6.3 Table calculation

6.3.1 Introduction

Table calculation is used to perform operations on attribute data in attribute tables. The operators and functions which can be used in Table Calculations are identical to those in Map Calculation. The difference is that the outcome of a calculation is not stored in a map but in a table, generally in a newly created column. Table calculations often take much less time compared to map calculations. Therefore it is recommended to perform calculations on geographic data in tables. Afterwards, results can always be displayed in an attribute map.

The following list gives an overview of the functionality:

- column and record calculations using arithmetic, relational, logical, conditional, exponential, logarithmic, and other operators and functions
- classification of columns
- aggregation of values
- calculations using columns of another table on disk (join)
- calculations using predefined variables and record specific operations
- calculations using map values, coordinates and colors
- calculations on point data
- application of user-defined functions.

For an overview of operators and functions, please refer to the first pages of this chapter.

The result of calculation can be written either:

- into the same table, in a new column, or
- in a new table, in a new column (only possible through the menu), or
- in another existing table, in a new column.

Calculations are made by typing your calculation formula on the command line of a Table window. The TabCalc command line may be hidden or shown with menu command Options, Command line.

6.3.2 General syntax of TabCalc formulae

A TabCalc formula or statement consists of an output column name that contains the result of the calculation, a definition symbol (=) or an assignment symbol (:=), and an expression.

```
Outputcolumn = Expression
```

or

```
Outputcolumn := Expression
```

Example
A simple TabCalc formula reads:

```
Column3 = Column1 + Column2
```

In the table, this results in:

Column1	Column2	Column3
1	10	11
2	20	22
3	30	33
4	40	44

Each record in Column3 (new or existing) stores the sum of the values of the records in existing Column1 and Column2.

Output column name
All column names used on the command line should start with a character between A to Z. There is no limitation on length of column names. This is different from other object names which have a limit of eight characters.
If necessary, the output object's domain or its domain and value range may be specified in a pair of curly brackets after the output object name.
To create for instance a column OUT with domain value 'MyVal', this would look like:

```
OUT {dom=MyVal} = expression
```

To create a column OUT2 with domain MYVAL2 where the column values should range between minimum value 0 and maximum value 10000, with a precision of 1, this would look like:

```
OUT2{dom=MyVal2;vr=0:10000:1} = expression
```

When you double-click the column name in the table window the Column Properties box appears on the screen. In this box you may edit several column properties. When necessary you can update your column values, ID’s or classes or you can break the dependency link.

Definition symbol = and assignment symbol :=
By using the definition symbol = on the command line, a dependent column is created: the definition of how the output column was created is stored. For more information, please refer to Basic concepts : dependent data objects.

By using the assignment symbol **:=** on the command line, an editable column is created: the dependency link is broken and the output values are directly assigned to the output fields in the column. In ILWIS 1.4, dependent columns did not exist.

Expression
The expression usually contains operators and/or functions to specify what calculation has to be performed such as +, -, /, etc. An overview of Tabcalc operators and functions is presented in the next topic. Normally, the expression performs the calculation on all records of the columns mentioned in the expression.

Further:

- When using class names or IDs in an expression, ”double quotes” are needed around the class names or IDs.
- Instead of using class names or group names in an expression, you can also use the codes of class names or group names. These codes can be an abbreviation of your class or group names.

☞ It does not matter whether you type spaces around the definition or assign symbol, or in the expression or not.

☞ It does not matter whether you type in capitals or in small characters; a TabCalc formula is not case-sensitive.

☞ The output column may be a new or an existing column. When you specify a new output column name, the results are written into a newly created column. When you specify an existing output column name, the values in that column are overwritten by the new results.

☞ TabCalc statements do not really have a limitation on. When the command line seems full, you can just continue typing. With the Left and Right Arrow -keys on the keyboard, you can move back and forth in your formula.

☞ The TabCalc command line has a history: use the Up Arrow-key on the keyboard to retrieve previously used expressions.

☞ Advanced users who are sure about the correctness of an expression may end their expression with a semi-colon(;). In this way, dialog boxes are skipped by accepting all the default values.

6.3.3 Description of Operators and functions

There are some differences in operations available for calculations on columns with domain Value and domain Class, ID, Group or String. Therefore they are listed and explained separately.

6.3.3.1 Operators on domain Value and Image

Arithmetic operators

+	*a* + *b*	add operator;
-	*a* - *b*	subtract operator;
*	*a* * *b*	multiply operator;
/	*a* / *b*	divide operator;
MOD	*a* MOD *b*	modulus operator; returns the remainder of *a* divided by *b*, e.g. 10 mod 3, returns 1
DIV	*a* DIV *b*	integer division operator; dividing integer *a* by *b* returns the quotient, e.g. 10 div 3, returns 3

Example of the + operator

```
Column3 = column1 + column2
```

column1	column2	column3
1	10	11
2	20	22
3	30	33
4	40	44

Each record in `column3` (new or existing) stores the sum of the values of the records of existing `column1` and `column2`.

Example of the / operator

```
PopDens = Populat/Areaha
```

Populat	Areaha	PopDens
10000	37	270
9000	61	148
8000	51	157
3000	36	83

This expression calculates the ratio of the columns `Populat` (Population) and `Areaha` (area in hectares) and stores the result in column `PopDens` (the population density). Column `PopDens` may already exist; in that case it is overwritten. It may also not exist yet; in that case it is created.

Example of the MOD and DIV operators

```
ResMod = col1 MOD col2
ResDiv = col1 DIV col2
```

col1	col2	ResMod	ResDiv
10	5	0	2
11	5	1	2
12	5	2	2
13	5	3	2

Column `ResMod` contains the result of `col mod col2`. The mod operator divides `col1` by `col2` and returns the remainder of the division.
Column `ResDiv` contains the result of `col1 div col2`. The `div` operator divides `col1` by `col2` and returns the quotient.

Relational operators

=	eq	*a* = *b*	*a* eq *b*	equal to
<	lt	*a* < *b*	*a* lt *b*	less than
<=	le	*a* <= *b*	*a* le *b*	less than or equal to
>	gt	*a* > *b*	*a* gt *b*	greater than
>=	ge	*a* >= *b*	*a* ge *b*	greater than or equal to
<>	ne	*a* <> *b*	*a* ne *b*	not equal to

When a relational operator is used, ILWIS tests whether the outcome of the statement containing this operator is true or false (Bool domain).

If for a certain record the expression is true, True is assigned for that record in the new column and when not it returns False. You may want to assign values in the column rather that True and False. Then you need to use the IFF function.
On non-value columns, only the =, eq, <>, ne operators apply.

When using the symbols it does not matter whether you type spaces around the operators or not. When using the characters, spaces are required on both sides of the operator.

Example of relational > operator (greater than)

```
Suitable = Soildep > 0.20
```

This expression may also be written as:

```
Suitable = Soildep gt 0.20
```

Soildep	Suitable
0.15	False
0.45	True
0.90	True
0.20	False

If the value of `Soildep` (soil depth) is greater than 0.20, the expression is True which appears in the output column `Suitable`. For values less than 0.20 the expression is False.

Relational operators are often used in combination with a conditional IFF function. To test whether values lie in a certain range, e.g. are greater than 0.20 but less than 0.80, the INRANGE function can be used.

6.3.3.2 Operators on domain Bool

Logical operators

AND	*a* and *b*	returns true if both expressions *a* and *b* are true
OR	*a* or *b*	returns true if one or both of the expressions *a* and *b* is true
XOR	*a* xor *b*	returns true if only one of the expressions *a* and *b* is true
NOT	not *a*	returns true if expression *a* is false

Examples of logical AND operator

```
ResAnd1 = (colA >= 40) AND (colB >= 70)
ResAnd2 = (colA >= 50) AND (colA <= 150)
```

colA	colB	ResAnd1	ResAnd2
5	5	False	False
10	90	False	False
20	5	False	False
30	75	False	False
40	80	True	False
75	20	False	True
110	40	False	True
155	35	False	False
200	85	True	False
230	90	True	False

The first expression is true for those records :
- where the expression colA greater than or equal to 20 is true, and
- where, at the same time, the expression colB is greater than or equal to 70

The second expression is true for those records:
- where the value in column ColA is greater than or equal to 50, and
- where at the same time ColA is less than or equal to 150.

If you want to assign values instead of True and False, use a conditional IFF function. Testing whether values lie in a certain range can also be performed with the INRANGE function.

Examples of logical OR operator
The result of A or B is true if either expression A, or expression B or if both expressions A and B are true. In mathematical terms, the logic or is called a union.

```
ResOr1 = (colA >= 40) OR (colB >= 70)
ResOr2 = (colA < 50) OR (colA > 150)
```

colA	colB	ResOr1	ResOr2
5	5	False	True
10	90	True	True
20	5	False	True
30	75	True	True
40	80	True	True
75	20	True	False
110	40	True	False
155	35	True	True
200	85	True	True
230	90	True	True

If you want to assign values instead of True and False, you can use a conditional IFF function.

Example of logical XOR operator
The result of A xor B is only True if either one of the expressions A and B is true. If both expressions A and B are true, or if both expressions A and B are false, the record is assigned False.

```
ResXor = (colA >= 20) XOR (colB >= 70)
```

colA	colB	ResXor
5	5	False
10	90	True
20	5	True
30	75	False
40	80	False

If you want to assign values instead of True and False, you can use a conditional IFF function.

Example of logical NOT operator
The result of NOT (A) is true when the expression A is not true.

```
ResNot = NOT(colA >= 20)
```

colA	ResNot
5	True
10	True
20	False
30	False
40	False

If you want to assign values instead of True and False, you can use a conditional IFF function.

6.3.3.3 Functions on domain Value and Image

Conditional IFF function

IFF(*a*,*b*,*c*)	If condition *a* is true, then return the outcome of expression *b*, else (when condition *a* is not true) return the outcome of expression *c*. Mind the double ff in iff (standing for IF Function).

- ☞ The conditional IFF may be used for all types of input data: values, IDs, Groups and classes.
- ☞ The IFF function is used very often and may be combined with other operators and functions.

Example of a conditional IFF function
The most simple kind of expression containing an IFF function looks like:

```
Suitable = IFF(Soildep > 0.20, 50, 1)
```

Soildep	Suitable
0.15	1
0.45	50
0.90	50
0.20	1

If the condition `Soildep` is greater than 0.20, then value 50 is assigned in column `Suitable`, else, if `Soildep` is less than or equal to 0.20, value 1 is assigned.

Examples of a conditional IFF with a logical operator
Three new columns are calculated using the soil depth and the phosphate content.

```
Result1 = IFF((Soildep>0.20) AND (Phosph>10), 1, 0)
Result2 = IFF((Soildep>0.20) OR (Phosph>10), 100, 5)
Result3 = IFF((Phosph<10) OR (Phosph>30), ?, Phosph)
```

Soildep	Phosph	Result1	Result2	Result3
0.15	8	0	5	?
0.45	34	1	100	?
0.90	10	0	100	10
0.20	25	0	100	25
1.25	18	1	100	18

In the first expression a logical AND was used and for only two records the expression was true resulting in a value of 1. All other records in column `Result1` were assigned a 0.

In the second expression the logical OR was used and the value 100 was returned when the expression was true. If the expression was false the value in the new column was 5.

In the third expression also a logical OR was used returning an undefined (?) when the expression was true and returning the value of the column `Phosph` when the expression was false. Assigning undefined values may be useful when you want to exclude extreme values from further calculations. The same result can be obtained using the INRANGE function:

```
Result3 = IFF(INRANGE(Phosph,10,25), Phosph, ?)
```

Examples of nested conditional IFF functions

You may use more than one IFF function in one expression, a so-called nested conditional IFF function. It is a kind of stepwise specification of, and assignment of values to certain conditions.

```
Result1 = IFF(Soildep>0.20, IFF(Phosph>10, 1, 2), 3))

Result2 = IFF(Soildep<0.20, 1, IFF(Soildep< 0.40, 2,
          IFF(Soildep<0.80, 3, IFF(Soildep< 1.20, 4,0))))
```

Soildep	Phosph	Result1	Result2
0.15	8	3	1
0.45	34	1	3
0.90	9	2	4
0.20	25	3	2
1.25	18	1	0

The first expression nests two IFF statements. Value 1 is assigned to column `Result1` if `Soildep` is greater than 0.20, and if also `Phosph` is greater than 10. If either one or both of the statements are false, value 0 is assigned. In other words: both requirements should be met in order to assign value 1. The same result could have been obtained by using the logical AND.

The second expression nests four IFF statements. In words the expression means: If column `Soildep` is less than 0.20, then assign a 1 to column `Result2,` else if column `Soildep` is less than 0.40, then assign a 2 to column `Result2,` else if column `Soildep` is less than 0.80, then assign a 3 to column `Result2,` else if column `Soildep` is less than 1.20, then assign a 4 to column `Result2,` else assign a 0 to column `Result2`.

With this kind of expression you can (re)classify values of a column. You may also use the special function classify (CLFY) which is explained later in this chapter.

INRANGE function

INRANGE(a,b,c)	tests whether values of expression or map *a* are contained by a range or closed interval with endpoints *b* and *c*. Mathematic notation: $b \leq a \leq c$ or $a \in [b;c]$

Example INRANGE function

The INRANGE(*a,b,c*) function tests whether value (*a*) is contained by a range or closed interval including the end points *b* and *c*. Mathematic notation: $b \leq a \leq c$ or $a \in [b;c]$. Below, the INRANGE function is combined with a conditional IFF. Not only values may be assigned; you can also use strings.

```
Result1 = IFF(INRANGE(Value,100,1000),Value,?)

Result2 = IFF(INRANGE(Value,100,1000),"Normal","Out of
          Range")
```

Value	Result1	Result2
306.309	306.309	Normal
501.626	501.626	Normal
37.926	?	Out of Range
325.583	325.583	Normal
484.975	484.975	Normal
375.206	375.206	Normal
2305.375	?	Out of Range

Original values are assigned to column Result1 if the values are in the specified range. Else the result is undefined (?). In column `Result2` the string "`Normal`" appears if the values are in the specified range. Else, the result is again undefined.

Exponential functions

SQ(*a*)	a^2	square function: *a*a*; *a* square
SQ(*a,b*)	$a^2 + b^2$	square function: *a*a + b*b*; *a* square plus *b* square
SQRT(*a*)	$\sqrt{a}$	square root function: calculates the positive square root of *a*
HYP(*a,b*)	$\sqrt{(a^2 + b^2)}$	hypotenuse: calculates the positive square root of the sum of *a* square and *b* square
POW(*a,b*)	a^b	exponential function: *a* raised to the power *b*. The n-th root of *a* is found by using this function in the form of: POW(*a*, 1/n)
EXP(*a*)	e^a	exponential function: value e (i.e. 2.718) raised to the power *a*

Memory refreshment: Laws of exponents

Negative exponents: $a^{-n} = 1/a^n$

Fractional exponents: $a^{m/n} = \sqrt[n]{(a^m)}$

Multiplication: $a^n a^m = a^{n+m}$

Division: $a^n/a^m = a^{n-m}$

Raising to a power: $(a^n)^m = a^{n*m}$

Example of square function SQ(a)

Function SQ(*a*) multiplies *a* with itself.

```
col2 = SQ(col1)
col3 = SQ(col2 - col1)
```

col1	col2	col3
2	4	4
4	16	144
6	36	900
8	64	3136

`Col2` stores the result of `col1 * col1`.
`Col3` stores the result of `(col2-col1) * (col2-col1)`.

Example of square function SQ(a,b)

Function SQ(*a*,*b*) sums the squares of arguments *a* and *b*.

```
col3 = SQ(col1,col2)
```

col1	col2	col3
2	10	104
4	20	416
6	30	936
8	40	1664

`Column3` stores the result of `(col1*col1) + (col2*col2)`.

Square root function SQRT(*a*)

Function SQRT(*a*) returns the value that when multiplied by itself gives the input value or expression.

```
col2 = SQRT(col1)
```

col1	col2
4	2
16	4
25	5
30	5.477

`Col2` stores the square root of `col1`.

Hypotenuse function HYP(*a,b*)

Function HYP(*a,b*) returns the root of the sum of *a* square and *b* square. In mathematic notation: $\sqrt{(a^2 + b^2)}$. Function HYP finds the side opposite the right angle in a right-angled triangle. This function is used often in slope calculations.

```
colC = hyp(colA,colB)
```

colA	colB	colC
1	1	1
1	3.873	4
1	2	2.236
3	4	5

`ColC` stores the result of `√((colA*colA)+(colB*colB))`

Example of exponential function POW(*a,b*)

Exponential function POW(*a,b*) raises *a* to the power *b*. Mathematic notation: E. Using the function in the form of POW(*a*, 1/*b*) returns the *b*-th root of *a*; see also the laws of exponents.

```
Col3 = POW(col1,col2)
Col4 = POW(col1,3)
Col5 = POW(col1,1/3)
```

col1	col2	col3	col4	col5
2	2	4	8	1.260
4	4	256	64	1.587
8	8	16777200	512	2.000

`Col3` stores the result of `col1` to the power `col2`.
`Col4` stores the result of `col1` to the power 3.
`Col5` stores the result of the third root of `col1`.

Example of exponential function EXP(a)

Exponential function EXP(*a*) raises e (= 2.718) to the power *a*.
Mathematic notation: e^a.
Constant value `e` can be used on the command line, by typing EXP(1).

```
col2 = EXP(col1)
```

col1	col2
1	2.718
2	7.389
4	54.598
8	2980.960

`col2` stores the results of e^{col1}.

Additional information: Introduction on exponential growth functions

`An = A0*(1 + perc/100)^n` and `An = A0*e^perc/100*n`

`An`	the amount after n years;
`A0`	the amount in year 0;
`perc`	the growth rate as a percentage per year;
`n`	the number of years.

Example
The expected total residential area after 3 years, of a residential area of 250 ha with an estimated growth rate of 5% per year, is:

Area3y = 250 * $(1.05)^3$ = 289 ha	(first formula)
`Area3y = 250 POW(1.05,3)`	(TabCalc syntax)

Area3y = 250 * $e^{0.05*3}$ = 290 ha	(second formula)
`Area3y = 250 EXP(0.05*3)`	(TabCalc syntax)

Exponential growth with POW and EXP functions
Introduction on growth functions:

$An = A0 * (1 + perc/100)^n$ ⇒ `An = A0* POW(1+perc/100, n)`
$An = A0 * e^{perc/100*n}$ ⇒ `An = A0 * EXP(perc/100 * n)`

```
AreaHaY4 = AreaHaY0 * POW(1 + (GrowPerc/100), 4)
AHaY4 = AreaHaY0 * EXP(GrowthPerc/100 * 4)
```

LandUse	AreaHaY0	GrowPerc	AreaHaY4	AHaY4
residential	1136	6	1434	1444
commercial	425	4	497	499
industrial	173	3	195	195
institutional	137	1	143	143
recreational	77	2	83	83
transportation	18	1	19	19

Column `AreaHaY0` contains the areas in hectares for each land use class in year 0;
column `GrowPerc` contains growth rates per year;
column `AreaHaY4` contains the areas of the land use classes after 4 years according to the first formula and column `AHaY4` according to the second formula.

Logarithmic functions

LOG(*a*)	$^{10}\log(a)$	calculates the base 10 logarithm of *a*
LN(*a*)	$^{e}\log(a)$	natural logarithm, calculates the base e (2.718) logarithm of *a*

Examples of LOG and LN functions

```
ExampleLOG = LOG(column)
ExampleLN   = LN(column)
```

Column	ExampleLOG	ExampleLN
1234	3.091	7.118
2345	3.370	7.760
3456	3.539	8.148
13579	4.133	9.516
24680	4.392	10.114

Column `ExampleLOG` contains the 10-based logarithm of column `Column`.
Column `ExampleLN` contains the e-based logarithm (or natural logarithm) of column `Column`.

Memory refreshment: Laws of logarithms

$\log(n*m) = \log(n) + \log(m)$

$\log(n/m) = \log(n) - \log(m)$

$\log(n^m) = m * \log(n)$

$\ln(EXP(n)) = n$

If you would like to calculate ${}^b\log(a)$, the *b*-based logarithm of *a*, instead of ${}^{10}\log(a)$, you can divide the old logarithm by the logarithm of the new base:

${}^b\log(a) = {}^{10}\log(a)/{}^{10}\log(b)$

Example of a logarithmic function

```
Result2log = LOG(column1)/LOG(2)
Result4log = LOG(column1)/LOG(4)
```

Column1	Result2log	Result4log
16	4	2
32	5	2.5
64	6	3
256	8	4

Column `Result2log` contains the values of ${}^2\log(\text{column1})$.
Column `Result4log` contains the values of ${}^4\log(\text{column1})$.

Random number generators

For certain statistical analyses you might want to create columns with random values. For instance to simulate a point data set for comparison with measured point data (Spatial Correlation and Pattern Analysis).

The following random number generators are available:

RND(*n*)	returns integer values in the range 1 to *n*. To simulate a die, use this function in the form of: RND(6)
RND(0)	returns a 0 or 1 at random
RND()	returns random real values in the range [0;1> , i.e. between 0 and 1, including 0 but excluding 1

☞ Use always := otherwise new random values will be assigned in every new calculation.

☞ Integer *n* has a maximum value of 2 billion ($2*10^9$)

Examples of the random functions

```
Random1 := RND(1000)
Random2 := RND(0)
Random3 := RND()
```

Random1	Random2	Random3
371	1	0.803
61	1	0.615
992	0	0.412
340	1	0.107
760	1	0.501

Column `Random1` contains long integer values in the interval [1;1000].
Column `Random2` contains a 0 or 1 at random.
Column `Random3` contains real values between 0 and 1.

Sign operator and functions

-(*a*)	returns *a* multiplied by -1
NEG(*a*)	returns *a* multiplied by -1
ABS(*a*)	returns the absolute (= positive) value of *a*
SGN(*a*)	returns -1 for negative values of *a*, 0 when *a* is 0, and 1 for positive values of *a*

Examples of the sign functions

```
ColNeg = -Col1
ColNeg = NEG(Col1)
ColAbs = ABS(Col1)
ColSgn = SGN(Col1)
```

Col1	ColNeg	ColAbs	ColSgn
56.48	-56.48	56.48	1
-42.72	42.72	42.72	-1
-5.10	5.10	5.10	-1
31.88	-31.88	31.88	1
19.33	-19.33	19.33	1
-21.92	21.92	21.92	-1

Column `ColAbs` contains the absolute values of column `Col1`.
Column `ColNeg` contains the value of column `Col1` multiplied by -1.
Column `ColSgn` contains a 1 when the value in `Col1` is positive, and -1 when negative.

Rounding functions

ROUND(*a*)	rounds *a* off to a long integer,
FLOOR(*a*)	rounds down; returns the largest long integer value smaller than input value (truncation),
CEIL(a)	rounds up; returns the smallest long integer value larger than input value,

Examples of rounding functions

```
Round1= ROUND(Col1)
Floor1= FLOOR(Col1)
Ceil1 = CEIL(Col1)
```

Col1	Round1	Floor1	Ceil1
56.48	56	56	57
-42.72	-43	-43	-42
-5.10	-5	-6	-5
31.88	32	31	32

Column `Round1` contains the rounded values of column `Col1`.
Column `Floor1` contains the truncated values of `Col1`.
Column `Ceil1` contains the smallest integer value larger than the values in `Col1`.

MinMax functions

MIN(*a*)	calculates the minimum value of all records that are the outcome of column *a*
MIN(*a*,*b*)	calculates the minimum value of expressions *a* and *b*
MIN(*a*,*b*,*c*)	calculates the minimum value of expressions *a*, *b* and *c*
MAX(*a*)	calculates the maximum value of all records that are the outcome of column *a*
MAX(*a*,*b*)	calculates the maximum value of expressions *a* and *b*
MAX(*a*,*b*,*c*)	calculates the maximum value of expressions *a*, *b* and *c*

Examples of MinMax functions

```
ResMin = MIN(colA)
ResMax = MAX(colA,colB)
```

colA	colB	ResMin	ResMax
13	57	13	57
34	93	13	93
75	42	13	75
29	19	13	29
18	66	13	66

The first formula returns the minimum value of colA.
The second formula compares the record values in colA and colB and returns the largest value of the two.

Average function AVG(a)

AVG(*a*)	returns the average value of column *a*
STDEV(*a*)	returns the standard deviation of column *a*

Example of the Average function

```
Average = AVG(Area)
Standdev = STDEV(Area)
```

Area	Average	Standdev
1000	2500	1118
2000	2500	1118
3000	2500	1118
4000	2500	1118

The column `Average` contains the average value of all values in column `Area`. Column `Standdev` contains the standard deviation of column `Area`.

To calculate weighted averages or averages per class refer to: `AGGAVG` function in the section aggregations under special Table Calculations.

Trigonometric functions

SIN(*a*)	sine (input angles specified in radians); returns real values in the range -1 to 1
COS(*a*)	cosine (input angles specified in radians); returns real values in the range -1 to 1
TAN(*a*)	tangent (input angles specified in radians)
ASIN(*a*)	arc sine (input values must be in the range -1 to 1); returns real values in radians in the range $-\pi/2$ to $\pi/2$
ACOS(*a*)	arc cosine (input values must be in the range -1 to 1); returns real values in radians in the range 0 to π
ATAN(*a*)	arc tangent; returns real values in radians in the range $-\pi/2$ to $\pi/2$
ATAN2(y,x)	returns the angle in radians of two input values; x is horizontal, y is vertical

- ATAN(y/x) = ATAN2(x,y) if x and y are both positive.
- The function ATAN and especially ATAN2 are often used in calculation of slope maps and aspect maps. Then use RADDEG(ATAN2(DX,DY)+PI).
- For the functions SIN, COS and TAN, the input angles have to be specified in radians. To convert degrees to radians, use the angular function DEGRAD. For example, the formula SIN(DEGRAD(60)) calculates the sine of 60°.
- For the functions ASIN, ACOS, ATAN and ATAN2, the output values are in radians. To convert radians to degrees, use the angular function RADDEG. For the functions ASIN and ACOS, the input values must be in the range -1 to 1.

Examples Trigonometric functions

```
Result1 = tan(colA)
Result2 = acos(colB)
```

colA	colB	Result1	Result2
-3.82	0.62	-0.8060	0.9021
-1.67	0.34	10.0472	1.2239
-0.75	-0.86	-0.9316	2.6061
0.33	-0.72	0.3425	2.3746
0.98	-0.03	1.4910	1.6008
1.41	-0.66	6.1654	2.2916
2.74	-0.48	-0.4247	2.0715

Result 1 shows the tangent of the values (in radians) of colA.
Result 2 shows the arccosine of the values (also in radians) in colB. These values lie between 0 and π.

Angular functions

DEGRAD(*a*)	converts degrees to radians; *a**2π/360
RADDEG(*a*)	converts radians to degrees; *a**360/2π MOD 360

The DEGRAD function converts the degree values in column a to radians: a is multiplied with 2π/360. The DEGRAD function is often used in combination with trigonometric functions. For example, to calculate the tangent of 60°, you use the following formula: `TAN(DEGRAD(60))`

Example of degrees to radians function DEGRAD(*a*)
To calculate the tangent of `col1` (in degrees), use the following expression:

```
col2 = (TAN(DEGRAD(col1))
```

col1	col2
30	0.577
45	1
60	1.732
90	?

Example of radians to degree conversion RADDEG(*a*)
The RADDEG function converts radian values in column *a* to degrees: *a* is multiplied with 360/2π MOD 360. ILWIS uses a default range of 0 - 360 with a precision of 0.01. The RADDEG function is often used in combination with trigonometric functions.

When you want to convert slope values in percentages (`slopepct`) to degrees (`slopedeg`), use the following expression:

```
slopedeg = RADDEG(ATAN(slopepct/100))
```

slopepct	slopedeg
10	5.71
25	14.04
50	26.57
100	45.00
173	59.97

Hyperbolic functions

SINH(*a*)	hyperbolic sine; $(e^a - e^{-a})/2$
COSH(*a*)	hyperbolic cosine; $(e^a + e^{-a})/2$
TANH(*a*)	hyperbolic tangent; tanh(*a*) = sinh(*a*)/cosh(*a*)

Hyperbolic functions are related to a hyperbola ($x^2 - y^2 = r^2$), in the same way as trigonometric functions are related to a circle. In the formulas above, *a* represents the x of the hyperbole.

6.3.3.4 Predefined values

ILWIS includes the following predefined values:

PI	value π : 3.141593...
PI2	value 2 * π : 6.283185...
PIDIV2	value ½ π : 1.570796...
PIDIV4	value 1/4 π: 0.785398...
EXP(1)	value e : 2.718282...

Predefined values may be used in every calculator (MapCalc, TabCalc, user-defined functions, and the pocket line calculator). Predefined values related to π are mostly used in combination with trigonometric functions.

Examples of Predefined PI

Using the following formula you can calculate what the radius would be from a circle as large as an area in your map. The column area can be found in a histogram table.

```
Radius = SQRT(Area/PI)
```

Area	Radius
4320890	1173
874283	528
74294	154
6286446	1415

The following formula converts degrees into radians without using the DEGRAD function:

```
radians = degrees * (PI2)/360
```

degrees	radians
15	0.26
30	0.52
45	0.79
60	1.05

6.3.3.5 Operators and functions on non-value columns (Class, Group, ID)

Below, the operators and functions are explained that can be used in Table calculation on columns with a domain Class, Group or ID. For an overview of all operators and functions available, please, refer to the schematic overview at the start of this chapter.

When using class or group names or IDs within an expression, these names and IDs should be put between double quotes, e.g. ”coffee”. In domains of the Class or Group type, you can enter codes and class names/group names. These codes can be an abbreviation of your class or group names. In expressions, also the codes can be used.

Relational operators

=	eq	*a* = *b*	equal to; tests whether the outcome of expression *a* is equal to the outcome of expression *b*
<>	ne	*a* <> *b*	not equal to; tests whether the outcome of expression *a* is not equal to the outcome of expression *b*

Relational operators on Class, ID, Group or String columns

```
Resident  = (Landuse = ”residential”)
Other = (Landuse <> ”residential”)
```

Parcel	Landuse	Resident	Other
00123	Residential	True	False
00124	Residential	True	False
00125	Commercial	False	True
00126	Residential	True	False
00127	Industrial	False	True
01272	Institutional	False	True
04625	Residential	True	False

Column Resident shows for which parcels the first expression is True or False. Column Other shows for which parcels the second expression is True or False.

Logical operators

AND	(*a*) AND (*b*)	logical and (intersection), returns true if both expressions *a* and *b* are true
OR	(*a*) OR (*b*)	logical or (union), returns true if one or both of the expressions *a* or *b* is true
XOR	(*a*) XOR (*b*)	logical xor, returns true if only one of the expressions *a* and *b* is true
NOT	NOT (*a*)	logical not, returns true if expression *a* is false

Examples logical operators on domain Class or ID columns

```
ResAND = (Landuse="residential") AND (CommVal > 20000)
ResOR  = (Landuse="commercial") OR (Landuse="industrial")
```

Parcel	Landuse	CommVal	ResAND	ResOR
00123	Residential	12500	False	False
00124	Residential	22500	True	False
00125	Commercial	45000	False	True
00126	Residential	30000	True	False
00127	Industrial	70000	False	True
04625	Residential	19000	False	False

Column `ResAND` is created from conditions on 2 other columns: `Landuse` and `CommVal` (commercial value). When a record in column `Landuse` is residential, and that record number has also a commercial value greater than 20000, then the expression is True. In other cases False is returned.
In column `ResOR` True is assigned for every record which has a commercial or industrial landuse, else False.

Conditional IFF function

IFF(*a*,*b*,*c*)	if condition *a* is true, then return the outcome of expression *b*, else return the outcome of expression *c*.

Example of a conditional IFF function:
A simple expression containing an IFF function may look like:

```
Result =  IFF (Landuse <> "Residential", "Other",  Landuse)
```

Parcel	Landuse	Result
00123	Residential	Residential
00124	Residential	Residential
00125	Commercial	Other
00126	Residential	Residential
00127	Industrial	Other
04625	Residential	Residential

If the Landuse is any other than "`Residential`" the description of the land use is replaced by "`Other`", else it will remain "`residential`".

The IFF function is used very often and may be combined with other operators and functions.

```
Result1 = IFF((Landuse = "forest") AND (altitude > 1600),
          100, 0)
Result2=  IFF((Landuse = "crops"), Landuse, IFF(INRANGE
          (altitude,1400,1700), "suitable","not suitable"))
```

Parcel	Landuse	Altitude	Result1	Result2
100	crops	1450	0	crops
127	forest	1675	100	suitable
131	forest	1840	100	not suitable
146	meadow	1725	0	not suitable
149	crops	1365	0	crops
177	forest	1490	0	suitable
180	crops	1650	0	crops
197	meadow	1585	0	suitable
198	crops	1720	0	crops

`Result1` assigns a value 100 to the records which have a land use type forest and which lie below an altitude of 1600 m. All other records get a value 0.

`Result2` is a string column. If the landuse is crops, `Result2` is assigned the word `crops`. Then, iff the altitude is between 1400 and 1700 m the string `suitable` is assigned else the string, else a parcel is classified as `not suitable`.

Additional operators and functions on Class/ID

+	s1 + s2	concatenation operator, glues *s1* and *s2* together
IN	IN(*s1, s2*)	tests whether *s2* is part of *s1*
INMASK	INMASK(col, s)	tests whether complete string *s* is present in a non-value column *col*; wildcards * and ? are allowed
STRPOS	STRPOS(*s1,s2*)	returns the starting position of *s2* in *s1*
LENGTH	LENGTH(*s*)	returns the number of characters (the length) of string *s*
SUB	SUB(*s,int1, int2*)	returns a substring of string *s*; starting with the character at position *int1* and returns *int2* number of characters
LEFT	LEFT(*s, int*)	returns the first int number of characters of string *s*
RIGHT	RIGHT(*s, int*)	returns the last int number of characters of string *s*
STRLT	STRLT(*s1, s2*)	returns true if in alphabetical order string *s1*
<	*s1 < s2*	comes before string *s2*
STRLE	STRLE(*s1, s2*)	returns true if in alphabetical order string *s1*
<=	*s1 <= s2*	comes before or on the same place as string *s2*
STRGT	STRGT(*s1, s2*)	returns true if in alphabetical order string *s1*
>	*s1 > s2*	comes after string *s2*
STRGE	STRGE(*s1, s2*)	returns true if in alphabetical order string *s1*
>=	*s1 >= s2*	comes after before or on the same place as string *s2*

Examples of additional functions on Class, ID, Group or String columns

```
Combined  = Terrain + LandUse

Result    = IN(combined,"H1F")
```

Terrain	DescrTerr	LandUse	Description Landuse	Combined	Result
C2	LagoonPlain	Pr	PaddyRiceRain	C2Pr	False
F	CollFootslope	H	HomestGarden	FH	False
H1	VolcHillsMod	Fb	SecForest	H1Fb	True
H1	VolcHillsMod	G	Grazing	H1G	False
H1	VolcHillsMod	Ut	UplandCropsTrees	H1Ut	False
H2	VolcHillsSteep	Fp	PrimForest	H2Fp	False
H2	VolcHillsSteep	Ut	UplandCropsTrees	H2Ut	False
U1	CoastUpland	G	Grazing	U1G	False

The first expression results in a column `Combined` which contains a combination of the class names in columns `Terrain` and `LandUse`.
The second expression is true for moderately steep hills (`H1`) with forest (`Fb or Fp`); in that case True is assigned in Column `Result`. Mind that using "h1f" would give a different result.

The INMASK function tests whether a specified search string is present in the mask presented in a column or a map. This offers the opportunity for a simple check if certain words, characters, numbers, codes, etc. are a part of one of the items in a column or map domain. You may use the wildcard ? when you are not interested in a character or number on a certain position. The wildcard * is used to omit one or more characters in your mask.

Examples of the INMASK function

```
Result1 =  INMASK(Combined,"*2*")

Result2 =  INMASK(Combined,"H?U*")

Result3 =  IFF(INMASK(Combined,"H?U*"),landuse,("not
           suitable")
```

Combined	Description Landuse	Result1	Result2	Result3
C2Pr	PaddyRiceRain	False	False	not suitable
FH	HomestGarden	False	False	not suitable
H1Fb	SecForest	False	False	not suitable
H1G	Grazing	False	False	not suitable
H1Ut	UplandCropsTrees	False	True	UplandCropsTrees
H2Fp	PrimForest	True	False	not suitable
H2Ut	UplandCropsTrees	True	True	UplandCropsTrees
U1G	Grazing	False	False	not suitable

Example of the STRPOS function

To look for the starting position of a substring in a certain column you may use the STRPOS function. In this example an attribute table of a geomorphology map is used. The starting position of the word "slope" will be retrieved if present in column Description.

```
Slopepos = STRPOS(Description, "slope")
```

	Description	Slopepos
aaf	active alluvial fan	0
dhm	denudational steep to very steep slopes	34
dsh	denudational moderately steep slopes	31
fls	fault-line scarp	0
frh	fault-related hills	0
fsl	steep face-slopes	12
fld	infilled lake	0
svd	steep dipslopes	10
syc	synclinal ridge	0

This might also be useful when you want to find out the starting position of your second input domain when using the domain of a cross table.

The next example uses the cross table of a landuse and a geomorphology map.

```
Pos = STRPOS(%k,"* ") + 2
```

	Pos
Airport * fld	11
Bare rock * dsh	13
Bare rock * fsl	13
Bare rock * svd	13
Crops no irrigation * aaf	23
Crops no irrigation * frh	23
Crops no irrigation * fld	23

In column Pos the starting position is found of sub-string * followed by a space. Then 2 is added to find the real starting position of the geomorphology code.

Examples of the other additional functions

```
Result1 = LENGTH(Input)
Result2 = SUB(Input,6,4)
Result3 = LEFT(Input,5)
Result4 = RIGHT(Input,6)
Result5 = IN(Input,"long")
```

Input	Result1	Result2	Result3	Result4	Result5
QuiteLongString	15	Long	Quite	String	True
AnotherString	13	erSt	Anoth	String	False

Column `Result1`	gives the length of the strings in column `Input`.
Column `Result2`	returns a substring of column `Input`, starting at position 6, and 4 characters long.
Column `Result3`	returns the first 5 characters of column `Input`.
Column `Result4`	returns the last 6 characters of column `Input`.
Column `Result5`	returns a 1 if ”long” is part of the string in column Input, and a 0 if not.

To test whether a string is placed in alphabetical order before, after or in the same position as *string2* you can use 4 functions:

```
Result6 = STRLT(Code,”h”)
Result6 = Code < ”h”

Result7 = IFF(STRLE(Code,”hd3”), 100, 5)
Result7 = IFF(Code <= ”hd3”, 100, 5)

Result8 = STRGT(code,code[%r-1])
Result8 = code > code[%r-1]

Result9 = STRGE(Code,”h”)
Result9 = Code >= ”h”
```

Code	Result6	Result7	Result8	Result9
ac3	True	100	?	False
ad1	True	100	True	False
ud3	False	5	True	True
pc1	False	5	False	True
hc2	False	100	False	True
hd3	False	5	True	True
bc2	True	100	False	False
bd3	True	100	True	False

`Result6` returns True if the string in column `Code` comes before character ”h” when you place them in alphabetical order.

When alphabetically ordered the string in column `Code` comes before or on the same position as the specified ”hd3”, then 100 is returned in column `Result7`, else 5.

In column `Result8` True is returned when the code comes later in the alphabetical order than the code of the record one position above it in the table. Else it returns False.

`Result9` returns True when the code starts with an ”h” or characters later in the alphabet. Else it returns False.