синтаксис:

```
writerow(["Имя", "Фамилия", "Отчество"])
```

Код программы для записи в CSV файл выглядит так:

```
import csv
with open("classmates.csv",mode="w",encoding='utf-8') as w_file:
        file_writer = csv.writer(w_file, delimiter = ",",
                                 lineterminator="\r")
        file_writer.writerow(["Имя", "Класс", "Возраст"])
        file_writer.writerow(["Женя", "3", "10"])
        file_writer.writerow(["Саша", "5", "12"])
        file_writer.writerow(["Маша", "11", "18"])
```

Обратите внимание, что при записи использовался, `lineterminator="\r"`. Это разделитель между строками таблицы, по умолчанию он "\r\n".

После выполнения программы в файле CSV будет следующий текст:

```
Имя,Класс,Возраст
Женя,3,10
Саша,5,12
Маша,11,18
```

В качестве параметра метод writerow() принимает список, элементы которого будут записаны в строку через символ-разделитель.

**Запись в файл также может быть осуществлена с помощью объекта DictWriter.**

Важно помнить, что он требует явного указания параметра **fieldnames**. В качестве аргумента метода **writerow** используется словарь.

Код программы выглядит так:

```
import csv
with    open("classmates.csv",    mode="w",    encoding='utf-8')
                                                as w_file:
    names = ["Имя", "Возраст"]
    file_writer = csv.DictWriter(w_file, delimiter = ",",
                                 lineterminator="\r",
                                     fieldnames=names)
    file_writer.writeheader()
    file_writer.writerow({"Имя": "Саша", "Возраст": "6"})
    file_writer.writerow({"Имя": "Маша", "Возраст": "15"})
    file_writer.writerow({"Имя": "Вова", "Возраст": "14"})
```

Вывод в файл будет следующим:

```
Имя,Возраст
Саша,6
Маша,15
Вова,14
```

**Дополнительные параметры DictWriter**

Объект writer также имеет атрибут **dialect**, который определяет, как будут форматироваться данные при записи в файл, про него будет описано ниже.

Кроме того, **writer имеет методы:**

- **writerows(rows)** — Записывает все элементы строк.
- **writeheader()** —Выводит заголовки для столбцов. Заголовки должны быть переданы объекту **writer** в виде списка, как атрибут **fieldnames**. **writeheader** был использован в предыдущем примере.

Рассмотрим применение writerows:

```
file_writer.writerows([{"Имя": "Саша", "Возраст": "6"},
                       {"Имя": "Маша", "Возраст": "15"},
                      {"Имя": "Вова", "Возраст": "14"}])
```

**Диалекты**

Чтобы каждый раз не указывать формат входных и выходных данных, определенные параметры форматирования сгруппированы в диалекты (**dialect**).

При создании объекта **reader** или **writer** программист может указать нужный ему диалект, кроме того, некоторые параметры диалекта можно переопределить вручную, также указав их при создании объекта.

Для создания диалекта используется команда:

```
register_dialect("имя", delimiter = "\t", ...)
```

Класс **Dialect** позволяет определить следующие **атрибуты форматирования**:

| Атрибут | Значение |
|---|---|
| `delimiter` | Устанавливает символ, с помощью которого разделяются элементы в файле. По умолчанию используется запятая. |
| `doublequote` | Если True, то символ quotechar удваивается, если False, то к символу qutechar добавляется ecsapechar в качестве префикса. |
| `escapechar` | Строка из одного символа, которая используется для экранирования символа-разделителя. |
| `lineterminator` | Определяет разделитель для строк, по умолчанию используется «\r\n» |
| `quotechar` | Определяет символ, который используется для окружения символа-разделителя. По умолчанию используются двойные кавычки, то есть quotechar = ‘ “ ‘. |
| `quoting` | Определяет символ, который используется для экранирования символа разделителя (если не используются кавычки). |
| `skipinitialspace` | Если установить значение этого параметра в True, то все пробелы после символа-разделителя будут игнорироваться. |
| `strict` | Если установить в True, то при неправильном вводе CSV будет возбуждаться исключение Error. |

**Пример использования:**

```
import csv
csv.register_dialect('my_dialect', delimiter=':',
                                   lineterminator="\r")
with open("classmates.csv", mode="w", encoding='utf-
8')
                                          as w_file:
    file_writer = csv.writer(w_file, 'my_dialect')
    file_writer.writerow(["Имя", "Класс", "Возраст"])
    file_writer.writerow(["Женя", "3", "10"])
    file_writer.writerow(["Саша", "5", "12"])
    file_writer.writerow(["Маша", "11", "18"])
```

В результате получим:

```
Имя:Класс:Возраст
```

```
Женя:3:10
Саша:5:12
Маша:11:18
```

**Пример**

```
import csv
FILENAME = "users.csv"
users = [
    [1, "математика", 75 ],
    [2, "программировани", 85],
    [3, "мат анализ", 34]
]

with open(FILENAME, "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(users)

with open(FILENAME, "a", newline="") as file:
    user = [4, "веб прог", 80]
    writer = csv.writer(file)
    writer.writerow(user)

d={}
with open(FILENAME, "r", newline="") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row[0], " - ", row[1], " - ", row[2])
        d[row[1]]=int(row[2])
print("d=",d)

#сформируем список D1 из словаря и d
# и его сортируем по ключу словаря key=lambda x:x[0]
# если хотим по значению, то пишите key=lambda x:x[1]
# если хотим в обратном порядке, то пишите key=lambda x:-x[1]
D1=sorted(d.items(),key=lambda x:-x[1]) # по ключам

print("D1=",D1)   # печать списка

#  отсортированный список в словарь D2 и его печать
D2={D1[i][0] : D1[i][1] for i in range(len(D1)) }
#печать таблицей D2
for kk, vv in D2.items():
    print(vv,'\t',kk)
```

```
    ll=list(D2.items())
    kkl=len(ll)-1
    print("maximum score of", ll[0][1], "in the subject", ll[0][0])
    print("minimum score of", ll[kkl][1], "in the subject",
ll[kkl][0])

    user1=[]
    nn=0
    for kk, vv in D2.items():
        user1.append([nn,kk,vv])
        nn+=1

    namerow=["№пп", "предмет", "оценка"]
    with open(FILENAME, "w", newline="") as file: #encoding='utf-8'
        writer=csv.writer(file, delimiter=",", lineterminator="\n")
        writer.writerow(namerow)
        writer.writerows(user1)
```

# VI. Модуль shelve (*.ini)

Для работы с бинарными файлами в Python может применяться еще один модуль - `shelve`. Он сохраняет объекты в файл с определенным ключом. Затем по этому ключу может извлечь ранее сохраненный объект из файла. Процесс работы с данными через модуль `shelve` напоминает работу со словарями, которые также используют ключи для сохранения и извлечения объектов.

Для открытия файла модуль shelve использует функцию open():

```
open(путь_к_файлу[, flag="c"[, protocol=None[,
                                    writeback=False]]])
```

Где параметр `flag` может принимать значения:

- **c**: файл открывается для чтения и записи (значение по умолчанию). Если файл не существует, то он создается.
- **r**: файл открывается только для чтения.
- **w**: файл открывается для записи.
- **n**: файл открывается для записи Если файл не существует, то он создается. Если он существует, то он перезаписывается

Для закрытия подключения к файлу вызывается метод `close()`:

```
import shelve
d = shelve.open(filename)
d.close()
```

Либо можно открывать файл с помощью оператора **`with`**.

Сохраним и считаем в файл несколько объектов:

```
import shelve

FILENAME = "states2"
with shelve.open(FILENAME) as states:
    states["London"] = "Great Britain"
    states["Paris"] = "France"
    states["Berlin"] = "Germany"
    states["Madrid"] = "Spain"

with shelve.open(FILENAME) as states:
    print(states["London"])
    print(states["Madrid"])
```

Запись данных предполагает установку значения для определенного ключа:

```
states["London"] = "Great Britain"
```

А чтение из файла эквивалентно получению значения по ключу:

```
print(states["London"])
```

В качестве ключей используются строковые значения.

При чтении данных, если запрашиваемый ключ отсутствует, то генерируется исключение. В этом случае перед получением мы можем проверять на наличие ключа с помощью оператора **in**:

```
with shelve.open(FILENAME) as states:
    key = "Brussels"
    if key in states:
        print(states[key])
```

Также можно использовать метод `get()`. Первый параметр метода - ключ, по которому следует получить значение, а второй - значение по умолчанию, которое возвращается, если ключ не найден.

```
with shelve.open(FILENAME) as states:
    state = states.get("Brussels", "Undefined")
    print(state)
```

Используя цикл **for**, можно перебрать все значения из файла:

```
with shelve.open(FILENAME) as states:
    for key in states:
        print(key," - ", states[key])
```

Метод `keys()` возвращает все ключи из файла, а метод `values()` - все значения:

```
with shelve.open(FILENAME) as states:

    for city in states.keys():
        print(city, end=" ")        # London Paris Berlin
Madrid
```

```
    print()
    for country in states.values():
        print(country, end=" ")      # Great Britain
                                     # France Germany Spain
```

Еще один метод `items()` возвращает набор кортежей. Каждый кортеж содержит ключ и значение.

```
with shelve.open(FILENAME) as states:

    for state in states.items():
        print(state)
```

Консольный вывод:

```
("London", "Great Britain")
("Paris", "France")
("Berlin", "Germany")
("Madrid", "Spain")
```

### Обновление данных

Для изменения данных достаточно присвоить по ключу новое значение, а для добавления данных - определить новый ключ:

```
import shelve

FILENAME = "states2"
with shelve.open(FILENAME) as states:
    states["London"] = "Great Britain"
    states["Paris"] = "France"
    states["Berlin"] = "Germany"
    states["Madrid"] = "Spain"

with shelve.open(FILENAME) as states:

    states["London"] = "United Kingdom"
    states["Brussels"] = "Belgium"
    for key in states:
        print(key, " - ", states[key])
```

### Удаление данных

Для удаления с одновременным получением можно использовать функцию `pop()`, в которую передается ключ элемента и значение по умолчанию, если ключ не найден:

```
with shelve.open(FILENAME) as states:

    state = states.pop("London", "NotFound")
    print(state)
```

Также для удаления может применяться оператор `del`:

```
with shelve.open(FILENAME) as states:
```

```
    del states["Madrid"]     # удаляем объект
                              #   с ключом Madrid
```

Для удаления всех элементов можно использовать метод `clear()`:

```
with shelve.open(FILENAME) as states:

    states.clear()
```