

## Основные типы данных и структуры данных в Maple

1. Основные типы данных в Maple.
2. Структуры данных в Maple: *expression sequence*, *set*, *list*, *array*, *string*.
3. Использование структур данных и специальные команды для работы с ними.

### § 1. Основные типы данных в Maple

В Maple существует около 200 типов данных, в том числе:

- типы математических операций:
  - ``+`` (сложение)
  - ``*`` (умножение)
  - ``^`` (возведение в степень)
- типы сравнения:
  - ``<`` (меньше)
  - ``<=`` (меньше или равно)
  - ``<>`` (не равно)
- типы логических операций
- числовые типы, например:
  - **integer** – целое число
  - **float** – число с плавающей точкой
  - **fraction** – рациональная дробь
  - и др.
- тип символ (**symbol**)

Для определения типа объекта используется команда **whattype**, в результате ее выполнения на экран выводится тип заданного объекта. Проверить объект на соответствие определенному типу можно с помощью команды **type**, результатом выполнения которой является одна из логических констант: **true** (истина), **false** (ложь) или **FAIL** (не определено).

**whattype(x)** – команда выдает тип объекта *x*  
**type(x, x\_type)** – команда проверяет объект *x* на соответствие типу с именем *x\_type*

Рассмотрим примеры и результаты применения этих команд.

```
> whattype(a+b);whattype(a-b);
      +
      +
> whattype(2>3);
      <
> whattype(1/2);
      fraction
> whattype(2);
      integer
> whattype(2.);
      float
> whattype(a);whattype(`a+b`);
      symbol
```

```

symbol
> type(2,integer);type(2,float);
true
false
> type(a+b,`+`);type(`a+b`,`+`);
true
false

```

Важно, что тип переменной может меняться в зависимости от присвоенного ей значения:

```

> a:=3; whattype(a);
a := 3
integer
> a:=sqrt(2);whattype(a);
a :=  $\sqrt{2}$ 
`^`

```

## § 2. Структуры данных в Maple

Выражения Maple могут быть составлены как из простых объектов, так и из сложных объектов и других выражений Maple. Среди сложных объектов выделяют *структуры данных*.

Maple различает следующие структуры данных:

- Последовательность выражений (Expression sequence)
- Множество, или набор (Set)
- Список (List)
- Строка (String)
- Массив (Array)

- Таблица (Table)
- Матрица, вектор (Matrix, Vector)

### § 2.1. Структуры данных в Maple: последовательность выражений (expression sequence)

**Последовательность выражений (Expression sequence)** – это группа выражений Maple, отделенных друг от друга запятыми.

#### Последовательность выражений

$^{\circ}, ^{\circ}, ^{\circ}, ^{\circ}$

Среди выражений в последовательности могут быть объекты разных типов данных. Тип результирующего объекта «последовательность выражений» называется **exprseq**.

Рассмотрим примеры.

```

> restart;
> 2,3,4;whattype(%);
2, 3, 4
exprseq
> s:=2,x,a+b,`a+b`,sin(x^2),x;
s := 2, x, a + b, a + b, sin(x2), x

```

```
> whattype(s);
```

*exprseq*

Доступ к одному из элементов последовательности осуществляется по номеру его позиции в структуре. Нумерация элементов осуществляется с начала или с конца:

```
1 2 3 4 5 6
s := 2, x, a + b, a + b, sin(x^2), x
-6 -5 -4 -3 -2 -1
```

```
> s[3];whattype(%);
```

*a + b*  
*`+`*

```
> s[-3];whattype(%);
```

*a + b*  
*symbol*

Проверим, равны ли второй и шестой элементы, а также третий и четвертый элементы. Для этого используем функцию вычисления логических выражений **evalb**.

```
> s[2];s[6];evalb(s[2]=s[6]);
```

*x*  
*x*  
*true*

Ответ – истина, то есть, эти элементы равны.

```
> s[3];s[4];evalb(s[3]=s[4]);
```

*a + b*  
*a + b*  
*false*

Ответ – ложь, то есть, эти элементы различны.

Рассмотрим пример доступа к нескольким элементам последовательности, например, со второго по четвертый:

```
> s[2..4];
```

*x, a + b, a + b*

Полученный объект – последовательность выражений:

```
> whattype(%);
```

*exprseq*

Для добавления элементов в последовательность новые элементы нужно дописать в конец последовательности через запятую:

```
> t:=s[2..4],8;
```

*t := x, 5 + b, a + b, 8*

Аналогично осуществляется приращение последовательности:

```
> t:=t,s[1],10;
```

*t := x, 5 + b, a + b, 8, 2, 10*

Можно задать пустую последовательность, которая не содержит ни одного элемента. Для этого используется зарезервированное имя **NULL**.

```
> s:=NULL;
```

*s :=*

```
> s:=s,7,b;
```

*s := 7, b*

Последовательность с определенной закономерностью изменения элементов можно создать с помощью оператора формирования последовательности (знак \$):

```
> $1..5;
```

1, 2, 3, 4, 5

```
> a[i] $ i = 1..3;
```

$a_1, a_2, a_3$

## § 2.2. Структуры данных в Maple: множество (set)

**Множество, или набор (Set)** – это группа выражений Maple, записанных в *фигурных* скобках через запятую.

**Множество**

{°, °, °, °}

Данный объект имеет все черты математического множества:

- 1) каждый элемент хранится в единственном экземпляре, т. е. повторяющиеся элементы хранятся один раз
- 2) заданный порядок элементов не хранится

Среди выражений во множестве могут быть объекты разных типов данных. Тип результирующего объекта «множества» называется **set**.

Рассмотрим пример.

```
> restart;
```

```
> m:={2,x,x,b,a,a};
```

$m := \{2, x, a, b\}$

```
> whattype(m);
```

set

Доступ к одному из элементов множества осуществляется по номеру его позиции в структуре хранящихся элементов. Синтаксис аналогичен синтаксису обращения к элементу последовательности.

```
> m[3];
```

a

```
> m[5];
```

Error, invalid subscript selector

Ошибка, так как хранится только четыре элемента.

Можно задать пустое множество, которое не содержит ни одного элемента:

```
> p:={};
```

$p := \{ \}$

Над объектами Maple, которые являются множествами, можно совершать обычные операции алгебры множеств:

- **объединение** (оператор **union**, символ из шаблонов Common Symbols: **U**);

```
> restart;
```

```
> S1:={a,b,c};
```

$S1 := \{a, b, c\}$

```
> S2:={b,c,d,e,d};
```

$S2 := \{b, c, d, e\}$

```
> S3:={a,f};
```

$S3 := \{a, f\}$

```
> SU:=S1 union S2;
```

- $SU := \{a, b, c, d, e\}$
- >  $SU := S1 \cup S2;$
- $SU := \{a, b, c, d, e\}$
- > **S1 union S2 union S3;**
- $\{a, b, c, d, e, f\}$
- > **S1 union {};**
- $\{a, b, c\}$
- **пересечение** (оператор **intersect**, символ из шаблонов Common Symbols:  $\cap$ );
- > **S1:={a,b,c};**
- $S1 := \{a, b, c\}$
- > **S2:={b,c,d,e,d};**
- $S2 := \{b, c, d, e\}$
- > **S3:={a,f};**
- $S3 := \{a, f\}$
- > **SI:=S1 intersect S2;**
- $SI := \{b, c\}$
- >  $SI := S1 \cap S2;$
- $SI := \{b, c\}$
- > **S1 intersect S2 intersect S3;**
- $\{\}$
- > **S1 intersect {};**
- $\{\}$
- **разность** (оператор **minus**, символ из шаблонов Common Symbols:  $\setminus$ );
- > **S1:={a,b,c};**
- $S1 := \{a, b, c\}$
- > **S2:={b,c,d,e,d};**
- $S2 := \{b, c, d, e\}$
- > **Sm1:=S1 minus S2;**
- $Sm1 := \{a\}$
- >  $Sm1 := S1 \setminus S2;$
- $Sm1 := \{a\}$
- > **Sm2:=S2 minus S1;**
- $Sm2 := \{d, e\}$
- >  $Sm2 := S2 \setminus S1;$
- $Sm2 := \{d, e\}$
- > **S1 minus {};**
- $\{a, b, c\}$
- > **{ } minus S1;**
- $\{\}$

Добавление элементов во множество осуществляется с помощью операции объединения. Продолжим предыдущий пример:

- >  $S := S1 \cup \{1, x\}$
- $S := \{1, a, b, c, x\}$

Аналогично осуществляется приращение множества:

>  $S := S \cup \{t\}$

$S := \{1, a, b, c, t, x\}$

### § 2.3. Структуры данных в Maple: список (list)

**Список (List)** – это группа выражений Maple, записанных в *квадратных* скобках через запятую.

**Список**  
[°, °, °, °]

Данный объект черты, противоположные множеству:

- 1) хранятся все повторяющиеся элементы
- 2) хранится заданный порядок элементов

Среди выражений в списке могут быть объекты разных типов данных. Тип результирующего объекта «список» называется **list**.

Рассмотрим пример.

> **L := [2, x, x, b, a, a];**

$L := [2, x, x, b, a, a]$

> **whattype(L);**

*list*

Доступ к одному из элементов списка осуществляется по номеру его позиции в структуре хранящихся элементов. Синтаксис аналогичен синтаксису обращения к элементу последовательности.

> **L[3];**

*x*

> **L[5];**

*a*

Аналогично осуществляется доступ к нескольким элементам списка:

> **L[3..5];**

$[x, b, a]$

Можно задать пустой список, который не содержит ни одного элемента:

> **Lp := [];**

$Lp := []$

Для добавления новых элементов и приращения списка используется команда **op** (синтаксис команды подробно описан далее).

> **M := [op(L), c];**

$M := [2, x, x, b, a, a, c]$

> **M := [op(M), sin(x)];**

$M := [2, x, x, b, a, a, c, \sin(x)]$

### § 2.4. Структуры данных в Maple: массив (array)

**Массив (Array)** – это обобщение списка на любую размерность (2, 3 и т.д.). Обычный список в сущности является одномерным массивом. Для индексов массива можно использовать любые целые числа, в том числе отрицательные и ноль. Для задания массива используется команда **array** (можно также использовать новую команду **Array**).

**Массив**

**array(indexfunc, dims, init)** – команда создает массив, элементы которого вычисляются задает (необязательная) индексирующая функция **indexfunc** (задает

структуру матрицы массива: симметричная, диагональная и т. д.), переменная **dims** – последовательность диапазонов изменения индексов, **init** – список начальных значений массива.

Чтобы задать массив, для каждого измерения нужно задать диапазон изменения индексов, а во вложенных списках указать значения элементов.

Рассмотрим одномерный массив длины 4. Его можно интерпретировать как массив координат 4-х точек на плоскости:

```
> a:=array(1..4, [2, -3, 4, 5]);  
a := [ 2 -3 4 5 ]
```

Рассмотрим двумерный массив размером 1..2×1..3. Его можно интерпретировать как массив координат двух точек в пространстве (первый индекс массива указывает на номер точки, второй – на номер координаты):

```
> b:=array(1..2, 1..3, [[1, 2, -1], [-2, 3, 1]]);  
b := [ 1 2 -1  
      -2 3 1 ]
```

Рассмотрим пример трехмерного массива:

```
> c:=array(1..2, 1..2, 1..2, [[1, 2], [3, 4]], [[5, 6], [7, 8]]);  
c := ARRAY([1..2, 1..2, 1..2], [(1, 1, 1) = 1, (1, 1, 2) = 2, (1, 2, 1)  
= 3, (1, 2, 2) = 4, (2, 1, 1) = 5, (2, 1, 2) = 6, (2, 2, 1) = 7, (2, 2, 2)  
= 8])
```

Доступ к одному из элементов массива осуществляется по номеру его позиции в структуре массива: для одномерного массива – по одному индексу, для двумерного массива – по двум индексам, для трехмерного массива – по трем индексам.

```
> b[1, 2]; b[2, 1];  
2  
-2  
  
> c[1, 2, 2];  
4
```

## § 2.5. Структуры данных в Maple: строка (string)

Строка (String) – это любой набор символов, заключенный в двойные кавычки. Длина строки в Maple практически не ограничена.

Строка " "
---------------

Тип объекта «строка» называется **string**.

Пример:

```
> S:="Это строка!";  
S := "Это строка!"  
  
> whattype(S);  
string
```

Доступ к одному или нескольким элементам строки осуществляется по номеру позиции элементов в структуре строки:

```
S := "Это строка!"  
1 2 3 4 5 6 7 8 9 10 11
```

```

> s[5];
                                     "c"
> s[9..11];
                                     "ка!"
> s[14];
                                     ""

```

### § 3. Использование структур данных и специальные команды для работы с ними

Структуры данных «множество» и «список» широко используются как аргументы различных команд. Некоторые команды допускают использования любой из этих двух структур, если порядок следования элементов не важен (можно использовать как фигурные скобки, так и квадратные). Если должен учитываться порядок элементов в структуре, то используется «список» (т.е. квадратные скобки).

Важно: сами аргументы команды задаются в круглых скобках, кроме того, круглые используются для определения порядка действий. Требования к аргументам команды и их типам всегда можно прочитать в справочной информации по данной команде.

#### Общие команды для списка и множества: команда подсчета элементов **nops** и команда извлечения элементов **op**

Для подсчета количества элементов в списке или множестве используется команда **nops**. Для извлечения определенного числа элементов из списка или множества можно использовать команду **op**.

**nops (x)** – команда выдает количество элементов в списке или множестве **x**  
**op (i, e)** – команда извлекает элемент, находящийся на позиции **i** в списке или множестве **e**  
**op (i..j, e)** – команда извлекает элементы, находящиеся на позициях с **i** по **j** в списке или множестве **e**, в качестве результата возвращается последовательность элементов  
**op (e)** – команда извлекает все элементы списка или множества **e**, в качестве результата возвращается последовательность элементов

Рассмотрим примеры и результаты применения этих команд.

```

> s:={a,b,4,-1,f,c,f};nops(s);
                                     s:={-1,4,a,f,b,c}
                                     6
> op(s);
                                     -1,4,c,a,b,f
> op(3,s);op(1..3,s);
                                     a
                                     -1,4,a
> s:=[a,b,4,-1,f,c,f];nops(s);
                                     s:=[a,b,4,-1,f,c,f]
                                     7
> op(4,s);op(1..4,s);
                                     -1
                                     a,b,4,-1

```



## Команда сортировки списка `sort`

`sort(L, f)` – команда выдает сортирует список `L`, необязательный аргумент `f` задает порядок сортировки

```
> s:=[2,1,3]:
> sort(s);
                                [1, 2, 3]
> sort(s, `>`);#сортировка по убыванию
                                [3, 2, 1]
> s:=[a,ba,aaa,aa]:
> sort(s,length);#сортировка по длине
                                [a, ba, aa, aaa]
> s:=[я,м,а,в,б]:
> sort(s);#для символов – по умолчанию сортировка по алфавиту
                                [а, б, в, м, я]
```

## Некоторые команды пакета `StringTools` для работы со строками

`Length(s)` – команда выдает длину строки `s`

`Split(s)` – команда выдает список строк, составляющих отдельные слова в строке `s`. При этом по умолчанию для определения слов команда анализирует расположение пробелов в строке

`Stem(s)` – команда выдает строку, содержащую основу слова `s` (работает только для английских слов)

```
> s:="Impressive string":
> with(StringTools):
> Length(s);
```

17

*Пример.* Вывести основу четвертого слова предложения “Programming is an extremely useful skill”.

```
> with(StringTools):
> s := "Programming is an extremely useful skill"
                                s := "Programming is an extremely useful skill"
> t := Split(s)
                                t := ["Programming", "is", "an", "extremely", "useful", "skill"]
> Stem(t[4])
                                "extrem"
```