



Vitesse Basic



Revision Log

| Date | Page(s) | Revision | Description of Revision | Release |
|------------|--------------------------|------------|---|---------|
| 22/04/2004 | All | T.Lisowski | General Update for M3 | M3 |
| 18/05/2004 | sections 4.2.2, 4.2.3 | T.Lisowski | Corrected interpretation of Data Extraction status code of 0 | M3 |
| 21/07/2004 | All | T.Lisowski | General review and update for M3 SP1 | M3 SP1 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Updates

Updates to this manual will be issued as replacement pages and a new Update History Sheet complete with instructions on which pages to remove and destroy, and where to insert the new sheets. Please ensure that you have received all the updates shown on the History Sheet.

All updates are highlighted by a revision code marker, which appears to the left of new material.

Suggestion/Problems

If you have a suggestion about this manual, the system to which it refers, or are unfortunate enough to encounter a problem, please report it to the training department at

Fax +44 191 201 0001

Email training@tribon.com

Copyright © 2004 Tribon Solutions

All rights reserved. No part of this publication may be reproduced or used in any form or by any means (graphic, electronic, mechanical, photocopying, recording, taping, or otherwise) without written permission of the publisher.

Python 2.3 is Copyright © 2001-2003 Python Software Foundation

Printed by Tribon Solutions (UK) Ltd on 17 January 2005

| | | |
|----------|---|-----------|
| 1 | Introduction | 7 |
| 1.1 | Objectives | 7 |
| 1.2 | Prerequisites for training course | 7 |
| 1.3 | Training methods | 7 |
| 1.4 | Overview | 8 |
| 1.5 | Duration | 8 |
| 1.6 | Using this document | 8 |
| 2 | Python Programming Language | 9 |
| 2.1 | Introduction | 9 |
| 2.2 | Assignments and Expressions | 9 |
| 2.2.1 | Numbers | 9 |
| 2.2.2 | Strings | 11 |
| 2.2.3 | % string operator | 12 |
| 2.2.4 | Converting non-strings into strings | 13 |
| 2.2.5 | String Methods | 13 |
| 2.3 | Data Structures (Lists, Tuples & Dictionaries) | 14 |
| 2.3.1 | Lists | 14 |
| 2.3.2 | Tuples | 16 |
| 2.3.3 | Dictionaries | 17 |
| 2.4 | First Steps to Programming | 18 |
| 2.4.1 | Multiple Assignments | 18 |
| 2.4.2 | WHILE Loops | 18 |
| 2.4.3 | Loop Indentation | 19 |
| 2.4.4 | Print Statement | 19 |
| 2.5 | The IF Statement | 19 |
| 2.5.1 | Conditions | 20 |
| 2.6 | The FOR Statement | 20 |
| 2.6.1 | Iterators | 21 |
| 2.7 | The RANGE() Function | 21 |
| 2.8 | List comprehension | 22 |
| 2.9 | BREAK and CONTINUE | 23 |
| 2.10 | Function definitions | 24 |
| 2.10.1 | Function's arguments | 24 |
| 2.10.2 | Generators | 25 |
| 2.11 | File handling | 26 |
| 2.12 | Exceptions: How to handle them? | 27 |
| 2.12.1 | Generator exception | 28 |
| 2.12.2 | Vitesse exceptions | 29 |
| 2.13 | Python Classes | 29 |
| 2.14 | Modules | 31 |
| 2.15 | How to choose the right data structure? | 32 |
| 2.16 | Advanced Python Programming | 33 |
| 3 | Tribon Vitesse Utilities | 35 |
| 3.1 | Introduction | 35 |
| 3.2 | Running Vitesse Programs and viewing the Output | 35 |
| 3.3 | Vitesse Python Classes | 36 |
| 3.4 | Messages | 38 |
| 3.5 | Basic requests | 40 |
| 3.6 | Point requests | 41 |
| 3.7 | Co-ordinate translation | 42 |

| | |
|---|-----------|
| Exercise 1: Creating the frame table | 43 |
| Exercise 2: Displaying co-ordinates of indicated point | 43 |
| 3.8 Selections..... | 44 |
| 3.9 Application's window management..... | 46 |
| 3.10 Batch Vitesse | 46 |
| 3.11 Miscellaneous functions | 47 |
| 4 Exploring the Tribon Product Information Model | 49 |
| 4.1 Accessing Tribon environment variables | 49 |
| 4.2 Data Extraction | 49 |
| 4.2.1 Introduction..... | 49 |
| 4.2.2 Data Extraction in Vitesse | 50 |
| 4.2.3 Advanced techniques | 51 |
| Exercise 3: Extracting transformation matrix of a panel..... | 52 |
| Exercise 4: Listing structure parts data..... | 52 |
| Exercise 5: Selecting a panel..... | 52 |
| 4.3 Listing objects in databanks | 52 |
| 5 Vitesse Drafting | 55 |
| 5.1 Introduction..... | 55 |
| 5.2 Drawing Functions | 55 |
| 5.2.1 Current drawing | 55 |
| 5.2.2 Storing the current drawing | 56 |
| 5.2.3 Activating a drawing | 57 |
| 5.2.4 Housekeeping functions | 57 |
| 5.2.5 Layer handling functions..... | 58 |
| 5.2.6 Visual area functions | 58 |
| 5.2.7 Document references | 59 |
| 5.2.8 Tribon Data Management functions | 59 |
| 5.3 Element Handles..... | 60 |
| 5.3.1 Type of elements identified by the handle | 60 |
| 5.3.2 Obtaining element handles..... | 61 |
| 5.3.3 Retrieving element's information from the handle | 63 |
| Exercise 6: Capturing model objects | 63 |
| 5.4 View Functions | 64 |
| 5.4.1 Creating or identifying the view | 64 |
| 5.4.2 Transforming the view | 64 |
| Exercise 7: Transforming a view | 65 |
| 5.4.3 View's projection..... | 65 |
| 5.4.4 Symbolic views | 65 |
| 5.4.5 Miscellaneous..... | 66 |
| 5.5 Model Handling Functions..... | 67 |
| 5.5.1 Drawing model objects | 67 |
| 5.5.2 Identifying and collecting model objects | 67 |
| Exercise 8: Displaying the weight of indicated structures | 68 |
| 5.5.3 Deleting model objects | 68 |
| 5.5.4 Modifying model object's modal properties | 68 |
| 5.5.5 Tribon Data Management functions | 68 |
| Exercise 9: Create model views..... | 69 |
| 5.6 Basic Geometry Entities | 69 |
| 5.6.1 Modal properties..... | 69 |
| 5.6.2 Creation of basic geometry entities | 70 |
| 5.6.3 Entity properties..... | 71 |
| 5.6.4 Highlighting elements | 71 |
| 5.7 Texts | 72 |
| 5.7.1 Modal properties..... | 72 |

| | | |
|---|---|-----------|
| 5.7.2 | Creation of texts | 73 |
| 5.8 | Symbols..... | 74 |
| 5.8.1 | Modal properties..... | 74 |
| 5.8.2 | Creation of symbols..... | 74 |
| 5.9 | Drawing Components | 74 |
| 5.9.1 | Hatching | 74 |
| 5.9.2 | Notes and Position Numbers..... | 75 |
| Exercise 10: Weight of a structure in a note | | 76 |
| 5.9.3 | Dimensioning..... | 77 |
| 5.9.4 | Other drawing components | 79 |
| 5.10 | Drafting Default Values | 80 |
| 5.11 | Subpicture Managing Functions | 80 |
| 5.12 | Drawing Element Handling..... | 81 |
| 5.13 | Shading functions..... | 83 |
| 5.14 | Traversing the drawing's structure | 83 |

1 Introduction

Tribon Vitesse is a way to create customised, user-developed macros in an object-oriented language with full access to the Tribon Product Information Model and Tribon Modelling functionality.

This course is designed to introduce the concept and use of Tribon Vitesse. After completion of the course the manual can also be used as a reference source in conjunction with the Tribon Vitesse User's Guide and the Vitesse source files delivered as a part of the Tribon M3 distribution.

1.1 Objectives

The aim of the course is to provide the knowledge required for creating Tribon Vitesse macros. After completing the course, the user should be in a position to immediately start using Vitesse system.

The objectives are:

- To understand the Tribon Vitesse concept.
- To become familiar with the Python programming language.
- To become familiar with the Tribon Vitesse functions in the area of:
 - Utility functions;
 - User interface;
 - Data Extraction;
 - Tribon databank access;
 - Drafting.
- To be able to create Vitesse macros.

1.2 Prerequisites for training course

During the training, the participants require access to a PC with an installation of the Tribon M3 system and the stand-alone Python interpreter with the Python source editor (e.g. PythonWin or ConTEXT).

The following skills are required from at least one person in each group:

- Working knowledge of Tribon Drafting system
- Working knowledge of Tribon Data Extraction system.
- Basic knowledge of the Tribon modelling subsystems:
 - Hull (Planar and Curved);
 - Outfitting (Structure, Pipe, Ventilation, Cable, Equipment, Volume);
 - Assembly
- Basic knowledge of Tribon databank handling
- Basic programming experience.
- A basic understanding of the Python language will be an additional benefit.

A copy of the training project must be installed for this training. It can be installed prior to the trainer arriving or installed during the first session of the training, as it is not required during this time.

1.3 Training methods

Presentations, demonstrations and practical exercises.

1.4 Overview

Tribon Vitesse is a productive way to create customised, user-developed macros or programs in an object-oriented language. These macros are then available within the interactive Tribon application through a special function on the menu.

Tribon Vitesse incorporates the programming language Python. This language has been integrated with Tribon to form the environment in which a program is created. The syntax includes all normal features in a programming language such as branching, loops, functions, etc., as well as full support for the object-oriented programming. The Python language itself gives the user the full freedom to work in a traditional function-oriented fashion. Its implementation in Vitesse, however, requires the programmer to gain also the basic skills in the object-oriented approach, because the Tribon entities and activities have been modelled in the Python language as classes and its methods.

Many tasks that earlier would have been complicated or even impossible to carry out may now be performed using Tribon Vitesse. Some examples are:

1. It is possible to create rule-based constructions, which automatically adjust to the Product Information Model, where the rules are defined by the customer. This can significantly reduce the input required by the particular Tribon application and minimise the error rate, since once the program is written it can be executed any number of times with little or no possibility for human mistakes.
2. The Product Information Model can be more easily updated, either by simply rerunning the original program or, for example in Tribon Hull Modelling, by executing a program that automatically regenerates panels.
3. The Product Information Model can be more easily visualised, for example by changing display features in a drawing depending on the status of the underlying item. This can be used to show the progress of a design, or to group models with common features for easy identification.
4. Lists may be created interactively. With full access to the Product Information Model, it is possible to create customised reports or extractions of data, for example by using an interactive identification of parts to be included.
5. Parametric structures (equipment foundations, etc.) may be generated automatically with full access to the surrounding model objects.
6. 2D drawing elements (e.g. 2D primitives in General Design), or even the whole drawings (e.g. profile sketches) may now be generated automatically. It is also possible to create various kinds of model views in the drawing.
7. Hull elements, cables, cableways, equipment items, volumes, pipe and ventilation systems and assemblies can be modelled automatically, taking into account the whole available model information.
8. The behaviour of the Tribon application can be modified by providing the appropriate trigger functions. It is also possible to create and modify the application's menu and toolbars.

1.5 Duration

3 days

1.6 Using this document

Certain text styles are used to indicate special situations throughout this document; here is a summary;

All examples will be displayed as bold text in the **Courier New font**, and the program output will be indented to the right with respect to the user input. System prompts should be bold and italic in single quotes, i.e. **'Choose function'**.

Annotation for trainees benefit:



Additional information



Refer to other documentation

Larger examples and solutions to the exercises have not been included in the Training Guide, but can be found in the folder '**Vitesse Basic Training**' under **SB_PYTHON** of the training project. References to these examples are annotated with:



Refer to the training examples

2 Python Programming Language

The following introduction to the Python programming language is based on a document entitled 'Python Tutorial', written by Guido van Rossum and Fred L. Drake, Jr., the editor. Copyright (c) 2001-2003 Python Software Foundation

2.1 Introduction

Python is a simple, yet powerful programming language that bridges the gap between 'C' and Shell programming. Its syntax is put together from constructs borrowed from a variety of other languages; most prominent are influences from ABC, 'C', Modula-3 and Icon. It runs under several environments including many UNIX's, MS-Windows, OS/2 and Macintosh operating systems. It includes many features of modern programming languages together with many useful software components.

This course deals with the Tribon Vitesse programs and therefore Python teaching is limited to only relevant Python concepts and features. The following section contains information on the majority of basic Python commands that may be used when writing Tribon Vitesse programs.

2.2 Assignments and Expressions

2.2.1 Numbers

At its very simplest level, Python acts as a basic calculator. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages. Parentheses can be used for grouping.

```
2+2
4
# This is a comment
2+2 # this is a comment on the same line as code
4
(50-5*6)/4
5
```

- ① Anything entered on a line after a hash symbol (#) will be ignored by the program and can therefore be used as a comment to help others understand the program.

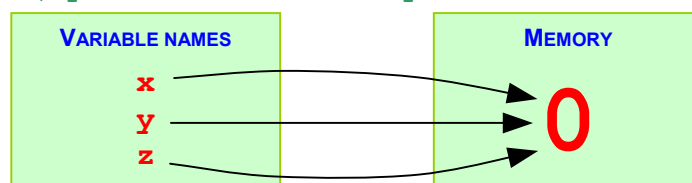
Like in 'C' the equals sign (=) is used to assign a value to a variable. When evaluating such an assignment Python does not display the assigned value:

```
width = 20
height = 5*9
width * height
900
```

- ① Python supports also the so-called **long integers**, recognised by the 'L' suffix. They can have arbitrarily large integer values (e.g. 123456789123456789L). Another numerical type available to Python are **complex numbers** (e.g. complex(1,2) → (1+2j))

A value can be assigned to several variables simultaneously:

```
x = y = z = 0 # Zero x, y and z simultaneously
x
0
y
0
z
0
```



- ❗ The assignment in Python should be understood as **binding** variables names to memory locations containing the expressions given on the right-hand side of the '=' operator. Thus, in the above example, we have a **single memory location** containing the value of ZERO, but **three variable names** ('x', 'y', and 'z') bound to it. This is a very important issue, because many other programming languages implement the assignment as **copying** of the value of the expression into the memory location identified by the given variable name. In that case, there would be **three** distinct memory locations, and **three** variable names.

Python language implements also the so-called augmented assignment operators:

| | | |
|----------|---|-----------|
| a += b | → | a = a + b |
| a -= b | → | a = a - b |
| a *= b | → | a = a * b |
| a /= b | → | a = a / b |
| ... etc. | | |

These new operators are not only nice, shorter equivalents of the assignments given on the right-hand side. They help generate a more efficient code, especially when the variable **a** is a member of some compound data structure.

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
4 * 2.5 / 3.3
3.0303030303
7.0 / 2
3.5
7 / 2
3
```



- ❗ The division operator '/' applied to the integer operands returns an integer, which is the result of rounding the exact result down to the nearest lower integer (the remainder is then always non-negative)

Apart from the typical mathematical operators: * (multiplication), / (division), + (addition), - (subtraction), Python supports additionally the following operators:

| Operator | Explanation | Examples |
|----------|-------------------|------------------------------------|
| // | Quotient | 13.0 // 3.0 → 4.0 13 // 3 → 4.0 |
| & | Bit-wise AND | 5 & 2 → 0 6 & 2 → 2 |
| | Bit-wise OR | 5 2 → 7 6 2 → 6 |
| ^ | Bit-wise XOR | 5 ^ 2 → 7 6 ^ 2 → 4 |
| % | Remainder | 13.0 % 3.0 → 1.0 13 % 3 → 1 |
| ** | Power | 2 ** 4 → 16 |
| << | Left shift | 2 << 3 → 16 |
| >> | Right shift | 16 >> 3 → 2 |
| ~ | Bit-wise negation | ~5 → -6 ~(-6) → 5 |

The logical and relational operators were not included in the above table, and will be explained in section 2.5.1.

Bit-wise operators are useful for managing various kinds of flags stored at the corresponding bit positions in an integer variable. As an example, let's analyse the **flags** variable, which stores the following settings concerning the creation of a symbolic view:

| Setting | Bit position | Value, if ON |
|---|--------------|------------------|
| Draw seams | 0 | 1 = '00000001' |
| Draw profiles | 1 | 2 = '00000010' |
| Automatic selection | 2 | 4 = '00000100' |
| Draw plane views | 3 | 8 = '00001000' |
| Draw RSO's | 4 | 16 = '00010000' |
| Draw as plate | 5 | 32 = '00100000' |
| Draw intersections | 6 | 64 = '01000000' |
| Design View (as opposed to Assembly View) | 7 | 128 = '10000000' |

So that **flags** variable with the value of 207 ('11001111') indicates, that the symbolic view should be drawn as a design view with automatic selection, and seams, profiles, plane views and intersections drawn. RSO's and Draw as plate is turned off. Then the following bit-wise operation are possible:

- flags = flags & ~4 – guarantees, that Automatic selection is turned OFF
- flags = flags | 16 – guarantees, that Draw RSO's is turned ON
- flags = flags ^ 32 – toggles the setting for Draw as plate (ON → OFF, and OFF → ON)
- if flags & 8: – verifies, if Draw plane views is turned ON

2.2.2 Strings

Beside numbers, Python can also manipulate strings, enclosed in single quotes or double quotes. Python allows to use one kind of enclosing quotes, while embedding safely the other kind of quotes in the string (examples below). Because of the use of single quotes in the Tribon Hull or Tribon Data Extraction commands, it is recommended that in the Python scripts the double quotes are used whenever possible to avoid confusion.

```
'tribon vitesse'
'tribon vitesse'
'doesn\'t'
"doesn't"
"doesn't"      # recommended arrangement
"doesn't"
'"yes", he said.'
'"yes", he said.'
"\\"yes\\", he said."
'"yes", he said.'
```

❶ *Note: by prefixing a single (') or double (") quote symbol with a backslash (\) it is possible to indicate that the symbol should be treated as-is and not as the character terminating a string. However, the use of the backslash can be minimised by the recommended use of the single and double quotes.*

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
word = "Str" + "ing"
word
'String'
"<" + word*5 + ">"
'<StringStringStringStringStringString>'
```

It is possible to check, if the string contains the given fragment using the **in** operator.

```
'rin' in word
True
'xxx' in word
False
```

❶ *The ability to perform this check for the multi-character fragments is new to Python 2.3. In previous versions it was possible to check the presence in a string of single-character fragments only. Python supports also the 'not in' operator, returning True, if the fragment **IS NOT** present in the string.*

Strings can be indexed like in 'C', and the first character of a string has index 0. There is no separate character type; a character is simply a string of size one. Like in Icon, substrings can be specified with the slice notation: two indices separated by a colon.

The notation **s[i:j]** returns a string containing the characters from the source string **s**, whose indices are between **i** and **j** (excluding **j**).

```
word[4]
'n'
word[0:4]
'Stri'
word[2:4]
'ri'
```

Slice indices have useful defaults: an omitted first index defaults to zero, omitted second index defaults to the size of the string being sliced.

```
word[:2] # The first two characters
'St'
word[2:] # All but the first two characters
'ring'
```

An index larger than the string size is replaced by the string size, an upper bound smaller than the lower bound causes an empty string to be returned.

```
word[1:100]
'tring'
word[10:]
''
word[2:1]
''
```

Indices may be negative numbers, to start counting from the right.

```
word[-1]    # The last character
'g'
word[-2]    # The last but one character
'n'
word[-2:]   # The last two characters
'ng'
word[: -2]  # All but the last two characters
'Stri'
```

The best way to remember how slices work is to think of the slices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

| | | | | | | | | | | | | |
|----|---|----|---|----|---|----|---|----|---|----|---|---|
| | s | | t | | r | | i | | n | | g | |
| | | | | | | | | | | | | |
| 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 |
| -6 | | -5 | | -4 | | -3 | | -2 | | -1 | | |

The first row of numbers above give the position of the positive indices 0 - 6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labelled *i* and *j* respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds, e.g. the length of `word[1:3]` is 2.

The built in function `len()` returns the length of a string:

```
s = "a very long string"
len(s)
18
```

① The negative character indices of a string can be easily mapped onto the normal, positive index values by adding the length of the whole string.

① Python 2.3 supports an extended slice notation: `s[i:j:k]`, where the third value (*k*) determines the step used, when going from *i* to *j* (default: 1). Examples for `s = 'equipment'`:

```
s[1:7:2] → 'qim'
s[::-1] → 'tnempiue'    (the reversed 'equipment' word)
```

2.2.3 % string operator

In order to construct pretty formatted strings (e.g. messages to the user, multicolumn report lines, etc.), Python provides the `%` string operator:

```
block = 'ES123'
weight = 980.5
⇒ "Hull block '%s' has the weight %0.3f kg" % (block, weight)
   "Hull block 'ES123' has the weight 980.500 kg"
```

The first argument to the `%` string operator is the format string, containing fixed text parts, and the format specifiers, which are placeholders for values taken from the tuple provided on the right-hand side of the `%` string operator.

Examples of format specifiers:

- `%s, %d` - a string (s) or integer (d) value, occupying as many character places, as there are characters required to represent the value
- `%5s, %5d` - a string (s) or integer (d) value occupying **at minimum** 5 character places. The string is **right-justified** in the output text field, and remaining character positions are filled with BLANKS.
- `%-5s, %-5d` - a string (s) or integer (d) value occupying **at minimum** 5 character places. The string is **left-justified** in the output text field, and remaining character positions are filled with BLANKS.
- `%8.1f, %-8.1f` - a floating point number (f) occupying **at minimum** 8 character places with 1 digit after decimal point. The string is **right-justified** (positive width) or **left-justified** (negative width) in the output text field, and remaining character positions are filled with BLANKS.
- `%08d, %08.1f` - an integer (d) or floating point (f) value occupying **at minimum** 8 character places (with 1 digit after decimal point for the floating point number). The string is **right-justified** in the output text field, and remaining character positions are filled with ZEROES

The example below prints out an example line of the report about the weight of the hull blocks on a project.

```
print "| %03d. | %-12s| %10.3f|" % (n, block, weight)
      | 011. | ES123      | 980.500|
```

2.2.4 Converting non-strings into strings

There are situations, when a numeric value has to be used in a string expression. This happens rather often, when it comes to a data transfer between the Tribon system and the Python language.

An example of this would be if the result of a transfer were given as an **integer**, while the Tribon Data Extraction statement requires a **string**. In cases such as this, it is possible to convert the integer (or virtually any other expression) to a string using the following statement:

```
reqd_string = str(integer to convert)
```

or

```
reqd_string = repr(integer to convert)
```

Example of this type of conversion can be seen in Vitesse scripts using Data Extraction, although the more complex conversions are better expressed using the % string operator.

i The **str()** and **repr()** functions can be used to obtain the string representation of any argument, not only of an integer. Typically, their results are the same, but for arguments belonging to some user-defined type (classes), these functions can be redefined to provide different string representations. In such cases, **repr()** should provide an 'official' string representation (used internally by Python interpreter), and **str()** is allowed to return an alternative.

2.2.5 String Methods


Since a long time Python provided the standard **string** module for more advanced string manipulation. Since Python 2.0, the string variables are given useful methods, being the equivalents of the **string** module functions. Although the **string** module is still supported, using the string methods is preferred. For any **string** module function, the corresponding string method can be derived using the rule given below (**s** is a string variable):

```
string.<method>(s, arg1, arg2, ...) → s.<method>(arg1, arg2, ...)
```

Some examples are given below (square brackets indicate optional arguments):

Old syntax (string module)

New, preferred syntax (string methods)

| | | |
|---|--|---|
| <code>string.upper(s)</code> | → <code>s.upper()</code> |  |
| <code>string.lower(s)</code> | → <code>s.lower()</code> | |
| <code>string.center(s, width)</code> | → <code>s.center(width)</code> | |
| <code>string.split(s [,separator [,maxsplit]])</code> | → <code>s.split([separator [, maxsplit]])</code> | |
| <code>string.find(s, sub [,start [,end]])</code> | → <code>s.find(sub [, start [, end]])</code> | |
| <code>string.join(list [,separator])</code> | → <code>separator.join(list)</code> | |

2.3 Data Structures (Lists, Tuples & Dictionaries)

2.3.1 Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
a = ["plane", "panel", 100, 1234]
a
['plane', 'panel', 100, 1234]
```

Like the string indices, list indices start at zero, and lists can be sliced, glued together and so on:

```
a[0]
'plane'
a[3]
1234
a[-2]
100
a[1:-1]
['panel', 100]
a[:2] + ["plate", 4]
['plane', 'panel', 'plate', 4]
2*a[:3] + ["deck 2"]
['plane', 'panel', 100, 'plane', 'panel', 100, 'deck 2']
```

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```
a[2] = a[2] + 23
a
['plane', 'panel', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list:

```
a[0:2] = [1, 12] # Replace some items
a
[1, 12, 123, 1234]
a[0:2] = []      # Remove some items by assigning an empty list
a
[123, 1234]
a[1:1] = ["pipe", "KCS-WW12"] # Insert some items
a
[123, 'pipe', 'KCS-WW12', 1234]
a[:0] = a # Insert a copy of the list itself at the beginning
a
[123, 'pipe', 'KCS-WW12', 1234, 123, 'pipe', 'KCS-WW12', 1234]
```

The built in function `len()` also applies to lists and returns the count of its elements:

```
len(a)
8
```


It is possible to nest lists (create lists containing other lists), to simulate multidimensional arrays. For example:

```
q = [2, 3]
p = [1, q, 4]
len(p)
3
p[1]
[2, 3]
p[1][0]
2
```

```

p[1].append("extra") # See below for 'append'
p
[1, [2, 3, 'extra'], 4]
q
[2, 3, 'extra']

```



① Note that in the above example the modification of **p[1]** has the same consequences as the direct modification of the list **q** (third element 'extra' is appended). It means, that both **q** and **p[1]** refer to the same object in memory, not to the distinct copies. In other words, nesting the list **q** in the list **p** includes the list **q** itself in the list **p**, not its copy.

The list data type has some important methods. One of them (append) has been already used in the example above. Here are all the methods of list objects:

| | |
|-------------------------|---|
| insert(i, x) | <p>Inserts an item x at the position i, moving the elements at position i and above up by one position. For example:</p> <p>a.insert(0, x) inserts the element x at the front of the list a a.insert(len(a), x) appends the element x at the end of the list a</p> |
| append(x) | a.append(x) is equivalent to a.insert(len(a), x) (see above) |
| index(x) | Returns the index in the list of the first occurrence of x . It is an error if there is no such item. Then an exception (ValueError) is raised, which can be caught by the program and properly handled. |
| remove(x) | Removes the first occurrence of x from the list. It is an error if there is no such item. Then an exception (ValueError) is raised, which can be caught by the program and properly handled. |
| sort([comp_fun]) | <p>Sorts the items of the list, in place. The <u>optional</u> comp_fun argument is a function determining the order of two list elements. This function should be defined so, that:</p> $\text{comp_fun}(x, y) = \begin{cases} -1 & \text{for } x < y, \\ 0 & \text{for } x = y, \\ 1 & \text{for } x > y \end{cases}$ <p>If the comp_fun argument is omitted, the list is sorted in an ascending order.</p> |
| reverse() | <p>Reverses the elements of the list, in place.</p> <p>NOTE: In order to sort the list in the descending order, it is usually faster to call sort() followed by reverse(), instead of calling sort() with a comp_fun argument taking care of the reversed sorting order.</p> |
| count(x) | Returns the number of times x appears in the list. |
| extend(list2) | Appends all the elements of list2 to the end of the current list |
| pop([index]) | Removes the list item at the position given by the <u>optional</u> argument index (default last), and returns it |

① All identifiers in a Python program are case-sensitive. Thus, for **mylist** being a Python list, **mylist.Append(x)** will not be recognised, but **mylist.append(x)** will.

Here is an example of all the above mentioned list methods:

```

a = ["ABC", 20, 1, "ABC", "abc"]
print a.count("ABC"), a.count(1), a.count("xyz")
2 1 0
a.insert(2, -1)
a.append(20)
a
['ABC', 20, -1, 1, 'ABC', 'abc', 20]
a.index(20)
1
a.remove(20)
a
['ABC', -1, 1, 'ABC', 'abc', 20]

```

```

a.reverse()
a
[20, 'abc', 'ABC', 1, -1, 'ABC']
a.sort()
a
[-1, 1, 20, 'ABC', 'ABC', 'abc']
a.pop(2)
20
a #See, that the item 20 has been removed!
[-1, 1, 'ABC', 'ABC', 'abc']
a.extend(['A', 'B', 'C'])
[-1, 1, 'ABC', 'ABC', 'abc', 'A', 'B', 'C']

```

It is possible to check quickly, if the given element exist in the list using the `in` operator. The result is the Boolean value **True** or **False**. This is faster, than using the list's `count` method!

```

a = [12, 100, 3, 100]
100 in a      # faster method
True
a.count(100) # non-zero (true), slower method
2
200 in a      # faster method
False
a.count(200) # zero (false), slower method
0

```

① Python 2.3 supports an extended slice notation: `a[i:j:k]`, where the third value determines the step used, when going from 'i' to 'j' (default value: 1). Examples for `a = [1, 2, 3, 'A', 'B', 'C', 'D', 'E']`:

```

a[1:7:2] → [2, 'A', 'C']
a[::-1] → ['E', 'D', 'C', 'B', 'A', 3, 2, 1]    similar to a.reverse(), but returns a separate list

```

2.3.2 Tuples

Lists belong to the mutable objects, i.e. you can modify its definition 'in place', add new elements, remove existing ones, modify the elements, etc. We can say that a tuple is the immutable version of the list. You can define a tuple, use indices, and make slices. You cannot modify, however, a previously defined tuple, unless you redefine it from scratch.

The tuple is defined using parentheses (square brackets are reserved for lists).

```

a = (10, 12, 'x')
a
(10, 12, 'x')
a[1]
12
a[1:]
(12, 'x')
a[2:]
('x', )
a = (5, 10) # redefinition of a tuple
a
(5, 10)

```

① Please note the syntax for a tuple containing a single element ('x',). This comma character is required in order to distinguish the single-element tuple from the expression in parentheses used for grouping.

Python accepts the multiple assignment syntax, where the right-hand side of the assignment is a tuple or a list. This is sometimes called the '*unpacking*' of the tuple:

```

a = (10, 12, 'x')
a
(10, 12, 'x')
height, width, mark = a

```

```

height
10
width
12
mark
'x'

```

2.3.3 Dictionaries

The dictionary is the compound data type, containing the pairs of values. The first value is the key and the second – the value attached to this key. The keys of the elements in the dictionary are unique. The dictionary itself is defined using curly brackets:

```

a = {'dogs':3, 'cats':1, 'mice':2}
a
{'dogs': 3, 'cats': 1, 'mice': 2}

```

You can refer to the value attached to the given key exactly in the same way, as if the key were the index of some 'list'. Using this facility you can modify the value under the given key, add new key/value pairs and delete the existing pairs:

```

a['cats']
1
a['mice']=5 # change of the value for 'mice'
a
{'dogs': 3, 'cats': 1, 'mice': 5}
a['horses']=1 # adding of the new element
a
{'dogs': 3, 'cats': 1, 'mice': 5, 'horses': 1}
del a['horses'] # removing the element
a
{'dogs': 3, 'cats': 1, 'mice': 5}

```

It is an error to retrieve a value from the key not existing in the dictionary. This situation is an exception **KeyError**, which can be detected by the program and properly handled.

The dictionary objects have some useful methods facilitating the handling of their contents:

| | |
|-------------------------------|---|
| keys() | Returns an unsorted list of the dictionary keys |
| values() | Returns an unsorted list of the dictionary values |
| has_key(x) | Returns 1 if the dictionary contains the key x , 0 otherwise. Can be used to check for existence of the given key without raising an exception, if the key does not exist. |
| items() | Returns a list of two-element tuples containing the key and its value. |
| clear() | Removes all items from the dictionary. |
| copy() | Returns a shallow copy of the dictionary, which is a dictionary object independent from the original, which may, however, contain references to the same nested objects, as the original dictionary. |
| update(dict) | Updates the current dictionary with the contents from dictionary dict . New key/value pairs are added, values at the existing keys are overwritten. |
| get(key, [def]) | Returns the item at the given key , if the dictionary contains it. Otherwise, returns the <u>optional</u> value def (default None). |
| setdefault(key, [def]) | Like get() , but additionally, the pair key: def is added to the dictionary, if the given key does not exist in the dictionary |
| fromkeys(seq, [value]) | Returns a new dictionary with keys taken from the sequence seq and values all set to value (default None). Items occurring in seq more than once appear in the resulting dictionary only once. |
| iteritems() | Returns a dictionary iterator, suitable for looping over key:value pairs of the dictionary |

| | |
|------------------------------|--|
| <code>iterkeys()</code> | Returns a dictionary iterator suitable for looping over the dictionary keys() |
| <code>itervalues()</code> | Returns a dictionary iterator suitable for looping over the dictionary values |
| <code>pop(key, [def])</code> | Removes the item with the given key from the dictionary, and returns the value associated with the key . If key is not found, def is returned (if given), or the <code>KeyError</code> exception is raised |
| <code>popitem()</code> | Removes an item from the dictionary and returns it as a tuple (key, value). If the dictionary is empty, <code>KeyError</code> exception is raised |

① Although we have no control over the order of items in the dictionary, we can be sure, that the order of items in the lists returned by the **keys()** and **values()** functions is consistent. Therefore, it is safe to fetch the key and the value from these lists from the same index – they form together the key/value pair present in the original dictionary. You may consider using **items()** method instead, which returns a list of **(key, value)** tuples.

For more details about the dictionary iterators, see section 2.6.1

① Python 2.3 introduces an alternative form of the test of the existence of the given **key** in the dictionary **dict**: **key in dict** is equivalent to **dict.has_key(key)**

① The **fromkeys()** method can help you clean a list from duplicated items.

Example for `aList = [1, 3, 'A', 3, 'B', 'A', 2]`:

```
temp = dict.fromkeys(aList) #Temporary dictionary
aList = temp.keys()         #Cleaned-up list
```

which produces a list `['A', 1, 'B', 3, 2]` quickly and efficiently. The disadvantage: you loose the original ordering of elements of your list.

2.4 First Steps to Programming

Obviously Python is capable of much more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
# Fibonacci series:
# The sum of two elements defines the next
a, b = 0, 1
while b < 10:
    print b
    a, b = b, a+b
1
1
2
3
5
8
```

This example introduces several new features ...

2.4.1 Multiple Assignments

In the first line the variables **a** and **b** simultaneously get the new values **0** and **1**, respectively. In the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first, then the assignments take place.

2.4.2 WHILE Loops

The **while** loop executes as long as the condition (here **b < 10**) remains true. In Python, like in 'C', any non-zero integer value is true; zero is false. The standard comparison operators are written the same as in 'C': **<**, **>**, **==**, **<=**, **>=** and **!=**.

2.4.3 Loop Indentation


Python uses the actual indentation of the body's code to detect a statement's start and finish. The amount of indentation used is not important, as long as it is consistent for all statements in a block. It can consist of any number of tabs or blanks.

- ❗ *The indentation is an essential feature of the Python syntax. An inconsistent indenting of the source code may produce syntax errors, or change the statement execution flow, causing the program to behave differently!*

Example:

*If we do not indent the last line from the previous example, the program will enter an infinite loop! The value of 1 will be constantly printed, because the variable **b** will not change (the statement changing it is now out of the scope of the loop)! Such a program can be stopped only by the workstation's operating system.*

```
# Fibonacci series:
# The sum of two elements defines the next
a, b = 0, 1
while b < 10:
    print b
⇒ a, b = b, a+b # wrong indentation
    1
    1
    ... infinitely
```



2.4.4 Print Statement

The **print** statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to print (as seen in the earlier examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so that you can format things nicely, like this:

```
i = 256*256
⇒ print "The value of i is", i
    The value of i is 65536
```

A trailing comma avoids the new line after the output:

```
a, b = 0, 1
while b < 1000:
⇒ print b, # Note the comma at the end!
    a, b = b, a+b
    1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

2.5 The IF Statement

Besides the **while** statement, Python recognises the usual control flow statements such as **if**. For example:

```
x = -1
if x < 0:
    x = 0
    print "Negative changed to zero"
elif x == 0:
    print "zero"
elif x == 1:
    print "single"
else:
    print "more"
Negative changed to zero
```

- ❗ *Note the difference between the assignment operator '=', and a comparison operator '=='!*

There can be zero or more **elif** parts, and the **else** part is optional. The keyword **elif** is short for **else if**, and is useful to avoid excessive indentation. An **if ... elif ... elif ...** sequence is a substitute for the **switch** or **case** statements found in other languages.

2.5.1 Conditions

Both **while** and **if** statements use conditions, which are expressions involving the following relational operators:

| Operator | Definition | Operator | Definition |
|--------------------|------------------------------|---------------------|--------------------------------------|
| <code>==</code> | equal | <code>!=</code> | not equal |
| <code>></code> | greater than | <code><</code> | less than |
| <code>>=</code> | greater than or equal | <code><=</code> | less than or equal |
| <code>in</code> | is a member (is a substring) | <code>not in</code> | is not a member (is not a substring) |
| <code>is</code> | is identical to | <code>not is</code> | is not identical to |

In Python 2.3 these operators produce a Boolean values: **True** or **False**, whereas in previous versions they returned an integer: either **1** or **0**. Every Python object, in fact, has a truth value and can be used in the **while** and **if** statements instead of the Boolean expression. The following values are considered false:

- `None`,
- **False**,
- a numerical zero (e.g. `0`, `0L`, `0.0`, `0j`),
- an empty sequence (e.g. an empty string, list or tuple),
- an empty mapping (e.g. an empty dictionary),
- instances of classes, that define a `__nonzero__()` or `__len__()` method, when they return the integer zero or Boolean value **False**.

All other values are considered **True**. Expression using relational operators can be combined using logical operators: **and**, **or**, **not**:

| Operator | Example | Explanation |
|------------------|----------------------|--|
| <code>and</code> | <code>x and y</code> | <code>x</code> is evaluated. if <code>x</code> is false, it is returned, otherwise <code>y</code> is evaluated and returned. |
| <code>or</code> | <code>x or y</code> | <code>x</code> is evaluated. if <code>x</code> is true, it is returned, otherwise <code>y</code> is evaluated and returned. |
| <code>not</code> | <code>not x</code> | <code>x</code> is evaluated. If <code>x</code> is false, True is returned, otherwise False |

For example, the statement:

```
s = name or "ABC" #assignment with a default value
```

will assign **name** to **s**, unless **name** is an empty string – then **s** will be assigned the default value of "ABC".

Logical operators have lower precedence than the relational operators, so it is safe to write the following expression without parentheses:

```
x < 10 and x > 0
```

which can be written in a yet simpler form:

```
0 < x < 10
```

Among the logical operators, **not** is evaluated before **and**, which is evaluated before **or**. In order to obtain a different evaluation order, use parentheses:

```
x > 0 and (y < 0 or y > 10)
```

2.6 The FOR Statement

Python's **for** statement basically iterates over the items of any sequence (e.g. a list, tuple or string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
a = ["cat", "window", "defenestrate"]
⇒ for x in a:
    print x, len(x)
cat 3
```

```
window 6
defenestrate 12
```

If you need to modify the list you are iterating, e.g. duplicate or remove selected items, you must iterate over a copy. The slice notation makes this particularly convenient:

```
⇒ for x in a[:]: # slice copy the entire list
    if len(x) > 6:
        a.insert(0, x)
a
["defenestrate", "cat", "window", "defenestrate"]
```



2.6.1 Iterators

In Python 2.3, the **for** loop is able to loop not only over a sequence, but also using an iterator. An iterator is a Python object having the following two properties:

- It has a **next()** method, providing the next value for the loop,
- When there is no next value to provide, the iterator raises the **StopIteration** exception.

An iterable object is an object able to produce iterators looping over the object's contents. For example, the dictionary is now an iterable object, by providing the following three iterators: **iteritems()** – to loop over the (key, value) pairs, **iterkeys()** – to loop over the keys, and **itervalues()** – to loop over the values. They can be used as follows (**dict** is an example dictionary):

```
for key, value in dict.iteritems():
    print key, value

for key in dict.iterkeys():
    print key

for value in dict.itervalues():
    print value
```

The default iterator for a dictionary is **iterkeys()**, so it is possible to write:

```
for key in dict:
    print key
```

The file objects are now their own iterators. In order to loop over the lines of a text file, we can write:

```
f = file("C:\\myfile.txt", "r")
⇒ for line in f:
    print line
f.close()
```

Another type of an iterator is a generator function. We will study them in details in section 2.10.2.

The conclusion is, that in Python 2.3 the **for** loop has been redesigned to loop not over a sequence, but over any iterable object. This includes, among other things, the lists, tuples, strings, dictionaries, and text files.



This behaviour can be extended also to user-defined classes. Readers interested in creating their own iterators should read at least the section 9.9 of the Python Tutorial.

2.7 The RANGE() Function

The built-in function **range()** is used if you need to iterate over a sequence of numbers. It generates lists containing arithmetic progressions of integer values, e.g.:

```
range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, exactly the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative):

```
range(5, 10)
[5, 6, 7, 8, 9]
range(0, 10, 3)
[0, 3, 6, 9]
range(10, 0, -3)
[10, 7, 4, 1]
```

To iterate over the indices of a sequence, combine `range()` and `len()` as follows:

```
a = ["Mary", "had", "a", "little", "lamb"]
⇒ for i in range(len(a)):
    print i, a[i]
0 Mary
1 had
2 a
3 little
4 lamb
```

- ① The `range()` function is able to generate lists consisting of integer numbers only! Its floating point arguments (if any) will be converted to integer numbers before processing.
- ① Python 2.3 introduces the function `enumerate()`, which facilitates creation of loops using both the element and its index in a list. Below you can see an equivalent code of the last example. The output is exactly the same.

```
a = ["Mary", "had", "a", "little", "lamb"]
for ind, word in enumerate(a):
    print ind, word
```

2.8 List comprehension

The `range()` function creates a list consisting of the terms of the arithmetic progression. Python language provides the list constructing syntax, called the list comprehension, that could be used to create lists following some other construction schemes:

```
list = range(5)
list
[0, 1, 2, 3, 4]
[x*x for x in list]           #list of squares
[0, 1, 4, 9, 16]
[x*x for x in list if x > 1]  #restricted list of squares
[4, 9, 16]
rows, cols = ['1', '2', '3'], ['A', 'B', 'C']
# some chessboard field names
fields = [col + row for col in cols for row in rows]
fields
['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']
[c + r for c in cols for r in rows if r < '2' and c > 'A']
['B1', 'C1']
```

The `for` part introduces the iteration over the elements of some source list, whereas the `if` part (optional) introduces the filter, restricting the iteration to a subset of the source list elements. The output list, consists of the results of evaluating the expression (given just after the opening square bracket) for all values of the variables, as specified in the `for` clauses, meeting the filter criteria possibly provided in the `if` clause. Let us compare now this new syntax to the 'old' style, using `for` loops and `if` conditional statements. This is the last example, rewritten in the 'old' style:

```
fields = []
for c in cols:                #first iteration
    for r in rows:            #second iteration
        if r < '2' and c > 'A': #filter - restriction
            fields.append(c + r) #list construction
```

```
fields
['B1', 'C1']
```

By using the list comprehension syntax, we can save a few lines of code, and additionally, make the code much clearer, by decreasing the need of an excessive indentation. Its power comes from the ability to iterate using more than a single variable, and from the possibility to include arbitrarily complex expressions and filter conditions, including user-defined functions.



Python language provided since a long time additional list manipulating functions. `map()`, `filter()`, and `zip()` usually can now be replaced by list comprehension. `reduce()` can be used to calculate a single result out of a list of arguments. Details can be found in Python Library Reference manual.

2.9 BREAK and CONTINUE

Both the **while** and **for** loop statements have means for defining, how long the loop should be executed. Sometimes, in the middle of the loop's body, we find out, that for some reasons the execution of the loop should be aborted. Python supports these situations by providing the **break** statement. Examples:

| | | |
|---|---|---|
| <pre>def f1(start, end): a = start while a < end: print a, if a >= 5: break a = a + 1 else: print "OK, end was below 6" print "THE END"</pre> <p>⇒</p> <pre>f1(3, 5) 3 4 OK, end was below 6 THE END f1(3, 7) 3 4 5 THE END</pre> | ⇒ | <pre>def f2(start, end): for a in range(start, end): print a, if a > 5: break else: print "OK, end was below 6" print "THE END"</pre> <pre>f2(3, 5) 3 4 OK, end was below 6 THE END f2(3, 7) 3 4 5 THE END</pre> |
|---|---|---|

In the second case, when the variable **a** reached the value of 6, the loop's execution was terminated by the **break** statement. Please notice also the use of an optional **else:** clause to the **while** and **for** statements. The statements, it contains will be executed after termination of the loop, **BUT NOT** if the loop was terminated by the **break** statement. This can help us distinguish between normal and premature termination of a loop.



*The **break** statement allows us to construct the loops of the form presented below, where the condition in the **while** loop is always **True**, creating an apparently infinite loop:*

```
while True:
    ...
    if some_condition:
        break
```

*We use such loops in situations, when it is difficult to define a concise looping condition at the start. During the execution of the loop's body various conditions are examined, and at some points the loop decides to terminate the loop, by calling the **break** statement.*

The **continue** statement is used inside the loop's body in order to request an immediate interruption of the current iteration and passing to the next iteration, if available. Examples:

| | | |
|---|---|--|
| <pre>def f1(start, end): a = start while a < end: a = a + 1 if 3 <= a <= 5: continue print a,</pre> <p>⇒</p> | ⇒ | <pre>def f2(start, end): for a in range(start, end): if 3 <= a <= 5: continue print a,</pre> |
|---|---|--|

```
f1(0, 7)
1 2 6 7
```

```
f2(0, 7)
0 1 2 6
```

In the above examples some of the integers were not printed out, because the **continue** statement requested the return to the beginning of the loop, so that the **print** statement was not reached.

- ❶ The **continue** statement must not be used inside the **try:** block of the **try: ... except: ...** statement (see section 2.12).

2.10 Function definitions

We can be even more flexible by writing the Fibonacci series example (section 2.4) as a function:

```
⇒ def Fibonacci(max_b):
    a, b = 0, 1
    while b < max_b: # search up to max_b
        print b,      # Note the comma at the end!
        a, b = b, a+b
Fibonacci(10)
1 1 2 3 5 8
Fibonacci(25)
1 1 2 3 5 8 13 21
```

- ❶ Python recognises the end of the function definition as the place where the source text's indentation level becomes the same as of the corresponding **def** statement. Thus the line **Fibonacci(10)** no longer belongs to the function definition and is normally executed, printing the expected results.

Using the above function it is possible to get the Fibonacci series printed to an arbitrary maximum value just by specifying it as the function's parameter within parentheses. You can also write your own functions returning result:

```
# Compute the volume of the cone
def volume(radius, height):
    pi = 3.1415926535 # the pi constant
    # determine the function's result
⇒ return pi*radius*radius*height/3.0

r, h = 5.0, 10.0 # Example cone dimensions
print "The cone has basis radius r =", r, \
    "and height h =", h
    The cone has basis radius r = 5.0 and height h = 10.0
print "The volume is V =", volume(r, h)
    The volume is V = 261.799387792
```

- ❶ The **return** statement determines the function's result.
- The backslash character **** used at the end of the line indicates, that the current source line is extended on the next one. The indentation of the continuation line is not important, as only the indentation level of the first line is taken into account.

2.10.1 Function's arguments

Let's discuss the function header with the following formal arguments:

```
def fun(x, y=0, side=1, *posArgs, **keyArgs):
```

and some examples of how it can be called with explanations, how the passed values are assigned to the function's formal arguments:

```
1. res1 = fun(1)
2. res2 = fun(1, 1000, -1)
3. res3 = fun(1, side = -1)
4. res4 = fun(1, 1000, -1, "EXAMPLE", 120.5)
5. res5 = fun(1, 1000, -1, "EXAMPLE", flags=0, name="DECK")
```

- 1) x = 1, y = 0 (default), side = 1 (default), posArgs = () (an empty tuple), keyArgs = {} (an empty dictionary)

- 2) `x = 1, y = 1000, side = -1, posArgs = (), keyArgs = {}`
- 3) `x = 1, y = 0 (default), side = -1, posArgs = (), keyArgs = {}`
- 4) `x = 1, y = 1000, side = -1, posArgs = ("EXAMPLE", 120.5), keyArgs = {}`
- 5) `x = 1, y = 1000, side = -1, posArgs = ("EXAMPLE",), keyArgs = {flags: 0, name: "DECK"}`

Looking at the above examples, we can see the following kinds of formal arguments (specified in the function's header):

- mandatory arguments – specified as the argument name **WITHOUT** the default value,
- optional arguments – specified as the argument name **WITH** the default value,
- special, optional arguments, whose name starts with a **SINGLE** or **DOUBLE** asterisks (`*posArgs`, `**keyArgs`)
 - an argument with a **SINGLE** asterisk, if present, it is a tuple receiving all additional positional actual arguments, that could not be assigned to the formal non-special arguments. There can be only one such argument, and must be specified as the **LAST** formal argument.
 - an argument with a **DOUBLE** asterisk, if present, it is a dictionary receiving all additional keyword actual arguments, with argument names not found in the formal argument list. There can be only one such argument, and must be specified as the **LAST** formal argument. If both special arguments are present, first comes the one with a **SINGLE** asterisk, next – the one with a **DOUBLE** asterisk prefix.

On the other hand, we have the following kinds of actual arguments (provided, when the function is called):

- positional (non-keyword) arguments – just values, without the argument name,
- keyword arguments – specified as `'name = value'`, where name is the name of the formal argument. Keyword arguments can be specified only **AFTER** the positional arguments.

Summing up, Python observes the following rules of assignment of actual arguments to the function's formal arguments, when the function is called:

- a positional actual argument is assigned to the next non-special formal argument, if available. If there is no next non-special formal argument, it is added to the tuple of additional positional arguments, if the function's header provides a special argument with a leading **SINGLE** asterisk. Otherwise, a syntax exception is raised.
- a keyword argument is assigned to the formal argument with the same name, if it exists in the function's header, or else it is added to the dictionary of additional keyword arguments, if the function's header provides a special argument with a leading **DOUBLE** asterisk. Otherwise, a syntax exception is raised.

Of course, the above example demonstrates the most complex situation, when the function accepts both positional and keyword arguments, mandatory and optional. Usually, we will have to deal with the simpler cases, like:

```
def Pre(*args):
def fun(x, *args):
def create(x, y, **options):
```

2.10.2 Generators

A generator is a function that generates the next result each time it is called. In Python 2.3 generators are standard functions that use the **yield** statement instead of **return** in order to define the function's value. When the **yield** statement is executed, before the generator returns the next result, the whole state of the function is "frozen". When the generator is called again, it resumes its execution **exactly** from the same place, where it was suspended by the last **yield** statement. Example:

```
def genPos(start, end, step):
    pos = start
    while pos < end:
        ⇒ yield pos
        pos = pos + step
```

The above generator simulates the **range()** function, but it does not have the limitations of the original **range()** function, and can accept floating point values as parameters. The functionality of the **for** loop has been extended to handle iterable objects (see section 2.6.1), in addition to the sequences (lists, tuples, strings). So we can write:

```

for x in genPos(0.0, 1.0, 0.2):
    print x,
0.0 0.2 0.4 0.6 0.8

```

❗ Please note, that it is not possible to use `range(0.0, 1.0, 0.2)` instead of our generator.

2.11 File handling

Python language has a built-in support for file handling. The process of handling a file can be described as a sequence of three stages:

1. Create the file object, using the `file()` function (the function `open()` is still accepted in Python 2.3, and is an alias for `file()`).
2. If successful, use the file object's methods, reading and writing the file.
3. Finally, close the file using the file object's `close()` method

❗ Closing the file guarantees, that the write buffer is flushed, and the allocated resources are properly released. That's why we recommend to enclose any file activity (stage 2) in `try... finally...` with the call to `close()` method in the `finally` section.

```
f = file("C:\\input.dat", "rb", 4096)
```

The above statement creates a file object, referring to the binary (b) file `C:\\input.dat` opened for reading (r) with the buffer size of 4kB. The second argument is a string defining the file opening mode:

- 'r' – open the file for reading (text)
- 'w' – open the file for writing (text)
- 'a' – open the file for appending (text)
- 'r+', 'w+', 'a+' open the file for updating (w+ truncates the file)
- 'rb', 'wb', 'ab' open the file in binary (non-text) mode
- If omitted, opening mode defaults to 'r'

The third argument is an integer number, defining the file's buffer size:

- 0 – file is unbuffered,
- 1 – file is line buffered,
- > 1 – approximate file buffer size,
- < 0 – use system default for file buffer size (default)

What file object's methods can be used at stage 2?

| | |
|------------------------|--|
| read(size) | – reads at most size bytes from the file, and returns the data read as a string. |
| read() | – reads ALL data from the file. |
| readline() | – reads a single line from the text file as a string |
| readlines() | – reads ALL lines from the text file as the list of strings |
| write(str) | – writes the string str to the file |
| writelines(lst) | – writes the lines from the list lst to the file |
| flush() | – flush the internal file buffer |
| tell() | – current position within file |
| seek(pos, from) | – sets the current position within the file to pos counting from from . Allowed from values: 0 – from the file's beginning 1 – from the current position 2 – from the file's end When the seek() method is used on a file opened in the 'a' or 'a+' mode, it will be undone at the next write to the file. |
| truncate(size) | – truncates the file size to size (default – current position) |

❗ In Python 2.3 we recommend to use the idiom `for line in f` to iterate over the lines of the file referred to by the `f` variable.

2.12 Exceptions: How to handle them?

The Python code operates on data coming from various sources. Very often the input data result from the user interaction or come from external files or databases. If the input data do not fulfil all the requirements of the program, an exception is possibly raised, which usually aborts the program's execution. It is undesirable to let the program terminate abnormally, without closing the open resources, or having a chance to handle the error.

The exception handling mechanism of Python is a good way of dealing with exceptions:

```
try:
    ... # Statements, that may raise an exception
except:
    ... # Exception handling code, which gets executed,
        # if an exception has been raised
else:
    ... # Optional, contains the statements executed,
        # if NO exception occurred
```

and

```
try:
    ... # Statements, that may raise an exception
finally:
    ... # Clean-up code executed no matter
        # if an exception occurred, or not
```

The `except` keyword may be followed by an optional identifier denoting the expected type of exception (see example 1 below). This means that the code indented under this keyword describes the program's reaction to that type of exceptions only. There can be several `except` clauses in the `try ... except ...` statement with different exception types. The `except` keyword without the exception type catches all types of exceptions.

If the raised exception matches one of the specified `except` clauses, its code is executed, and the exception is considered handled. Then the program continues, skipping the remaining `except` clauses, and a possibly existing `else` block. If no `except` clause matches the exception, it propagates to the surrounding code blocks until it is handled at a higher level, or until it reaches the Python interpreter, which then terminates the program. If the `try` block does not raise an exception at all, the optional `else` block is executed, and the program continues.

❗ The `try ... finally ...` construction does not handle the exception! It only guarantees that the statements given in the `finally` clause WILL be executed, even if the exception is raised. For handling this exception, one more surrounding `try ... except ...` construction is needed.

The examples below demonstrate typical situations, where the Python exception handling abilities are used:

1. We fetch a value from the dictionary `dict` from the given `key`, and if the key does not exist, we would like to have the value of `0` to be returned.

```
try:
    value = dict[key]
except KeyError:
    value = 0      # The given key does not exist!
```

❗ In this simple example, the statement `value = dict.get(key, 0)` would have the same effect.

2. We want to know the index of the item `x` in the list `a`. It is, however, possible, that the item `x` does not exist in the list `a`, so ...

```
try:
    ind = a.index(x)
except:
    ind = -1 # Item x not found!
```

3. We have a string parameter **s**, which should contain an integer. What if it does not ... ?

```
try:
    n = int(s)
except:
    print "s is not an integer!"
```

- ① **int()** is a built-in function trying to convert its argument to an integer. It will raise an exception, if the conversion is not possible. Python provides also the **float()** function, converting the argument to the floating point number.

4. A little more elaborated example showing the basics of reading data from external text files with error checking. It is a good example of the **try ... finally ...** construction.

```
try:
    f = file("TEST.DAT","r") # Open the file 'TEST.DAT' for reading
    try:
        for s in f:          # Loop over the lines of the file
            ...              # Work on data
    finally:                 # Even if an exception has been raised
        f.close()           # ... close the file
except:
    print "Error reading the file!" # Report an error
```

5. We process data from the list **a** until a non-integer element is found. In any case we want to print the resulting list **b** of processed data

```
b = []                      # Initialise resulting list
try:
    try:
        for element in a: # Loop over all elements in the list a
            b.append(int(element)+1) # Process the list, int() call may fail!
    finally:
        print b             # Print the resulting list
except:
    pass                    # Do nothing = ignore the exception
```

In the last example, an exception will be raised, if **element** could not be interpreted as an integer (**int()** function call fails). Even if this happens, the resulting list (as constructed so far) is printed on standard output. The last example shows also a statement **pass**, which does nothing, but can be used, when the compound statement syntax requires a statement, but it is in fact not needed.

2.12.1 Generator exception

Apart from using the generators in the **for** loop, as shown in section 2.10.2, we can use the generators directly. By calling:

```
gen = genPos(0.0, 1.0, 0.2)
```

we obtain the generator object. Now we can use the **next()** method to generate consecutive values:

```
val1 = gen.next()
val2 = gen.next() # and so on ...
```

but after returning the last value, the next call to **gen.next()** will raise a **StopIteration** exception. If we want to go through the generated values ourselves, we must use **try: ... except: ...** statement to avoid this exception.

```
while True:
    try:
        => val = gen.next()
        ... #Do something with the value
    except StopIteration:
        break #break out of the loop
```

Of course, the **for** loop handles the **StopIteration** exception by terminating the loop, without breaking your program.

2.12.2 Vitesse exceptions

Every Vitesse programmer must become familiar with the Python exception handling mechanisms, because many Vitesse functions raise exceptions in order to indicate error situations. Therefore, it is important to master the `try ... except ...`, and `try ... finally ...` constructions, and use them in the program, when required.

In the traditional approach, Vitesse would provide a separate exception type for each error that might happen during the execution of the program. Due to the large number of various error situations, Vitesse exceptions have been designed in a different way.

Almost every Vitesse module provides an `'error'` variable, which could be used for determining the exact type of error that happened. In all error situations, a standard `SystemError` exception is raised, and the `'error'` variable is assigned a specific string value describing the given error.

In the simplest case, when an error is encountered, the program only writes the `'error'` variable to the application's standard output file, notifying the user about the problem. Example: (for `kcs_draft` module description, see chapter 5)

```
try:
    ... # here exceptions can be raised
except:
    print kcs_draft.error
```

Typically, a more detailed error control is required. Then the program uses the `if ... elif ... else ...` construction, and specifies distinct actions to different errors, basing on the module's `'error'` variable. Example:

```
try:
    ... # here exceptions can be raised
except:
    if kcs_draft.error == "kcs_DrawingCurrent":
        ... # save & close the current drawing
    elif kcs_draft.error == "kcs_FormNotFound":
        ... # drawing form not found - notify the user
    elif kcs_draft.error == ... # some other errors
        ... # actions to be performed
    else: # another error not listed above
        ... # actions to be performed
```

i The strings `'kcs_DrawingCurrent'`, `'kcs_FormNotFound'`, etc. are examples of the predefined descriptions of errors that can happen while using the Vitesse API. They are assigned to the module's `error` variable, when a Vitesse exception is raised. The documentation of each function lists the possible values of the module's `error` variable (error descriptions).

If the protected code fragment is able to raise not only Vitesse exceptions, but also some standard Python exceptions, these should be handled **before** the Vitesse exceptions, using separate `except:` blocks.

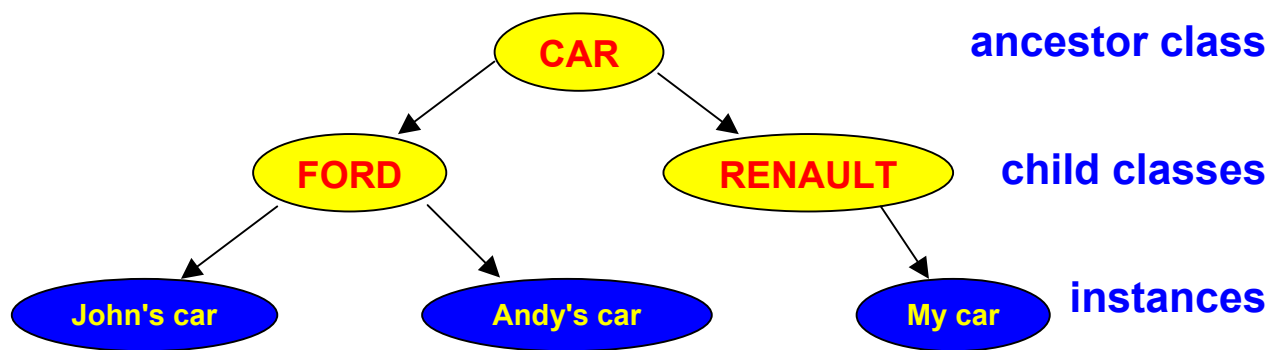
Example:

```
try:
    ... # here various exceptions can be raised
except KeyError:
    ... # some dictionary lookup failed
except IndexError:
    ... # some list/tuple indexing failed
except ...: # some other standard Python exception
    ... # actions to be performed
except: # any other exception is caught here!
    ... # actions to be performed in the case of either
        # Vitesse exceptions, or some other exceptions
        # not listed above
```

2.13 Python Classes

As Tribon Vitesse makes often use of Python classes, we have to get some basic understanding of this language concept. Python classes are in principle similar to the object or class types commonly used in Delphi, C++ or Java

languages. They are mainly used to encapsulate the data structures and behaviour of some real-life or abstract objects. Tribon Vitesse defines, among other things, the basic geometric entities as Python classes.



First, the class type has to be defined, which describes the general look of all instances of that class type. Roughly, the structure of the class definition may look as follows (example):

```

class Panel:
    cName = 'Panel class'           # An example of class variable
    ...
    def __init__(self, name, weight, numPlates): # Constructor
        self.name = name           # Examples of instance variables
        self.weight = weight
        self.nPlates = numPlates

    def SetWeight(self, value):      # Example of class method
        self.weight = value        # Accessing instance variables

    def PrintName(self):            # Example of class method
        print self.cName           # Accessing class variables

    def __repr__(self):            # Another 'special' method
        return "%s: '%s', weight %0.3f kg, number of plates: %d" % \
            (Panel.cName, self.name, self.weight, self.nPlates)
    ...
  
```

As we can see, the whole class definition is indented under the first line (**class Panel:**). The class can define three types of items:

- **class variables**
They belong in fact not to the instance, but to its class type, and thus are common to all instances of the given class. They are defined within the class, but outside of any of its methods, and are accessible using either the class or instance prefix. Example: **self.cName**, or **Panel.cName**
- **instance variables**
Each of the instances of the given class holds its own copies of these variables. They are defined in a method within a class definition, and are accessible using the instance prefix. Example: **self.name**, **self.weight**, **self.nPlates**. **NOTE: Class prefix cannot be used for instance variables!**
- **class methods**
They are functions defined within the class definition. Their first parameter (commonly called **self**) points to the class instance, for which the given method has been called. They are accessible using the class or instance prefix. Examples: **Panel.__init__()**, **x.SetWeight()**, **Panel.__repr__()**, **self.PrintName()**

i In the class method definition we can refer to the class and instance variables using the **self.<variable name>** notation, as shown in the example above.

i **NOTE:** As an exception to the above rule, don't use **self.<variable name>** syntax to **DEFINE** the class variable. By doing this, you would create instead a new **instance variable** with the same name as an existing **class variable**. The proper way of modifying a class variable is to use the **class_name.<variable_name>** syntax in the assignment (example below).

After a class type has been defined, instances of the class can be created, that can use all of the class' variables and methods.

```

x = Panel('ESB-AY012', 120.0, 4)      #x is the new Panel class instance
print x.name, x.nPlates              #Accessing instance variables
    ESB-AY012 4
x.SetWeight(100.0)                   #Class method called
x.weight                             #Modifying the instance variable
    100.0
Panel.SetWeight(x, 100.0)             #An alternative syntax
x.PrintName()                        #Another class method called
    Panel class
Panel.PrintName(x)                   #An alternative syntax
    Panel class
y = Panel('ESB-GIRD01', 432.0, 9)     #another Panel class instance
#Accessing class and instance variables through the instance prefix
print x.cName, x.nPlates, y.cName, y.nPlates
    Panel class 4 Panel class 9
Panel.cName = "New"                  #change of the class variable
print x.cName, y.cName               #both x and y have got the new name
    New New

```

- ① When the instance calls one of the class methods, it uses 'dot' notation and omits the first 'self' parameter from the class method's header. The class instance takes the place of the omitted 'self' parameter. This means, that:

x.SetWeight(value)

is an equivalent of:

Panel.SetWeight(x, value)

The Tribon M3 distribution contains many examples of Python class definitions (see, for example, the files: KcsPoint2D.py, KcsVector3D.py, KcsTransformation3D.py, etc.).



The Python language manuals contain additional information about the Python classes. In particular the reader might be interested in studying the inheritance concept and the special class methods (like e.g. `__init__`), which help the programmer to better integrate his classes with the standard Python language abilities.

2.14 Modules

Modules are the collections of functions, classes, variables and other statements. They help to build libraries of Python code, that can be re-used, and to separated logically various definitions used in Python scripts. In order to get access to the module's definitions, the module must be imported to the program.

```

import math
a = math.sin(math.pi/4.0)

```

The above form of the **import** statement interprets the given module, and grants the access the module's definitions using the syntax **<module name>.<item name>**.

```

from math import sin, pi
a = sin(pi/4.0)

```

The **from <module> import <item>** syntax interprets the module, and grants the access to the given module's definitions using the item's name directly. In the above example, the program does not gain access, for example, to the **math.cos** function, since it has not been listed after the **import** keyword.

```

from math import *
a = sin(pi/4.0) + cos(pi/3.0)

```

The last form of the **import** statement interprets the given module, and grants the access to ALL module's definitions using the item's name directly.

- ① When using the **from module import ...** syntax there is a danger of overwriting (loosing access to) any possibly existing identifiers with the same name as the one's defined in the imported module.
A module is interpreted only once. If the program contains multiple **import** statements referring to the same module, the module is not reinterpreted.

2.15 How to choose the right data structure?

We have discussed so far the available Python standard data structures. Each of the compound data structures has its advantages and drawbacks. Let's analyse them and learn, how to choose the right data structure for our application.

Lists

- 1) ☺ They are well suited for collecting atomic (elementary) data – numbers, strings, etc. We can use the **append** method to accumulate the information, when exploring the Tribon Product Model, using e.g. Data Extraction. Then, we can loop over the collected data and analyse further each item.

Example: the list of hull block names

- 2) ☺ The list keeps the order of collected items, so this would be your data structure of choice, if your application requires the proper ordering of information.

Example: the list of part IDs for consecutive parts in a branch of a pipe

- 3) ☹ If the data to be collected are not atomic (many data for each item), we cannot just append the data to the list, because then it is difficult to know, which data belong to the given item. In this case we should consider one of the three alternatives:

- a) ☹ A nested list. In this case we manage the list of items, which are lists themselves, containing the item's data. The data to be collected are just individual elements in this inner list. The disadvantage: the item's data management is not so easy (e.g. you have to remember, that `data[1]` is your panel's weight)

Example: `info = [['TRAIN0-AY012', 120.0, 10], ['TRAIN0-AY013', 87.5, 7], ...]` – the inner list contains: the panel name, its weight, and number of plates. `info[1][1]` is the weight of the panel 'TRAIN0-AY013'.

- b) ☹ A dictionary. In this case, the items must have a piece of information that uniquely identifies the given item. This piece of information becomes the key of the dictionary, and the associated value is the list of remaining data associated with the item. This solution has the same drawback, as the previous one (item's data are contained in a list). In this case, however, we can refer to the items not by an index in the outer list, but by the unique identifier, which usually is easier to understand, than a purely numerical index. So, it is still not the best solution, but a little better, than the previous one.

Example: `info = {'TRAIN0-AY012': [120.0, 10], 'TRAIN0-AY013': [87.5, 7], ...}` – the keys are panel names, and the values are the remaining data – the weight and number of plates. `info['TRAIN0-AY013'][0]` is the weight of the panel 'TRAIN0-AY013'. It is easier to read, than `info[1][1]`, isn't it?

- c) ☺ A dictionary with a class instance as the value. In this case, we define a class Panel (see page 30), having the attributes weight and nPlates. This allows us not only to design a very clear structure of data, but additionally we can add some interesting functionality to the Panel class to make it more robust. This is clearly the solution of choice for any larger application.

Example: `info = {'TRAIN0-AY012': pan1, 'TRAIN0-AY013': pan2, ...}` – the keys are panel names, and the values (pan1, pan2, etc.) are the Panel class instances, holding the weight and number of plates information.

`info['TRAIN0-AY013'].weight` is the weight of the panel 'TRAIN0-AY013'

- d) If the nesting structure of your data is even more complicated, just use the similar approach on each level of the nesting hierarchy.

Example: Let's collect the information about the panels (as above), but now we will group the panels belonging to the same hull block together. This information could be stored as a dictionary, where the keys would be the block names, and the values would be the nested dictionaries for the panels from that block, as defined in solutions 3b or 3c

```
info = {'TRAIN0': {'TRAIN0-AY012': pan1, 'TRAIN0-AY013': pan2, ...},
        'ES123': {'SPECBY012': pan10, 'ES123-BY013': pan11, ...},
        ... }
```

Then `info['ES123']['SPECBY012'].weight` would be the weight of the panel 'SPECBY012', belonging to the block 'ES123'.

- 4) ☺ Lists are well suited to simulate stacks (LIFO = last-in, first-out) and queues (FIFO = first-in, first-out). Just use **append()** to add elements to the stack or queue, and **pop()** to retrieve the elements. The difference between the stack and the queue lies in the way items are popped off the list. For stacks, the **pop()** function should be used directly (last item added will be popped off). For queues you should use **pop(0)**, retrieving the first item from the queue.

Tuples

- 1) ☹ They cannot be modified, so it is impossible to use them to collect the data, like when using lists. They do not have methods – it is a very simple data type, without much functionality.

- 2) ☺ The simplicity of tuples makes it an ideal data type for use in functions that return many data. In this case, the function just returns a tuple consisting of the data to be returned by the function.

Example: we can write a function `getPanelData()`, returning a tuple consisting of: status, panel name, weight, number of plates, etc.

- 3) ☺ The easy low-level programming interface to tuples, makes a tuple a data type of choice for Python modules written in other programming languages (e.g. C). The arguments and return values are very often passed along as tuples.

Example: `kcs_ui.string_req()` function returns a tuple (status, result).

Dictionaries

- 1) ☺ The dictionaries can be used to collect pairs of related data (e.g. panel name and panel weight, part ID and component name of the pipe part, etc.)

Example: `info = {-1001:'COMP1', -1002:'COMP2', ...}`. `info[-1002]` is the component name for the pipe part with the ID of -1002.

- 2) ☺ If there are more related data, than two, make one of them the key, and put the rest in a tuple, list or class instance as the associated value. NOTE: the key must belong to the data type, that can guarantee uniqueness of values (integers, strings, tuples). Floating point numbers, lists, dictionaries CANNOT be used as dictionary keys.

- 3) ☹ The dictionaries cannot be used, if the order of collected items is important, since the dictionary DOES NOT preserve the item's order. In this case, you'd better choose the list of nested lists.

Example: part ID and component name of the pipe parts on a branch of a pipe.

`info = [[-1007, 'COMP1'], [-1001, 'COMP2'], [-1003, 'COMP3'], ...]`. Now it is clear, that there is a sequence of parts with IDs: -1007, -1001, -1003.

Summary

- 1) Use lists:

- a) when collecting data in general, especially for simulating stacks and queues,
- b) when the order of items is important.

- 2) Use tuples:

- a) as return value from functions, and to define functions with variable number of arguments,
- b) when writing Python modules in another programming language,
- c) when using the % string operator.

- 3) Use dictionaries:

- a) when collecting items consisting of many related data, that can be identified by a key,
- b) when the order of collected data IS NOT important,
- c) when defining functions accepting keyword arguments.

- 4) Use classes:

- a) when defining data structures combined with some functionality using these data,
- b) in order to hide the implementation of some manipulation of the data,
- c) in order to clarify the usage of some compound data structures.

- 5) General notes:

- a) Nest appropriate structures, if necessary, to obtain the right organisation of your data,
- b) Use the available methods of lists and dictionaries to manipulate the data,
- c) With the growing programming experience, start using classes, which are the best tools for organising your data.

2.16 Advanced Python Programming

For the more advanced programmer there is a great deal of literature available on the Python language. A good source of information is "Programming Python" by Mark Lutz (published by O'Reilly & Associates Inc.). There is also a great deal of information freely available on the Internet. A good starting place for anyone interested in the latest news about the Python language development is <http://www.python.org>.

3 Tribon Vitesse Utilities

3.1 Introduction

The following chapters describe the functions that have been created within Tribon Vitesse as the basic programming interface between the Tribon applications, and the Python interpreter. These include possibilities to:

- get input data from the user,
- interactively define co-ordinates, and translate them between the available co-ordinate systems,
- manage the application's window,
- use Vitesse in batch.

The functions have been split into two Vitesse modules: **kcs_util** (utilities) and **kcs_ui** (user interface). Since they often work together, instead of studying these modules separately, we will concentrate on the typical patterns of their co-operation.



The description in this manual is intended to give only a basic understanding of the purpose and usage of these functions. A more detailed description can be found in the Tribon Vitesse User's Guide.

Before going into the details of Tribon Vitesse Utilities, let's see, how Vitesse functionality is used in Tribon system,

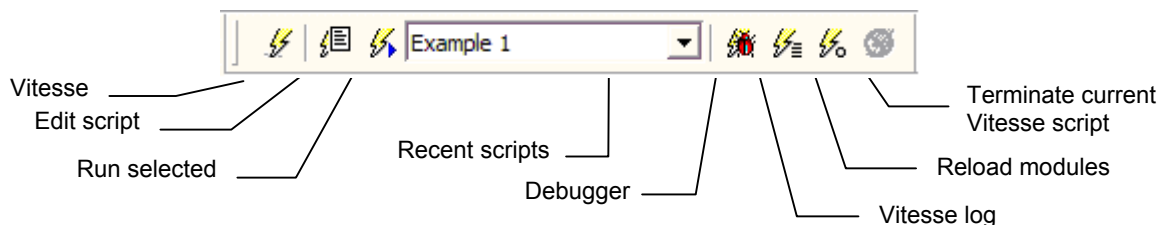
3.2 Running Vitesse Programs and viewing the Output

Vitesse programs are prepared using a text editor (e.g. Notepad, PythonWin, or ConTEXT) and must be saved on the disk in the text format as documents with the **.py** extension. The location may be arbitrary, but each project defines a special folder, where Tribon looks for Vitesse macros by default. The name of this folder is specified by the **SB_PYTHON** Tribon environment variable.



***SB_PYTHON** is a place for your Vitesse programs. On the other hand, all your Python modules (appearing in the **import** statements) must be stored in one of the folders listed in the definition of the **PYTHONPATH** system environment variable.*

In Tribon M3, the source text of the currently selected macro can also be edited from within the application, using the menu command **Tools → Vitesse → Edit**. In order to run a Vitesse program from the interactive Tribon application you have to click on the menu **Tools → Vitesse → Run script**, select the Vitesse program in the file browser, and click **OK**. Tribon applications make it easy to re-run recently used Vitesse programs by selecting its name from the provided combo-box on the Vitesse toolbar, and clicking on the re-run button, or using the menu command **Tools → Vitesse → Run selected**. Each use of the **Tools → Vitesse → Run script** menu command adds the script name to the above-mentioned combo-box for a later re-run.

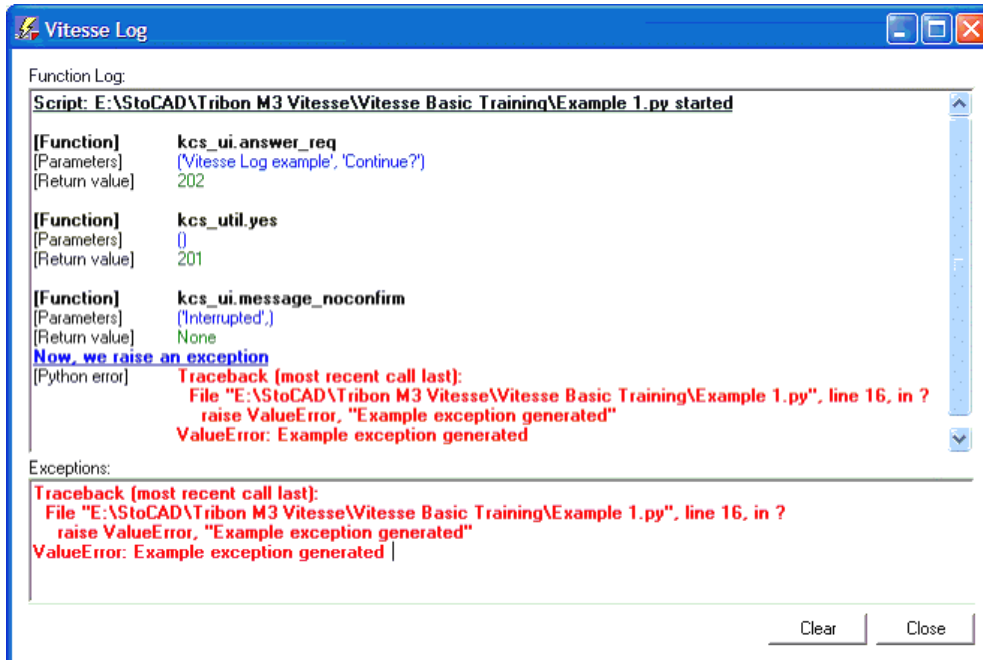


If you have configured an external Python debugger (e.g. Wing IDE), you can launch it using the menu command **Tools → Vitesse → Debug**, which comes in handy during the program's development stage.

Another nice tool is the **Vitesse Log Window** that can be activated by the **Tools → Vitesse → Log window** menu command or the corresponding button on the Vitesse toolbar. This window holds the important information about the execution of the Vitesse program, such as: the program's file name, executed triggers, Vitesse functions called with their parameters and return values, and finally the exceptions information. The program is able also to print its own messages to this log window to facilitate the debugging process during the program's development phase. The picture on the next page demonstrates the possible contents of the Vitesse Log Window after running the "Example 1.py" script. The

window contains the information about the Vitesse functions called (their name, parameters and result), generated exceptions (red text), and the text written by the program (blue, underlined text).

The menu command **Tools → Vitesse → Options** configures certain aspects of Vitesse, such as: the editor used for editing the Vitesse programs, logging options, debug script name, and the number of recently run script names, kept in the combo-box on the Vitesse toolbar for a quick access. The menu command **Tools → Vitesse → Reload** reloads all imported modules, so that their most current definition is used (important during the development phase!).



A Vitesse program terminates when the last statement in the program's source text is executed. Sometimes however, the program terminates abnormally via an unhandled exception. The reason for the program termination, and the entire program output, can be viewed in the standard output file of the interactive Tribon application from which the Vitesse program has been launched. The standard output file (the log file) can be found in the subdirectory defined by the environment variable `SB_SHIPPRINT` and can be viewed with any text editor. The simplest method is perhaps to use the **Log Viewer** facility, which locates automatically the log file of the given application.

i Since the log file is in use while the application is running, you may need to terminate the application in order to see the contents of the log file.

When the Vitesse program terminates as the result of an unhandled exception, the log file will contain the following traceback information:

- the source line that caused the error,
- source line numbers from the call stack,
- short description of the exception.

Of course, the exception information can be also read from the Vitesse Log Window, as described above, but this window will not contain the program's output, which must be read directly from the log file of the application.

3.3 Vitesse Python Classes

The interface between the Vitesse program and Tribon environment often deals with 2D or 3D points, lines, arcs, string contours, texts, symbols, model objects, attributes of various objects, etc. These objects have their own specific properties and behaviour. The object-oriented programming methodology helps to encapsulate them in well-defined 'black boxes'. The programmer does not need to think about the details of their functionality, but can focus on the overall program on which he is working. This is also a tool for implementing the multi-layer program paradigm.

The Tribon M3 distribution contains a number of Python modules describing basic objects encountered often in the real Vitesse programs. In most cases, each class is defined in a separate Python module having the name formed by adding the '**Kcs**' prefix to the class name.

Example: the **Point2D** class is contained in the module **KcsPoint2D**, in accordance with the above rule, but the class **SymbolicView** is defined in the module **KcsInterpretationObject**, together with the class **CurvedPanelView** (a rare exception to the above rule).

Below you can find some examples from a long list of the available Python classes:

I. 2D geometry objects

- Point2D, Vector2D - 2D point and 2D vector, defined as two real co-ordinates
- Rline2D - 2D line segment, defined as two 2D points (uses: Point2D)
- Rectangle2D - 2D rectangle, defined as two 2D corner points (uses: Point2D)
- Arc2D - 2D arc, defined as two 2D points and an amplitude (uses: Point2D)
- Circle2D - 2D circle, defined as a 2D point and a radius (uses: Point2D)
- Contour2D - 2D contour, defined as a list of straight segments or arc segments (uses: Point2D)

II. 3D geometry objects

- Point3D, Vector3D - 3D point and 3D vector, defined as three real co-ordinates
- Arc3D - 3D arc, defined as two 3D points and an amplitude (uses: Point3D)

III. User interface

- CursorType - type of cursor used for point indication (uses: Point2D)
- Stat_point2D_req - settings for 2D point indication (uses: Point2D)
- Stat_point3D_req - settings for 3D point indication (uses: Point3D, Vector3D)
- HighlightSet - information about the drag cursor (highlighted elements to be dragged)

IV. Pipe & Ventilation

- PipeName - pipe name string structure
- PipeSpoolProp - pipe spool properties
- PipePartAddCriteria - criteria of adding/inserting a part into a pipe (uses: Point3D, Vector3D)
- PipeMaterial - pipe material information
- PipeRoute - pipe routing information (uses: Point3D, Vector3D)

V. Hull

- BodyPlanViewOptions - options of the body plans view (uses: Point3D, Colour)
- CopyPanOptions - options of the panel copy operation
- PanelSchema - panel's schema statement manipulation functions

VI. Volume

- VolPrimitiveBlock - information about the Block volume primitive (uses: Point3D, Box)
- VolPrimitiveGeneralCylinder - information about the General Cylinder volume primitive (uses: Point3D, Vector3D, Contour2D)
- VolPrimitiveTorusSegment - information about the Torus Segment volume primitive (uses: Arc3D)
- VolPrimitiveTruncatedCone - information about the Truncated Cone volume primitive (uses: Point3D, Vector3D)

VII. Assembly

- Assembly - assembly information (uses: Date, Point3D, Transformation3D)
- AssemblyKeyInItem - assembly key-in item information (uses: Point3D)

VIII. Weld information

- WeldTable - weld table information
- WeldedJoint - welded joint information
- Weld - information about a weld (uses: Vector3D)

IX. Date & Time

- Date - date information
- Time - time information
- DateTime - combined date and time information (uses: Date, Time)

X. Miscellaneous

- Transformation3D - 3D transformation matrix for defining transformation of 3D objects
- Transformation2D - 2D transformation matrix for defining transformation of 2D objects
- Colour, Linetype - information about the colour and line type
- Model - information about the model object (whole model and its part)
- Stringlist - list of selections to be displayed to the user
- CaptureRegion2D - definition of a drawing's area, containing some drawing elements (uses: Rectangle2D, Contour2D)
- DocumentReference - information about the document reference associated e.g. with the given model



The complete list of available classes can be found in the Tribon M3 documentation in the section **Tribon M3 Developer's Toolkit → Vitesse → Python Class Quick Reference Index**. The source files of these classes contain also many useful comments, and are the ultimate source of information about their proper use in Vitesse programs.

Tribon Vitesse provides a set of useful functions to create programs able to work with the Tribon system. Many of these functions use the instances of the Vitesse classes as their parameters or return values. In order to create and use these class instances, the program must contain the **import** statements including the necessary class definitions.



In order to create an instance of one of the Vitesse classes, you need often to create some auxiliary objects, required for that instance definition. In this case you must also import the modules containing the definitions of these auxiliary classes. Example:

```
handle = kcs_draft.circle_new(circle)
```

The above statement creates a circle in the current drawing by calling the function **circle_new()** from the module **kcs_draft** (described in section 5.6.2). The circle to be drawn is defined as the parameter **circle**, being the Circle2D class instance. The circle definition requires the **centre** defined as a 2D point (Point2D class instance), and a **radius** (a real number). Thus, in order to make this work, the program must import not only the **kcs_draft** module, but also the modules **KcsCircle2D** and **KcsPoint2D**. The example with added missing statements is presented below:

```
import kcs_draft
import KcsPoint2D
import KcsCircle2D

centre = KcsPoint2D.Point2D(100, 200)      # the centre
circle = KcsCircle2D.Circle2D(centre, 50)  # the circle
handle = kcs_draft.circle_new(circle)
```

3.4 Messages

We begin our discussion of the available utility and user interface functions from the functions displaying the messages to the user. In order to use this functionality, we need to import the **kcs_util** and **kcs_ui** modules. That's why in our programs we will often find these two lines:

```
import kcs_util
import kcs_ui
```

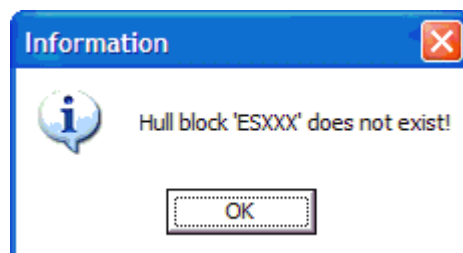


*In all the examples in the Training Guide we will assume, that the proper modules have been imported to the program using the **import** statement. To keep the examples short, sometimes we will mark the missing parts of the code by ellipsis '...'*

The **print** statement writes a message to the application's log file. We can also write our messages to the user-defined file, using the Python file management abilities. Both these methods do not allow the messages to be instantly visible to the user – in order to see them, the user must open the application's log file or the user-defined file. The **kcs_ui** module provides appropriate functions for displaying messages on the screen.

In order to display a dialog box with the informational message to the user, the **kcs_ui.message_confirm()** function should be used:

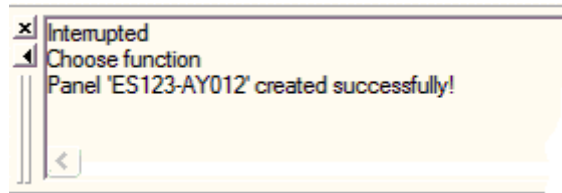
```
kcs_ui.message_confirm("Hull block '%s' does not exist!" % block)
```



Constructing message strings to be displayed to the user is a typical task for the % string operator (see section 2.2.3). The user must acknowledge the above message, before the program continues, that's why this function is often used for displaying error messages and warnings. For displaying multi-line messages, just include '\n' within the message string

as a line separator. If we want to display a message not requiring the acknowledgement, we can use the function `kcs_ui.message_noconfirm()`:

```
kcs_ui.message_noconfirm("Panel '%s' created successfully" % panel)
```



The message is displayed in the message area (lower left corner) of the running application without interrupting the program's execution. We will use this function for displaying informational messages, and the other output from a program. During the program's development we often include additional messages, informing the programmer about the program's status, values of some expressions, etc. Such debug messages can be also displayed in the Vitesse Log window (remember to turn it on!) using the function `kcs_ui.message_debug()`:

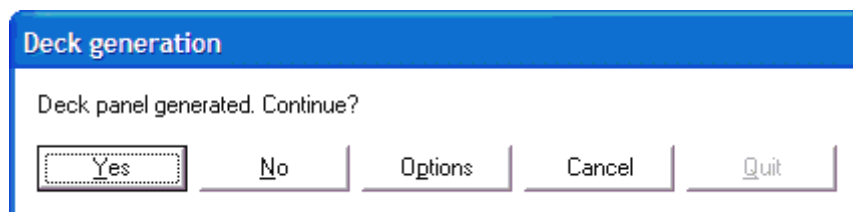
```
kcs_ui.message_debug("Now we raise an exception", (0, 0, 255), 1, 1)
```

[See the effect on the Vitesse Log picture on page 36](#)

The message is added to the Vitesse Log window. The second argument is a 3-element tuple of RGB (Red, Green, Blue) values, defining the colour of the output. Each of the values in a tuple is an intensity of the given basic colour, and is an integer from the interval $<0, 255>$. The above example defines the pure Blue colour. The message colour is optional – defaults to Black (0, 0, 0), if not given. The third argument is an optional bold flag – if non-zero, the text will be bold. The fourth argument is the optional underline flag – if non-zero, the text will be underlined. The advantage of using `kcs_ui.message_debug()` function is that it provides the appropriate information to the programmer, without disturbing the user, who typically does not see the Vitesse Log window.

The last message displaying function is `kcs_ui.answer_req()`, which we will use for asking the user a question. In order to use it, we have to import also the `kcs_util` module, because it allows us to investigate the value returned by this function, which is the user's response.

```
=> res = kcs_ui.answer_req("Deck generation", \
    "Deck panel generated. Continue?")
if res == kcs_util.yes(): #Did the user confirm?
    kcs_ui.message_confirm("Prepare data for the next deck")
```



The first argument is the dialog box title, the second is the message itself. The user can click on one of the following buttons:

| | | | |
|----------------|-----------------------------------|---------------|----------------------------------|
| Yes | - <code>kcs_util.yes()</code> | No | - <code>kcs_util.no()</code> |
| Options | - <code>kcs_util.options()</code> | Cancel | - <code>kcs_util.cancel()</code> |

Each of the buttons has a corresponding identification code (defined by the above functions from `kcs_util` module) which is returned, when the user clicks the appropriate button. By using the `if` statement we can write appropriate actions to be taken, when the user clicks the given button.

The `kcs_util` module contains also the functions returning the identification codes for various other buttons present in the Tribon environment (e.g. on the Control toolbar):

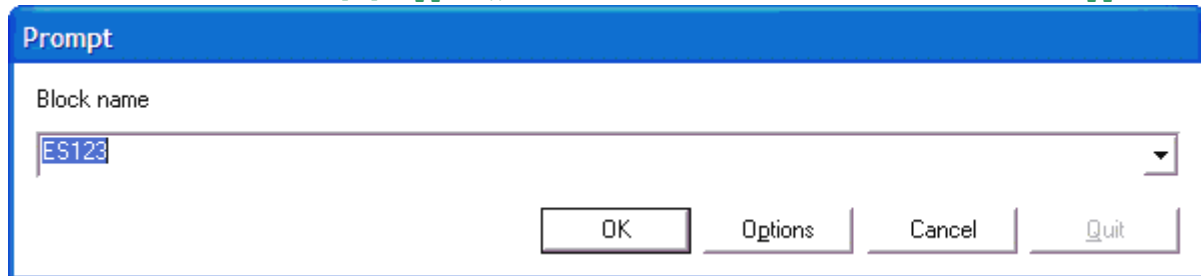
| | | | |
|---------------|----------------------------------|---------------------------|---|
| OK | - <code>kcs_util.ok()</code> | OPERATION COMPLETE | - <code>kcs_util.operation_complete()</code> |
| QUIT | - <code>kcs_util.quit()</code> | ALL | - <code>kcs_util.all()</code> |
| UNDO | - <code>kcs_util.undo()</code> | EXIT FUNCTION | - <code>kcs_util.exit_function()</code> (identical to QUIT) |
| REJECT | - <code>kcs_util.reject()</code> | | |

[See the script 'Example 2.py' in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project, which demonstrates the use of Tribon Vitesse messaging functions.](#)

3.5 Basic requests

One of the common parts of every Vitesse program is the process of getting data from the user. The basic requests are those that ask for string and numerical data. Let's prompt first the user for the hull block name ...

```
⇒ res = kcs_ui.string_req("Block name", "ES123")
   if res[0] == kcs_util.ok(): #Has the user provided the value?
       block = res[1].upper() #Get that block name and make it uppercase
```

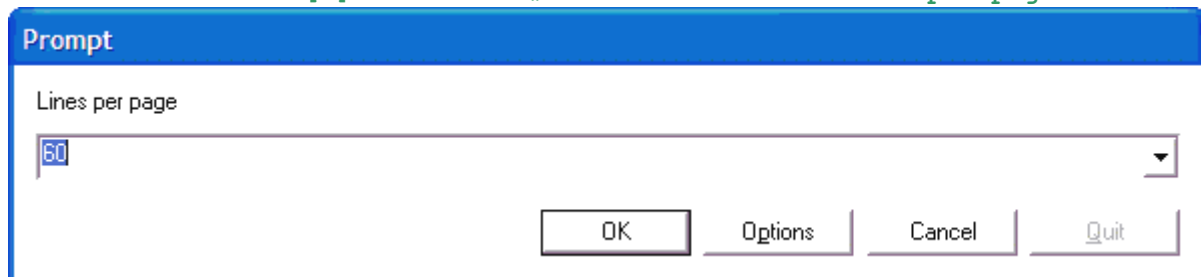


The first argument of `kcs_ui.string_req()` function is the message displayed in the dialog box. The second, optional argument is the default value, shown in the entry field. The user can immediately accept it by clicking the OK button. Of course, the user can change the value before clicking the OK button.

The result from the `kcs_ui.string_req()` function is a tuple consisting of a status code and of the returned string. In our example above we verify first, if the user has given the block name (OK button clicked), and then we get the block name from the second element of the returned tuple, converting it to uppercase for safety, as the user might have typed the block name using lowercase characters.

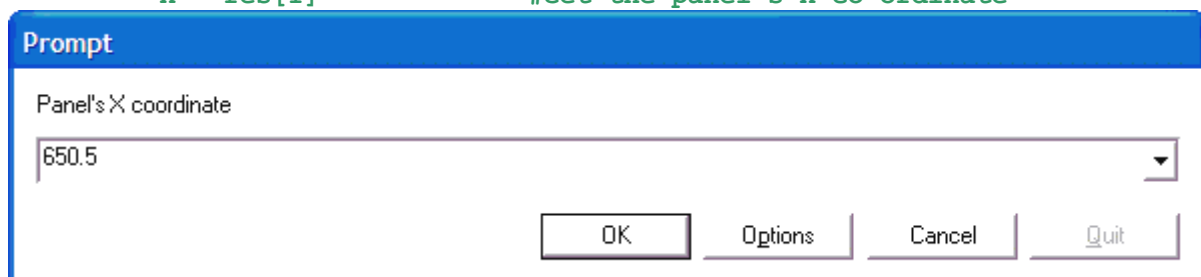
For simplicity, our example does not check, if the user has clicked Options, Cancel, or any other button – all user responses other than OK are considered as the "No data available" answer. Of course, your program can take the other possible user responses into account by adding some more `elif` blocks. Now, let's get the number of lines per page in a report (an integer number) using the `kcs_ui.int_req()` function ...

```
⇒ res = kcs_ui.int_req("Lines per page", 60)
   if res[0] == kcs_util.ok(): #Has the user provided the value?
       LPP = res[1]           #Get the number of lines per page
```



This time the result (LPP) is an integer number. You can observe a similar pattern as for the function `kcs_ui.string_req()`: displaying a dialog box, analysing the button's identification code, and if it indicates, that the OK button has been clicked – getting the value. The third function in this group is the function `kcs_ui.real_req()`, which prompts the user for the floating point value ...

```
⇒ res = kcs_ui.real_req("Panel's X co-ordinate")
   if res[0] == kcs_util.ok(): #Has the user provided the value?
       X = res[1]              #Get the panel's X co-ordinate
```



This time you won't see any default value in the dialog box, because it has not been provided in example. The user has to key in some value for the X co-ordinate. Again, the pattern of using this function is exactly the same.

🔗 Open 'Example 3.py' in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project to see the code generating the above dialog boxes.

3.6 Point requests

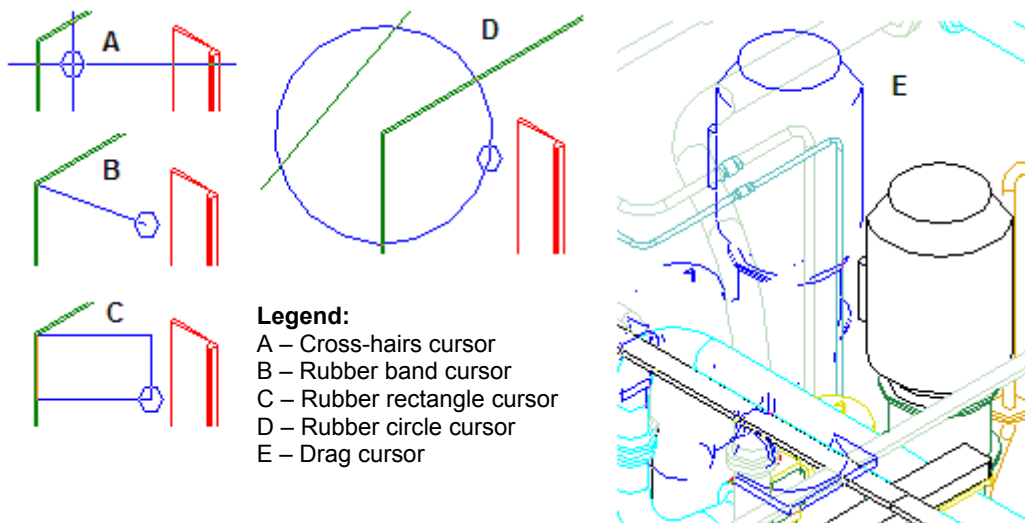
In this section we will concentrate on the functions prompting the user to indicate points on the drawing or in the global co-ordinate system. Tribon Vitesse handles points using Point2D and Point3D classes – make sure you are familiar with their definition, and remember to import the appropriate class modules into your program. Let's study the first example of a 2D point request:

```
point = KcsPoint2D.Point2D()
⇒ res = kcs_ui.point2D_req("Indicate a point", point)
if res[0] == kcs_util.ok(): #Has the user indicated a point?
    kcs_ui.message_confirm("X=%0.1f, Y=%0.1f" % (point.X, point.Y))
```

Of course, the **KcsPoint2D**, **kcs_util**, and **kcs_ui** modules have to be imported. The last three lines form an already familiar 3-stage pattern: prompt the user, check if the status code (**res[0]**) is equal to **kcs_util.ok()**, and then use the co-ordinates of the indicated point.

❗ Since here we are using a class instance (**Point2D**), an additional stage is necessary – the initialisation of the class instance, before we proceed with the usual 3-stage pattern, described above.

The obtained co-ordinates are expressed in the drawing co-ordinate system with point (0, 0) at the origin of the drawing form. The **kcs_ui.point2D_req()** function by default displays a cross-hair cursor and sets the Cursor point indication mode. If other settings are desired, we must supply them to the function in a properly configured **Stat_point2D_req** class instance (third argument). It is also possible to handle line locking buttons by providing appropriate settings in a properly configured **ButtonState** class instance (fourth argument). The picture below demonstrates the available cursor types:



🔗 See also the example function **VTBasic.getPoint2D()** which shows, how to set up cursor type and point indication mode. The **Point2DLockReq()** function in **CommonSample** module in 'Vitesse\Lib' folder is a comprehensive example of handling the locking buttons.

```
cor1 = KcsPoint2D.Point2D(100, 100)
⇒ res = VTBasic.getPoint2D("Indicate second corner", \ #message
                           "ModeNode", \             #point indication mode
                           "RubberRectangle", \       #cursor type
                           cor1)                     #reference point
if res[0] == kcs_util.ok():
    cor2 = res[1]
    kcs_ui.message_confirm("X=%0.1f, Y=%0.1f" % (cor2.X, cor2.Y))
```

The last example was made significantly easier, thanks to the use of the `VTBasic.getPoint2D()` function, where all the details were hidden. Two additional classes (`CursorType`, `Stat_point2D_req`) had to be imported, the cursor object was defined and used for defining the settings object. Finally our `kcs_ui.point2D_req()` function was called, using the prepared settings.



See the documentation of the `Stat_point2D_req` class for the available point indication modes, and the class `CursorType` for the available cursor types.

The co-ordinates in the drawing's co-ordinate system are important for placing elements on the drawing, and sizing drawing elements. If the indicated point lies on the model view, we can translate it to the 3D co-ordinate system using the co-ordinate translation functions described in the next section (3.7).

We can also prompt the user to indicate a 3D point directly.

```
settings = KcsStat_point3D_req.Stat_point3D_req()
point = KcsPoint3D.Point3D()
⇒ res = kcs_ui.point3D_req("Indicate a point", settings, point)
if res[0] == kcs_util.ok():
    kcs_ui.message_confirm("X=%0.1f, Y=%0.1f, Z=%0.1f" % \
                           (point.X, point.Y, point.Z))
```

The function `kcs_ui.point3D_req()` requires the settings object to be provided as the second argument. We have to import the following class modules: `KcsStat_point3D_req`, and `KcsPoint3D`. The simple example above does not set any special settings – all settings take default values. In this case the `kcs_ui.point3D_req()` function expects the user to indicate an event point and does not impose any locking. If you want the function to behave differently, you must set up appropriate attributes of the settings variable.

See the function `VTBasic.getPoint3D()` for an example of using the settings for the function `kcs_ui.point3D_req()`.

Again see, how this function can simplify your code:

```
vector = KcsVector3D.Vector3D(0, 0, 1) #a vertical line
point = KcsPoint3D.Point3D(0, 0, 1000) #going through (0,0,1000)
⇒ res = VTBasic.getPoint3D("Indicate the panel's corner", \
    "ModeNode", "ModePickLine", "Lock line", point, vector)
if res[0] == kcs_util.ok():
    point2 = res[1]
    kcs_ui.message_confirm("X=%0.1f, Y=%0.1f, Z=%0.1f" % \
                           (point2.X, point2.Y, point2.Z))
```

Thanks to the imposed line lock (vertical), you should see the message with the co-ordinates X=0.0, Y=0.0, and some Z co-ordinate depending on the actual point indicated. Again, our usual 3-stage pattern is preceded by the initialisation of some auxiliary class instances (vector, point).



The function `kcs_ui.point3D_req()` adjusts the co-ordinates of the indicated point, if some locking has been imposed. On the other hand, the function `kcs_ui.point2D_req()` does **NOT** adjust the co-ordinates, if the locking buttons were used – in this case the program must take care of the proper co-ordinate adjustment.

3.7 Co-ordinate translation

In Tribon we work with different kinds of co-ordinate systems:

- drawing co-ordinate system, (U, V)
- global co-ordinate system, (X, Y, Z)
- panel co-ordinate system (U, V, W)

Tribon Vitesse provides means for translating between these co-ordinate systems. The example below shows a translation from the drawing co-ordinate system to the global co-ordinate system. The translated `point` lies on the panel, whose name is provided in the call:

```
... #point variable refers to the 2D point indicated by the user
panel = 'ES123-AY012'
⇒ res = kcs_util.tra_coord_ship(point.X, point.Y, panel)
```

```

if res[0] == 0: #Success!
    #The translated 3D point ...
    p3D = KcsPoint3D.Point3D(res[1], res[2], res[3])

```

If we don't provide the panel name, the translated point lies on the plane of the closest view.

❗ **WARNING!** This function should be used on principal plane views only, and basically we can trust only the values of the co-ordinates along the axes of that view. The co-ordinate along the axis perpendicular to the view's plane may not be correct! This is especially important, if you **DON'T** provide the panel's name.

The `kcs_ui.tra_coord_ship()` function returns a tuple consisting of four items: status, X, Y, and Z co-ordinates. If status is non-zero, an error occurred during co-ordinate translation.

If you want to get the co-ordinates in the panel's co-ordinate system, just replace `tra_coord_ship()` with `tra_coord_pan()` and be sure to provide the appropriate panel's name as the third argument. The output is also a tuple, consisting of: status, U, V, and W co-ordinates. For this function, the same warning applies, as above.

The `kcs_draft` module provides the function `point_transform()`, performing the opposite translation – from the ship's co-ordinate system to the drawing co-ordinate system.

```

import kcs_draft
... # p3D is a point in the ship's co-ordinate system
... # handle is the model view's identifier
point = KcsPoint2D.Point2D()
⇒ kcs_draft.point_transform(handle, p3D, point) #translate p3D to point
kcs_ui.message_confirm("X = %0.1f, V = %0.1f" % (point.X, point.Y))

```

❗ For a more detailed discussion of view handles, and the `kcs_draft` module in general, see chapter 5.3

In `kcs_util` module there are also two functions translating between the global co-ordinates, and the logical co-ordinates on the X, Y and Z axes (frames, horizontal and vertical longitudinal positions):

```

X = 20000 # the X co-ordinate
⇒ res = kcs_util.coord_to_pos(1, X) #1 - translate an X coordinate
if res[0] == 0: #Success
    kcs_ui.message_confirm("Frame %d, offset %0.1f" % (res[1], res[2]))

```

The first argument in `kcs_util.coord_to_pos()` function is the axis index (1 for X, 2 for Y, and 3 for Z axis). The second is the co-ordinate in the global co-ordinate system to be translated. The result is a tuple consisting of: status, FR or LP number, and the offset. The translation is successful, if status is ZERO. The opposite translation is done using the `kcs_util.pos_to_coord()` function:

```

yLP = 7 #horizontal longitudinal position
⇒ res = kcs_util.pos_to_coord(2, yLP) #2 - translate the horizontal LP
if res[0] == 0: #Success
    kcs_ui.message_confirm("Y co-ordinate: %0.1f" % res[1])

```

Again, the first argument is the axis index, but the second is the logical co-ordinate (LP7 in this case) to be translated. The result is a tuple consisting of: status, and the translated co-ordinated in the global co-ordinate system. The translation is successful, if status is ZERO.

Exercise 1: Creating the frame table

Get from the user the frame number range, and create a file containing the frame table showing the frame number and the corresponding X co-ordinate (in metres) from the given frame number range in three columns. For output formatting, use the '%' operator.

Exercise 2: Displaying co-ordinates of indicated point

Open or create a drawing with at least two model views, then indicate a 3D point. Translate the point co-ordinates to the logical FR and LP co-ordinates with offsets and display them on the screen. If the translation fails (e.g. LP co-ordinates not defined), display the corresponding co-ordinate as a real number.

3.8 Selections

Examples of selections are the pop-up menus, where the user selects an item by clicking a button, and the dialog boxes giving a choice of panel symmetry settings, or the pipes from the given module. Tribon Vitesse provides functions for presenting selections to the user, and allowing him to make a choice. The list of alternatives should be provided as the list of strings

```
choices = ["P", "S", "SBP", "SP"] #alternatives
⇒ res = kcs_ui.choice_select("Panel generation", \
    "Select the panel's symmetry", choices) #Make a choice
if res[0] == kcs_util.ok():
    index = res[1]                #Index of the alternative (1 .. 4)
    text = choices[index-1]       #Text of the alternative
```

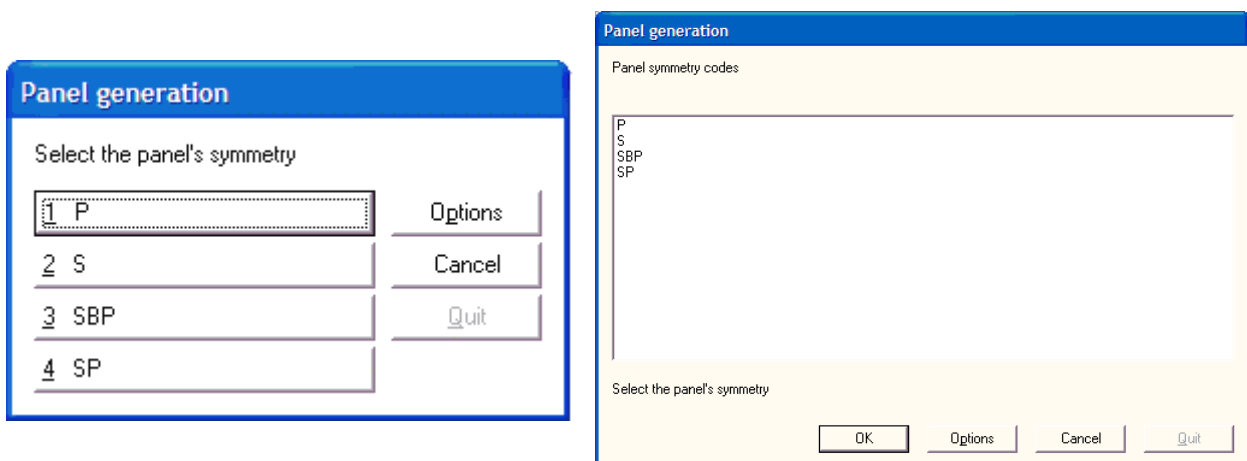
The `kcs_ui.choice_select()` function displays a dialog box with buttons having the alternatives as captions. The user makes a choice by clicking the appropriate button. The arguments of the function are: the title of the dialog box, the prompt, and a list of alternatives (strings). The result is a tuple consisting of:

- the status code, and
- the index of the chosen alternative (counted from 1, not from 0!).

This function cannot handle more than 20 alternatives. If you need to present a choice from more than 20 items, use the function `kcs_ui.string_select()`. The above example rewritten to use this function looks as follows:

```
⇒ res = kcs_ui.string_select("Panel generation", "Panel symmetry codes", \
    "Select the panel's symmetry", choices)
if res[0] == kcs_util.ok():
    index = res[1]
    text = choices[index-1]
```

The difference is that the `kcs_ui.string_select()` function displays the items not as buttons, but in a list control, and in order to choose one of them, the user has to click first an item, then the OK button. There is also an additional argument in the function call – the header, which is provided between the window's title and the prompt. It is displayed above the alternatives. The number of alternatives is limited only by the available memory. The differences between these functions are shown on the picture below:



For compatibility reasons, these functions still accept the `Stringlist` class instance as the last argument, to define the alternatives. Then, the definition of alternatives looks as follows:

```
import KcsStringlist
choices = KcsStringlist.Stringlist("P") #initialise (first alternative)
choices.AddString("S")    #add second alternative
choices.AddString("SBP") #... etc.
choices.AddString("SP")
```

When the user makes a choice, in order to retrieve the text of the chosen alternative, we have to access the **StrList** attribute of the **Stringlist** class, as shown below

```
if res[0] == kcs_util.ok():
    index = res[1]                    #Index of the alternative (1 .. 4)
=>    text = choices.StrList[index-1] #Text of the alternative
```

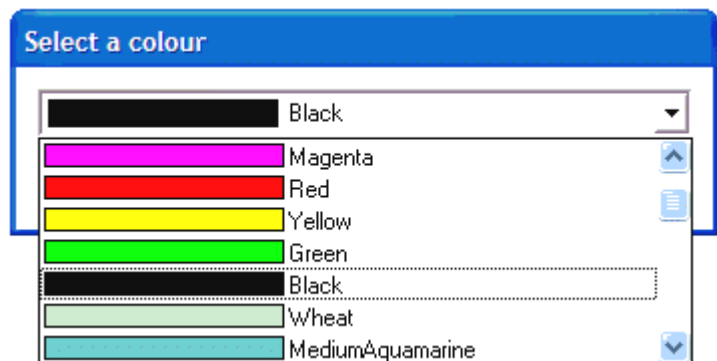
❗ The use of the **Stringlist** class becomes deprecated now. It is much easier to handle the alternatives using standard Python lists of strings.

Tribon Vitesse lets us also select a colour, using the **kcs_ui.colour_select()** function. Since colours in Tribon Vitesse are managed by the **Colour** class instances, we have to import the **KcsColour** module.

```
colour = KcsColour.Colour()
=> res = kcs_ui.colour_select("Select a colour", colour)
if res[0] == kcs_util.ok():
    kcs_ui.message_confirm("Selected colour: " + colour.GetName())
```

The function displays a list of bars in the available colours. By clicking a bar, the corresponding colour is selected. This is also another example of our 3-stage pattern, also preceded by the initialisation of the proper class instance (**Colour**).

❗ The use of this function **DOES NOT** mean that the selected colour becomes the current modal colour!

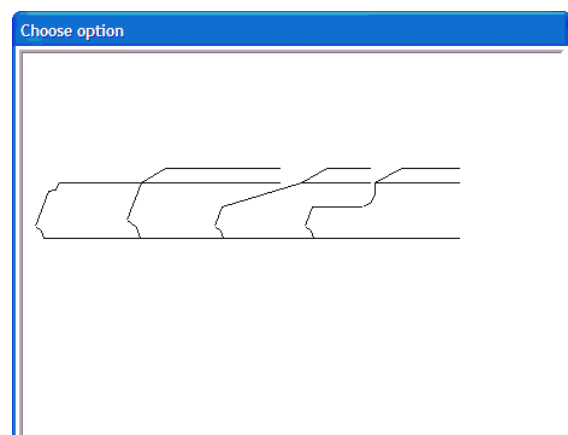
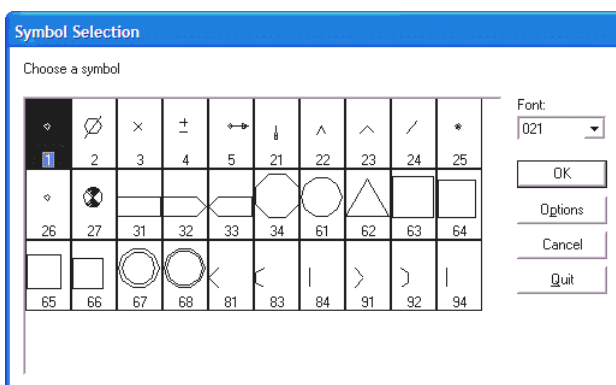


The last function giving the user a choice is **kcs_ui.symbol_select()**, which presents a list of symbols to choose from.

```
font = 21
=> res = kcs_ui.symbol_select("Choose a symbol", font)
if res[0] == kcs_util.ok():
    symbolNo = res[1] #chosen symbol from the given font
```

or

```
import KcsSymbolist
sList = KcsSymbolist.Symbolist(93, 1) #font 93, symbol 1
sList.AddSymbol(93, 3) #font 93, symbol 3
... #add more symbols from the desired fonts
=> res = kcs_ui.symbol_select("Choose a symbol", sList)
if res[0] == kcs_util.ok():
    font, symbolNo = res[1], res[2] #chosen font and symbol numbers
```



The first variant displays all symbols from the given font. The second variant employs the class **Symbolist** to collect symbols from various fonts to be displayed for selection. In this case, the returned tuple contains the status, and the font and symbol numbers (the first variant does not include the font number in the returned tuple). Additionally, the variant with the **Symbolist** class instance displays the provided prompt in the message area of the application, not inside the dialogue box.

3.9 Application's window management

Tribon Vitesse provides the following functions for basic window manipulation:

```
kcs_util.app_window_minimize() – turns the window to an icon on the taskbar
kcs_util.app_window_maximize() – enlarges the window to the maximum possible size
kcs_util.app_window_restore() – restores the original window size
kcs_util.app_window_refresh() – repaints the window
```

The last function could be useful, when using third-party graphical user interface packages (e.g. wxPython, Tkinter). If the Tribon application's window is not refreshed automatically (because another window grabs all Windows event messages), you can force the Tribon application's window repainting by calling this function.

Let us mention one more function from the `kcs_gui` module: `kcs_gui.frame_title_set()`, which allows to modify the Tribon application's window title.

```
kcs_gui.frame_title_set("Drafting", "Drawing %o on project %p")
```

If we are just working on the drawing 'DECK2' on project 'ES', then the window's title will read:

'Drawing DECK2 on project ES – Drafting'

 *For the explanation of the available formatting codes, see the function's documentation.*

3.10 Batch Vitesse

Tribon M3 supports the automated execution of Vitesse scripts, not requiring the interaction with the user, using the Batch Vitesse functionality. All Tribon applications are launched by the special program, called the Job Launcher (`tbstartjob.exe`). It provides the proper working environment for the application (e.g. assigns the job number, opens the log file, registers the job in the Log Viewer). It can also instruct the application to execute the given Vitesse script at startup. On the other hand, the `kcs_util` module provides the function `kcs_util.exit_program()`, which is a request to the application to terminate itself.

Combining these two features we get the Batch Vitesse functionality. It is used as follows ...

1. Create a Vitesse script not requiring any interaction with the user, and containing a final call to `kcs_util.exit_program()`,
2. Use Job Launcher to run the desired application, and specify the Vitesse script to be executed at startup, optionally with the proper arguments.
3. The application will start, execute the Vitesse script, and thanks to the `kcs_util.exit_program()` call it will terminate automatically.

This gives us an opportunity to create scripts, that can run unattended – nightly reports from the model, verification of the model, creation of automatic drawings, panel splitting, etc.

The Job Launcher accepts the following command-line arguments:

```
-application "application_name"
-script "Vitesse_script_full_path"
-scriptargs argument
-minimize
-nosplash
```

Notes:

1. The valid application names and input/output specification can be found in the applications.xml file (SB_SYSTEM folder)
2. The full path to the Vitesse script must be given (e.g. "C:\Tribon\M3\Vitesse\Report.py").
3. One `-scriptargs` option defines one argument. If you want to pass more arguments to the script, use multiple `-scriptargs` options. The arguments can be retrieved using the `sys.argv` list from the `sys` module.
4. If `-minimize` is used, the application will start as an icon on the taskbar
5. If `-nosplash` is used, the splash window, usually displayed when the application's starts, will not be shown.

Example:

```
tbstartjob.exe -application "Drafting" -script "C:\BV.py" -scriptargs 10 -scriptargs "Second arg" -minimize
```

The above command launches Drafting, executes the the BV.py script and closes the application.

The script BV.py can be found in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project.

*ⓘ In order to test the above command, copy it to C:\ or provide instead the correct path to the script. You may want also to add the **Bin** folder of Tribon M3 to your **PATH**.*

3.11 Miscellaneous functions

Tribon Vitesse provides yet the following Utility functions:

```
kcs_util.clean_workspace()
```

which performs the **Tools → Clean Workspace** function of the Tribon application, and

```
kcs_ui.model_info()
```

initiating the **Model Info** function of the Tribon application. These functions do not return any value.

4 Exploring the Tribon Product Information Model

Tribon system stores the information in the PIM (Product Information Model). It is often used by Tribon Vitesse programs to determine proper attributes of modelled items, and to choose the right algorithm path to follow. In order to make it possible, Tribon Vitesse provides three groups of functions

- accessing the Tribon environment variables,
- using Data Extraction engine to query the Tribon Product Information Model,
- listing databank objects meeting certain criteria.

4.1 Accessing Tribon environment variables

Tribon Vitesse offers you the possibility to read and set the values of Tribon environment variables.

```
⇒ project = kcs_util.TB_environment_get("SB_PROJ")
   oldSymbDir = kcs_util.TB_environment_get("SBB_SYMBDIR")
⇒ kcs_util.TB_environment_set("SBB_SYMBDIR", "C:\\MySymbols")
   ... #use symbol fonts located in the user-defined folder
   kcs_util.TB_environment_set("SBB_SYMBDIR", oldSymbDir)
```

The function `kcs_util.TB_environment_get()` returns the value of the Tribon environment variable passed as an argument, or raises the exception, if the variable is not found.

The function `kcs_util.TB_environment_set()` sets the value of the Tribon environment variable.

i *The value of the Tribon environment variable is changed for the current process only. It is not possible to change the value of the Tribon environment variables defining the databank paths.*

4.2 Data Extraction

4.2.1 Introduction

Data Extraction is a tool that retrieves the information stored in various Tribon databanks. It accepts commands from the user and returns the results. The commands consist of tokens separated by a dot ('.') character. The available information is tree-structured, and each token in the command string corresponds to the specific node in this information tree.

Each token is defined by a keyword, followed sometimes by an argument in parentheses. The argument can be:

- a number `BOUNDARY(1)`
- a range `BOUNDARY(1:4)` – boundaries from 1 to 4
- a list of numbers `BOUNDARY(1,3)` – boundaries no. 1 and 3 only
- a string `PANEL('ES1030-VX01341')`
- a wildcard `PANEL(*)` – all panel names
- a string with wildcards `PANEL('ES1030-*')` – all panel names beginning with 'ES1030-'
- none `GEN_PROPERTY` – block of general properties of e.g. a pipe



Full description of the syntax of the Data Extraction commands can be found in the Data Extraction User's Guide.

Below are some examples of the Data Extraction commands:

1. `HULL.BLOCK(*).NAME` – the names of all hull blocks
2. `HULL.BLOCK('ES1030').PANEL(*).WEIGHT` – weights of all panels from the hull block 'ES1030'
3. `HULL.PANEL('ES1030-VX01341').BOUNDARY(1:100).CORNER` – co-ordinates of all boundary corners of the panel 'ES1030-VX01341'. There won't be 100 boundaries in this panel, so by specifying an upper limit of 100 we

can be rather sure, that we catch all of them. After the last corner, Data Extraction will simply stop printing any further results. Alternatively, we can first get the value of `HULL.PANEL('ES1030-VX01341').NBOUNDARY`, which is an actual number of boundaries.

4. `EQUIPMENT('ES').ITEM('431*').COMP_NAME` – component names of all equipment in the project 'ES', whose names start with '431'.
5. `PIPE('ES').PIPMODEL('DW-IGH151').CONNECTION(1:2).REFERENCE.CONN_NAME` – names of the connected pipes or equipments for two connections of the pipe 'DW-IGH151' in the project 'ES'

❶ *Instead of using the Data Extraction User's Guide it is possible to find interactively the form of the desired Data Extraction command using the Query program (SX700). At each level of the information tree, you can append a '.HELP' token to find out, which tokens are available at the next tree level.*

4.2.2 Data Extraction in Vitesse

The access to Data Extraction engine is provided by the `kcs_dex` module, which has to be imported to the program. The example below shows a typical pattern of using Data Extraction in Vitesse to obtain a single value.

```
st = "EQUIPMENT('ES').ITEM('431PUMP').COMP_NAME"
1⇒ if kcs_dex.extract(st) == 0:           #was extraction successful?
2⇒     if kcs_dex.next_result() == 3:     #is a string-type data available?
3⇒         comp = kcs_dex.get_string() #get the component name
```

First we construct the Data Extraction command string. Then we actually perform Data Extraction by calling the `kcs_dex.extract()` function. If the result is ZERO, then there was no error during Data Extraction. Getting the value is a 2-stage process: first, we "access" the data item using the `kcs_dex.next_result()` function, then we actually get that value, using an appropriate `kcs_dex.get_XXX()` function (`kcs_dex.get_string()` for string-type data).

The value returned by `kcs_dex.next_result()` function indicates the data type of the data item to be retrieved and the `kcs_dex.get_XXX()` function to be called:

| next_result() | Data type | kcs_dex.get_XXX() function | Comments |
|---------------|----------------|-------------------------------------|--|
| 1 | integer | <code>kcs_dex.get_int()</code> | |
| 2 | real | <code>kcs_dex.get_real()</code> | |
| 3 | string | <code>kcs_dex.get_string()</code> | |
| 4 | real vector 3D | <code>kcs_dex.get_reavec3d()</code> | [x, y, z] |
| 5 | box | <code>kcs_dex.get_box()</code> | [x _{min} , y _{min} , z _{min} , x _{max} , y _{max} , z _{max}] |
| 6 | real vector 2D | <code>kcs_dex.get_reavec2d()</code> | [x, y] |
| 0 | - | - | empty tree part - no data available |
| -1 | - | - | end of result tree – no MORE data available |

If the Data Extraction command string contains wildcards, we can expect many data to be retrieved. Then our pattern must be changed accordingly.

```
st = "HULL.BLOCK(*) .NAME"
1⇒ if kcs_dex.extract(st) == 0: #was extraction successful?
    dataType = kcs_dex.next_result()
2⇒ while dataType >= 0:      #are we still extracting data?
    if dataType == 3 #is there a string available
        block = kcs_dex.get_string() #get the next block name
        ... #use the value, e.g. append it to some list
        dataType = kcs_dex.next_result() #move on to the next result
```

Here we can see, that the `if` statement has been replaced by the `while` loop. We may also understand better the meaning of the `kcs_dex.next_result()` function: it "accesses" the **NEXT** data item, and returns the appropriate data type, telling if the next data item is available or not. The code becomes a little bit more complicated, because we have to take into account the possibility of having both ZERO and POSITIVE data types returned during extraction (see below).

IMPORTANT!

When the function `kcs_dex.next_result()` returns 0, it **DOES NOT MEAN** the end of extraction yet. For interpretation of the data types returned from this function, we should apply the following rules:

- **data type > 0**

Data IS AVAILABLE for the current item, which can be found by analysing the arguments of the current Data Extraction command string, returned by the `kcs_dex.get_commandstring()` function (see section 4.2.3). Use the corresponding `kcs_dex.get_XXX()` function (see table on page 50) to retrieve the value. Extraction IS NOT terminated yet, and the next data type should be fetched using `kcs_dex.next_result()` function.

- **data type = 0**

Data IS NOT AVAILABLE for the current item, which can be found by analysing the arguments of the current Data Extraction command string, returned by the `kcs_dex.get_commandstring()` function (see section 4.2.3). Extraction IS NOT terminated yet, and the next data type should be fetched using `kcs_dex.next_result()` function.

- **data type = -1**

No more data available. The extraction IS terminated.

❗ *It is not possible to run multiple Data Extraction queries at the same time. If you need to get various kinds of data, requiring different Data Extraction command strings, run these extractions one-by-one, storing the results in an appropriated data structure (see section 2.15)*

4.2.3 Advanced techniques

In some cases, when we have to extract various kinds of data, we can decrease the number of necessary Data Extraction calls by using information stored in the command string for the first extraction.

Example: let's extract the available information about the weights of all structures in the ship. Using the standard approach, we would have to extract first the structure names, then the weights of the structures. The code below does this in a more efficient way:

```
info = {} #keys - structure names, values - weights
project = kcs_util.TB_environment_get("SB_PROJ_STRUC")
st = "STRUCTURE('%s').ITEM(*).WEIGHT" % project
if kcs_dex.extract(st) == 0:
    dataType = kcs_dex.next_result()
    while dataType >= 0: #still extracting ...?
        if dataType == 2: #real-type result
            weight = kcs_dex.get_real() #the weight
            #com has the proper ITEM argument filled in ...
            => com = kcs_dex.get_commandstring()
            n1 = com.find('M(') #locate the ITEM's opening parenthese
            n2 = com.find(')', n1) #locate the ITEM's closing parenthese
            structName = com[n1+2:n2] #isolate the structure name
            info[structName] = weight
            dataType = kcs_dex.next_result()
```

The function `kcs_dex.get_commandstring()` returns the Data Extraction command string with the arguments filled with the current data (e.g. current structure name). By analysing this string and isolating the appropriate fragment, we are able to obtain additional information about the extracted value.

There are Data Extraction queries, that return a variable number of arguments (e.g. profile parameters). In such cases the `kcs_dex.next_result()` function returns a value greater than 10:

```
st = "HULL.TRANS(9).PART('SPT9-S1').PROFILE.PARAMETER"
if kcs_dex.extract(st) == 0:
    code = kcs_dex.next_result() #code is > 10
    if code > 10:
        nData = code - 10 #code - 10 is the number of parameters
        => param = [kcs_dex.get_indexedreal(n) for n in range(nData)]
```

When `kcs_dex.next_result()` returns the code greater than 10, then the result is a sequence of floating point numbers. The code decreased by 10 gives the number of item in this sequence. In order to get the items, the function `kcs_dex.get_indexedreal()` should be called with the argument being an item index in the sequence, The list comprehension expression in the last line above collects all the items (see Example 5.py).

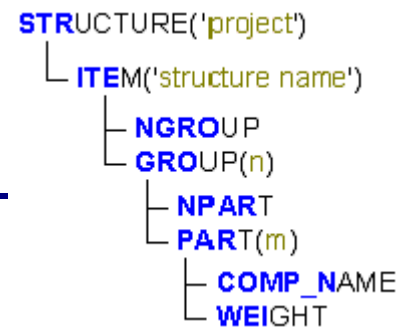
Exercise 3: Extracting transformation matrix of a panel

Write the function extracting the transformation matrix of the panel, whose name is given as an argument. The matrix should be expressed as a Transformation3D object. Then prompt the user to provide the panel name, and use your function to obtain the transformation matrix of this panel.

Hint: Use `SetByRowFromArray()` method of the Transformation3D class.

Exercise 4: Listing structure parts data

Ask for a structure name, and print the list of its parts (component name and weight). At the end of the list, print the total number of structure parts and the total weight of the structure.



Exercise 5: Selecting a panel

Extract the hull block names, and let the user select one of them. Then extract the names of the panels from the selected block, and let the user select one of them. Finally, display a message with the selected block and panel names.

Hint: Use the `kcs_ui.string_select()` function.

The solutions to the exercises can be found in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project.

Finally, let's study a little bit more advanced example, where for performing Data Extraction we are using a very flexible generator function (see section 2.10.2):

```

def DEX(command, dataTypeCode, getFunction):
    if kcs_dex.extract(command) == 0:
        dataType = kcs_dex.next_result()
        while dataType >= 0:
            if dataType == dataTypeCode:
                yield getFunction()
                dataType = kcs_dex.next_result()

⇒ #Example of usage:
st = "HULL.BLOCK(*).NAME"
⇒ for block in DEX(st, 3, kcs_dex.get_string):
    kcs_ui.message_noconfirm("BLOCK: " + block)

```

In the DEX() function we supply all variable parts of our Data Extraction pattern as the function's arguments. Additionally, the use of a generator function allows us to go through the extracted values using the `for` loop.

4.3 Listing objects in databanks

Data Extraction engine provides a very limited support for limiting directly the result set to the items meeting certain criteria. In fact, you can only use wildcards and index ranges in the arguments of Data Extraction command strings. It is impossible, for example, to ask objects created in the given time interval, having the given minimum size, or having an appropriate Object Code. Often you need to extract large amount of data in order to process them and select those meeting your criteria by analysing their properties in Python

Tribon Vitesse supports querying Tribon databanks through the `kcs_db` module. Currently it exports only one function – `object_list_get()`, which uses `ObjectCriteria` class argument for specifying the selection criteria and returns a list of `Object` class instances, describing the objects meeting these criteria.

```
... #Set up selection criteria
objList = []
⇒ kcs_db.object_list_get(criteria, "SB_OGDB", objList)
   for object in objList:
       ... #analyse the collected objects (Object class instances)
```

🔗 Please see the 'Example 6.py', that prints out the information about the panels (OC2=101) with DT (datatype) equal to 101 (OC1=101), created before the 15-03-2004.

It is also possible to set criteria regarding the object's name and size. Please play a little with this example, modifying the criteria and comparing the results with the Tribon M3 DB Utility application.

5 Vitesse Drafting

5.1 Introduction

Vitesse for Drafting contains functions for many different purposes. There are functions for creating a new drawing, placing text in drawing, creating geometric entities or drawing components, highlighting, subpicture and model objects handling, etc. The functions described belong to the following categories:

- Drawing functions
- Element handle functions
- View handling functions
- Model handling functions
- Basic geometric entities
- Texts
- Hatching
- Symbols
- Notes and position numbers
- General Design default values
- Dimensioning
- Subpicture functions
- Drawing element handling

In order to make use of these functions the Vitesse program must include the statement

```
import kcs_draft
```

As these functions often use Vitesse Python classes, additional **import** statements for the Python modules defining the needed classes are necessary before any of them are used. The **kcs_draft** module provides an 'error' variable **kcs_draft.error** (see section 2.12.2), which is set to a string describing the type of error, when an exception is raised.

Some examples of the possible values of the **kcs_draft.error** variable are given below:

| | |
|-------------------------|---|
| 'kcs_DrawingNotCurrent' | current drawing is required, but there is no current drawing |
| 'kcs_DrawingExists' | the given drawing name already exists in the drawing databank |
| 'kcs_FormNotFound' | the given drawing form is not found in the databank |
| 'kcs_NameError' | the given drawing name is not acceptable |
| 'kcs_Error' | general error occurred |



*The detailed information about the possible values of the **kcs_draft.error** variable and their meanings is given in the Tribon Vitesse documentation at each function's description.*

5.2 Drawing Functions

This section describes the functions in the **kcs_draft** module that handle the drawing as a whole. The default drawing databank is defined by the Tribon environment variable **SB_PDB**.

5.2.1 Current drawing



There can be only one drawing active at any time! Therefore before creating a new drawing or opening an existing one, first the current drawing must be closed!

In order to check if there is a current drawing, the Vitesse program can call the function

```
kcs_draft.dwg_current()
```

This function returns **1** if a drawing is current, and **0** if no drawing is current. Some functions require a drawing to be current, while the other will complain, if a drawing is current. If the current status does not meet the requirement of a function, it will raise an exception. So, we have a choice: either use the function **kcs_draft.dwg_current()**, and check, if a drawing is current, or use **try: ... except: ...** statement, and catch the exception raised by Vitesse functions.

If a drawing is current, the Vitesse program is able to get its name and the name of its drawing form, using the functions

```
kcs_draft.dwg_name_get(), and  
kcs_draft.form_name_get()
```

If the Vitesse program wants to create a new drawing or open an existing one, the current drawing must be closed first using the function

```
kcs_draft.dwg_close()
```

5.2.2 Storing the current drawing

The above function closes the current drawing **without storing**, which is usually undesirable. To store the drawing the **kcs_draft** module provides two functions:

kcs_draft.dwg_save() stores the current drawing under the current name, overwriting a possibly existing previous version of the drawing

kcs_draft.dwg_save_as(NewName, Databank) stores the current drawing under the name given as a parameter. The optional **Databank** parameter can be either 'SB_ASSPDB' (automatic assembly drawings) or 'SB_PDB' (default, standard drawing databank)

i *Saving the drawing does not close it.*

The above functions store the drawing in the native Tribon format. The latter will fail, if the drawing **NewName** exists in the databank, when this function is called. You may want to check first the existence of the given drawing, and possibly to delete it from the databank, before calling **kcs_draft.dwg_save_as()**. In such case, use the function:

```
kcs_draft.dwg_exist(dwgName, dwgType)
```

which checks, if the drawing **dwgName** exist on the databank specified by the optional **dwgType** argument (default: standard drawing databank). The result is either **1** (exists), or **0** (does not exist). Then, if you decide to remove the drawing from the databank, just use the function

```
kcs_draft.dwg_delete(dwgName, dwgType)
```

If no exception is raised, then the given drawing is removed from the databank, and you can safely use its name again, when storing your drawing with the **kcs_draft.dwg_save_as()** function.

In order to exchange drawings with other systems, Tribon provides appropriate functions for exporting the drawings into the DXF or WMF format. Tribon is able to produce either 2D or faceted 3D DXF format files. The function given below stores the current drawing in the 2D DXF format. The DXF file name (path) is given as an argument.

```
kcs_draft.dwg_dxf_export(fileName)
```

Tribon provides a limited (export only) support for the faceted 3D DXF format. The Vitesse API contains an appropriate function for this purpose:

```
kcs_draft.dwg_dxf_3d_export(fileName, viewSubviewList, detailLevel)
```

where **fileName** is the file name of the resulting 3D DXF file, **viewSubviewList** is a list of **ElementHandle** class instances identifying the views or subviews being handled, and **detailLevel** is an optional integer number, determining the exported image detail level:

- 1 – low detail
- 2 – medium detail
- 3 – high detail (default)
- 4 – extra high detail

The views or subviews, identified by the handles stored in the **viewSubviewList** parameter, are exported to the faceted 3D DXF format.

i *For a more detailed discussion of element handles, see section 5.3*

```
kcs_draft.dwg_wmf_export(fileName)
```

The above function exports the current drawing to the Windows metafile, whose name (path) is provided as the argument. The given file must not exist, or an exception will be raised.

Vitesse is also able to print the current drawing using the Windows printing services. In order to print the drawing, the program must set up an instance of the **PrintOptions** class, and pass it to the function:

```
kcs_draft.dwg_print(options)
```

where the options parameter defines important features of the printout, like: the selected printer, page orientation, number of copies, scaling, scope of the drawing to be printed, printer name, etc. See the file KcsPrintOptions.py for details.

5.2.3 Activating a drawing

If we have made sure, that no drawing is current (either by using **kcs_draft.dwg_close()** explicitly, or checking the status with **kcs_draft.dwg_current()**), we can activate another drawing or create a new one. In order to open an existing drawing we use the function

```
kcs_draft.dwg_open(dwgName, dwgType, openMode, dwgRevision, \
                  envelopeMode)
```

where **dwgName** is the name of the drawing to be opened and made current. All remaining arguments are optional, and have the following meaning:

| | |
|---------------------|--|
| dwgType | drawing type (logical databank name, default: "SB_PDB", or special drawing type constants – see the documentation) |
| openMode | kcs_draft.kcsOPENMODE_READONLY , or kcs_draft.kcsOPENMODE_READWRITE (default) |
| dwgRevision | (TDM) drawing revision name. Empty string – latest revision (default), kcs_draft.kcsBASE_REVISION – Base Revision |
| envelopeMode | kcs_draft.kcsENVELOPE_NONE – no envelope (default), kcs_draft.kcsENVELOPE_INITIAL – initial envelope, or kcs_draft.kcsENVELOPE_PERMANENT – permanent envelope |

If you don't use the optional arguments, the latest revision of the drawing will be opened from the standard drawing databank, in the read-write mode, and without envelopes. By using the optional arguments, you can, for example, open the assembly drawing, or request a specific revision of the drawing. If the given drawing does not exist on the databank, an exception will be raised. In order to avoid it, you may want to call first the function **kcs_draft.dwg_exist()**.

It is possible to create a new drawing using the function


```
kcs_draft.dwg_new(dwgName, formName, dwgType)
```

where in addition to the drawing name you can provide optionally the name of the drawing form (default – no drawing form), and the drawing type specification (see the description of the function **kcs_draft.dwg_open()**). The drawing is initialised according to the arguments, and made current. If the drawing **dwgName** exists on the databank, when this function is called, an exception is raised. Again, you may use the function **kcs_draft.dwg_exist()** to check the existence of the given drawing on the databank. Then if you decide to remove the existing drawing first, use the function **kcs_draft.dwg_delete()**. Both functions are described in section 5.2.2.

Finally, the new drawing can be initialised from a 2D DXF file using the import facility.

```
kcs_draft.dwg_dxf_import(fileName, dwgName)
```

which reads the DXF file **fileName** as a native Tribon drawing named **dwgName**.

 Please study the script 'Example 7.py' located in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project, which contains examples of using the Vitesse drawing functions discussed so far.

5.2.4 Housekeeping functions

The following functions perform some housekeeping work on the current drawing:

| | |
|--|--|
| kcs_draft.dwg_pack() | – packs the current drawing |
| kcs_draft.dwg_purge() | – removes empty subpictures |
| kcs_draft.dwg_reference_show(show, views) | – expands or collapses drawing references in the current drawing |

The **show** argument is a Boolean value: **True** causes the expansion, **False** – collapsing of drawing references. The optional **views** argument is a list of handles to the views to be expanded or collapsed. If omitted, all drawing references will be considered.

During the work on the ship's model, it happens quite often, that a drawing does not show an up-to-date information on the model views. In such cases, Tribon applications provide the validating function, which detects the model objects, present on the model views, that do not exist anymore in the model or are outdated. Tribon Vitesse supports this functionality through the function

```
kcs_draft.dwg_validate(OutOfDate, NotFound)
```

which fills up the lists **OutOfDate** and **NotFound** with **Model** class instances. The **OutOfDate** list contains all model objects, that still exist in the model, but have been updated, after the drawing views have been created or exchanged from the model. These objects should be redrawn to make the views up-to-date. On the other hand, the **NotFound** list contains all model objects, that do not exist anymore in the model, and should be removed from the views.

i *The given object appears only once in the list, and is always unreflected (panels). The interactive Validate function gives the possibility to update the views according to the results of the check. In Vitesse, you must use the returned lists of Model class instances, and update the views yourself.*

5.2.5 Layer handling functions

Tribon applications can work in one of two modes, regarding the drawing layers: show selected layers or hide selected layers. The two functions given below, return the current setting of the layer treatment mode:

```
kcs_draft.dwg_layers_is_shown()
kcs_draft.dwg_layers_is_hidden()
```

They return **1**, if the given layer treatment mode is active, otherwise **0**. In order to get the list of the selected layers (to be shown or hidden, respectively), use the functions:

```
kcs_draft.dwg_layers_shown_get()
kcs_draft.dwg_layers_hidden_get()
```

The actual showing and hiding of layers is done by the functions:

```
kcs_draft.layer_show(layer)
kcs_draft.layer_hide(layer)
```

The last two functions accept either a single instance of the **Layer** class, or a list of such instances. If the corresponding layer treatment mode has not yet been selected, these functions select this mode, resetting the list of selected layers to the ones provided as the argument. If the corresponding layer treatment mode has already been selected, the layers provided as the argument are simply added to the list of selected layers. In order to show all layers (and reset the layer treatment mode to show), use the function

```
kcs_draft.layer_show_all()
```

5.2.6 Visual area functions

When the drawing is current, Vitesse is able to show only a portion of the drawing by using the zooming function:

```
p1 = KcsPoint2D.Point2D(100, 100)
p2 = KcsPoint2D.Point2D(500, 300)
rectangle = KcsRectangle2D.Rectangle2D(p1, p2)
⇒ kcs_draft.dwg_zoom(rectangle)
```

where **rectangle** is the **Rectangle2D** class instance, defining the axis-parallel rectangle to zoom the drawing's display to. The current zooming rectangle can be retrieved using the function:

```
rectangle = KcsRectangle2D.Rectangle2D()
⇒ kcs_draft.zoom_extent_get(rectangle)
```

where **rectangle** is an initialised **Rectangle2D** class instance, that is updated with the co-ordinates of the zooming rectangle. This function also returns this **rectangle** as the result.

- ❗ If the drawing is empty, the returned **rectangle** will be also empty. This can be tested using the call to **rectangle.IsEmpty()**.

Finally, the whole drawing can be redrawn using the function:

```
kcs_draft.dwg_repaint()
```

- ❗ Note the difference: **kcs_ui.app_window_refresh()** updates the whole application's window, but **kcs_draft.dwg_repaint()** updates only the drawing's area of this window.

5.2.7 Document references

Document references are the links to other drawings or external files. Tribon Vitesse supports this concept through the **DocumentReference** class. It handles such attributes of a document reference, as:

- Type ('unknown', 'drawing', 'file', 'vitesse', 'document')
- Document (document name)
- Description (document description)
- Purpose (for 'drawing' references, it should contain the numerical code of the drawing database – see the documentation for details)

Document references can be attached virtually to any object in Tribon. In the case of drawings, Tribon Vitesse provides the following functions handling the document references attached to the current drawing:

```
kcs_draft.document_reference_get() - get a list of document references associated with  
the current drawing  
kcs_draft.document_reference_add(docRef) - add a document reference to the current drawing  
kcs_draft.document_reference_remove(docRef) - remove a document reference from the active  
drawing
```

5.2.8 Tribon Data Management functions

If Tribon Data Management (TDM) is enabled in the Tribon system, then the **kcs_draft** module provides functions, that interact with the TDM, and manipulate some aspects of the drawings.

One of the most important features provided by TDM is the revisioning system. Various objects (including drawings) can exist in multiple revisions. Revisions can be frozen, effectively preventing the possibility of updating the given revision. When a new revision is created, it becomes current. The previous revisions are then automatically frozen. We have seen already the function **kcs_draft.dwg_open()**, able to open a given revision of a drawing. Below you can see the other functions manipulating the revisions of drawings.

```
kcs_draft.dwg_revision_freeze() - freeze the latest revision of the current drawing  
kcs_draft.dwg_revision_new() - create a new revision of the current drawing (previous one is  
'frozen')  
kcs_draft.dwg_revision_unfreeze() - unfreeze the latest revision of the current drawing
```

Additionally, TDM allows to set some properties of the current drawing, using the **CommonProperties** class instance, which includes such attributes, as: type code, planning unit, cost code, alias, description and remarks. In order to set the properties, construct the **CommonProperties** class instance, set the appropriate attributes, and call the function

```
kcs_draft.dwg_properties_set(props)
```

The status of the current drawing can be changed with the function

```
kcs_draft.dwg_status_set(statusType, statusValue)
```

The **kcs_model** module can help you provide the valid values for the arguments. **statusType** can be one of the following constants defined in this module: **kcsSTATTYPE_DESIGN**, **kcsSTATTYPE_MANUFACTURING**, **kcsSTATTYPE_ASSEMBLY**, or **kcsSTATTYPE_MATERIAL_CONTROL**. Valid values of the **statusValue** argument can be determined by calling the function **kcs_model.status_values_get(statusType)**, which returns a

dictionary containing pairs **statusValue:statusValue_String**, where **statusValue_String** describes the meaning of the given **statusValue**.

5.3 Element Handles

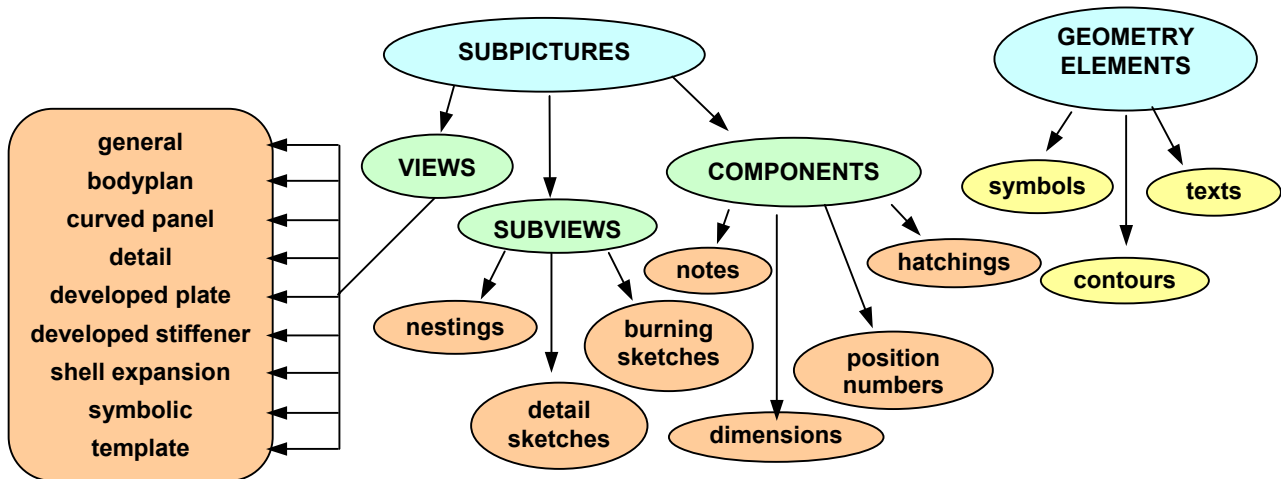
The drawing's structure is hierarchical. The topmost elements are views, which are composed from subviews. The subviews are then composed from components, which in turn are built from various geometry elements. Each drawing element is uniquely identified by a numerical id, called the element's handle. Vitesse for Drafting uses handles for identifying the drawing element, on which the operation is performed. The element handles are represented in Vitesse as instances of the **ElementHandle** class.

5.3.1 Type of elements identified by the handle

This class currently manages only one element attribute – the numerical id (the handle). The class itself does not have an information about the type of element that the handle is referring to. Fortunately, this information is contained in the object referred by the handle. For identifying the object type, Vitesse provides a set of functions, returning **1**, if the object is of the right type, or **0**, if it is not. All the functions specified below take an **ElementHandle** class instance as a parameter.

| | |
|--|--|
| <code>kcs_draft.element_is_view(handle)</code> | – a view |
| <code>kcs_draft.element_is_subpicture(handle)</code> | – a subpicture (any kind) |
| <code>kcs_draft.element_is_subview(handle)</code> | – a subview (any kind, in a view) |
| <code>kcs_draft.element_is_component(handle)</code> | – a component (any kind, in a subview) |
| <code>kcs_draft.element_is_nesting(handle)</code> | – a subpicture being a nesting subview or a component belonging to it |
| <code>kcs_draft.element_is_burning_sketch(handle)</code> | – a subpicture being a burning sketch subview or a component belonging to it |
| <code>kcs_draft.element_is_detail_sketch(handle)</code> | – a subpicture being a detail sketch subview or a component belonging to it |
| <code>kcs_draft.element_is_note(handle)</code> | – a note component |
| <code>kcs_draft.element_is_posno(handle)</code> | – a position number component |
| <code>kcs_draft.element_is_dimension(handle)</code> | – a dimensioning component |
| <code>kcs_draft.element_is_hatch(handle)</code> | – a hatching component |
| <code>kcs_draft.element_is_contour(handle)</code> | – any 2D contour element |
| <code>kcs_draft.element_is_text(handle)</code> | – a text element |
| <code>kcs_draft.element_is_symbol(handle)</code> | – a symbol element |
| <code>kcs_draft.element_is_bodyplan_view(handle)</code> | – a bodyplan view |
| <code>kcs_draft.element_is_curpanel_view(handle)</code> | – a curved panel view |
| <code>kcs_draft.element_is_detail_view(handle)</code> | – a detail view |
| <code>kcs_draft.element_is_devpla_view(handle)</code> | – a developed plate view |
| <code>kcs_draft.element_is_devsti_view(handle)</code> | – a developed stiffener view |
| <code>kcs_draft.element_is_general_view(handle)</code> | – a general view |
| <code>kcs_draft.element_is_shellx_view(handle)</code> | – a shell expansion view |
| <code>kcs_draft.element_is_symbolic_view(handle)</code> | – a symbolic view |
| <code>kcs_draft.element_is_tmpl_view(handle)</code> | – a template view |

Please note that the element categories recognised by these functions are not separate. Every note, position number, dimension, and hatching is a component. Every nesting, burning sketch, detail sketch is a subview. Every component, subview, and view is a subpicture, etc. The picture below illustrates these relations.



5.3.2 Obtaining element handles

The element handles are required for manipulating the elements. The program can obtain the handles using one of the three methods given below:

1. **By creating the object.**
Various object creating functions return always a handle to the newly created entity. If the object is created explicitly by the program, its handle is directly available at the object's creation stage.
2. **By identifying the object.**
The program can identify (detect) any object located in the neighbourhood of a point in the drawing (defined as the **Point2D** class instance). Subpictures can be also identified by its name. If the identification is successful, the handle is returned.
3. **By capturing objects in the given region.**
The program can collect the objects of the given kind, located inside (or outside) the given region. The region is defined as the **CaptureRegion2D** class instance. It is not only a simple closed contour, but can also decide, whether the objects are collected inside or outside the contour, and whether the objects crossing the contour's boundary should be taken into account, or not. If the capturing is successful, the program receives a list of handles to the objects caught by the given region.

The objects creating functions are described in various places in this chapter. The reader should find there the detailed information about how the given object is created.

The object identifying functions, however, look similar. The first group of identification functions takes a single parameter being the **Point2D** class instance (denoting the point, around which the detection is done), and returns the element's handle.

| | |
|---|------------------------------|
| <code>kcs_draft.subview_identify(point)</code> | – subviews |
| <code>kcs_draft.component_identify(point)</code> | – components |
| <code>kcs_draft.dim_identify(point)</code> | – dimensioning components |
| <code>kcs_draft.note_identify(point)</code> | – note components |
| <code>kcs_draft.posno_identify(point)</code> | – position number components |
| <code>kcs_draft.hatch_identify(point)</code> | – hatch components |
| <code>kcs_draft.geometry_identify(point)</code> | – geometries |
| <code>kcs_draft.contour_identify(point)</code> | – contours |
| <code>kcs_draft.text_identify(point)</code> | – text elements |
| <code>kcs_draft.symbol_identify(point)</code> | – symbols |
| <code>kcs_draft.point_identify(point)</code> | – points |
| <code>kcs_draft.view_identify(point, PictWinExt)</code> | – views |

The function `kcs_draft.view_identify()` accepts an optional, second argument, being an instance of the **PictWinExt** class, which modifies the method of the view identification:

1. If **PictWinExt** is initialised to **"Small"** (the default), Tribon selects first the views, for which the axis-parallel rectangle circumscribing the view, and enlarged symmetrically by 10%, contains the given indication point. Then Tribon finds the geometry, belonging to one of these views, closest to the indication point. The result is the handle of the view containing this geometry. If no view is found, whose enlarged rectangle contains the given indication point, an exception is raised.

2. If **PictWinExt** is initialised to **"Big"**, the procedure is similar to the one described above, but the rectangle is enlarged symmetrically by 50%.
3. If **PictWinExt** is initialised to **"Infinite"**, Tribon does not check the view windows at all, so all geometries in the drawing are considered.

ⓘ ***WARNING!** The settings of "Infinite" should be used sparingly, as it may take a significant amount of time to check distances to ALL geometries on a big drawing.*

The views, subviews, and components can be named. Therefore, we have three more identification functions:

```
kcs_draft.view_identify(name)      – views
kcs_draft.subview_identify(name)   – subviews
kcs_draft.component_identify(name) – components
```

and the most general one:

```
kcs_draft.element_identify(name)   – any named subpicture
```

If there are more objects with the same name, only the first one is identified.

ⓘ *It should be noted, that the function **kcs_draft.model_identify()**, introduced in section 5.5.2, does not belong to the above family of functions, because its parameters and returned value are quite different.*

The object's capturing functions also share some common properties. All take a **CaptureRegion2D** class instance as a parameter, and return a list, whose first element is the number of handles found, and the remaining ones are the **ElementHandle** class instances being the handles to the captured objects.

ⓘ *When no object is captured, these functions raise an exception with **kcs_draft.error** set to 'kcs_NotFound', instead of returning a single-element list [0] indicating that the number of captured objects is **ZERO**.*

```
kcs_draft.view_capture(region)      – views
kcs_draft.subview_capture(region)   – subviews
kcs_draft.component_capture(region) – components
kcs_draft.model_capture(region)     – model objects
kcs_draft.dim_capture(region)       – dimensioning components
kcs_draft.note_capture(region)      – note components
kcs_draft.posno_capture(region)     – position number components
kcs_draft.hatch_capture(region)     – hatch components
kcs_draft.geometry_capture(region)  – geometries
kcs_draft.contour_capture(region)   – contours
kcs_draft.text_capture(region)      – text elements
kcs_draft.symbol_capture(region)    – symbols
kcs_draft.point_capture(region)     – points
```

ⓘ *While capturing the objects, only the visible layers are considered.*

The **region** argument defines a closed contour and special properties of handling the captured object's location with respect to the contour's boundary. The most important methods are listed below:

```
region.SetRectangle(rect)           – defines a rectangular region
region.SetContour(cont)             – defines an arbitrary closed contour as the region's boundary
region.SetInside()                  – objects INSIDE the region will be captured
region.SetOutside()                – objects OUTSIDE the region will be captured
region.SetCut()                    – objects crossing the region's boundary WILL be captured
region.SetNoCut()                  – objects crossing the region's boundary WILL NOT be captured
```

Some other functions also return element handles (e.g. **kcs_draft.model_handle_get()**). See the appropriate definition in the User's Guide for details.

See the script 'Example 8.py' located in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project, which contains an example of capturing geometries.

5.3.3 Retrieving element's information from the handle

Various kinds of drawing elements are represented in Vitesse by the appropriate classes. The following functions return the information about the element referred to by a handle passed as an argument. The information is stored in the attributes of the appropriate class instance representing the investigated element, and additionally, the class instance is returned as the result.

```
... #handle is the handle to the model, its part or geometry element
model = KcsModel.Model() #initialised Model class instance
⇒ kcs_draft.model_properties_get(handle, model)
   kcs_ui.message_confirm("Detected %s '%s'" % (model.Type, model.Name))
```

The function `kcs_draft.model_properties_get()` stores the model information in the `model` parameter. If `handle` refers to the model's part, you may use also the attributes `model.PartType` and `model.PartId`. An exception is raised, if the given handle does not identify a model object.

Exercise 6: Capturing model objects

Basing on Example 8, modify the script, so that it captures model objects instead of geometries. Then display a list of the captured model objects (include the model type, name, and reflection setting).

Hint: Use the function `kcs_ui.string_select()` to display the list of model objects.

The solution to this exercise can be found in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project.

The next function returns the contour's definition from a handle. It will handle any kind of geometries, that can be expressed as contours (e.g. straight line segments, arcs, circles, rectangles, squares, polylines).

```
... #handle is the handle to the contour (any kind)
p = KcsPoint2D.Point2D() #dummy point - an existing one can be used
contour = KcsContour2D.Contour2D(p) #initialised class instance
⇒ kcs_draft.contour_properties_get(handle, contour)
```

The function `kcs_draft.contour_properties_get()` stores the contour information in the `contour` parameter, which has to be initialised `Contour2D` class instance.

See the script 'Example 9.py' located in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project, which contains an example of handling the contour's information retrieved from a handle.

The function `kcs_draft.text_properties_get()` stores the text attributes into the `Text` class instance,

```
... #handle is the handle to the text element
text = KcsText.Text() #initialised Text class instance
⇒ kcs_draft.text_properties_get(handle, text)
   theString = text.GetString() #the contents of the text element
   point = text.GetPosition()   #the location of the text element
```

In a similar way, we can get symbol properties, using the function `kcs_draft.symbol_properties_get()`. This function uses the `Symbol` class instance to store the information.

```
... #handle is the handle to the symbol element
symb = KcsSymbol.Symbol() #initialised Symbol class instance
⇒ kcs_draft.symbol_properties_get(handle, symb)
   fontId, symbId = symb.GetFontId(), symb.GetSymbolId()
```

The `kcs_draft` module contains more element handling functions using element handles described here. They are introduced in section 5.12.

5.4 View Functions

The **kcs_draft** module provides a set of functions dealing with model views. Additional **import** statements may be necessary if the selected function uses the Vitesse class instances as parameters (Vector3D, Point3D, and Point2D):

```
import KcsVector3D
import KcsPoint3D
import KcsPoint2D
```

5.4.1 Creating or identifying the view

All views in the drawing are uniquely identified by a handle (**ElementHandle** class instance). The handle can be obtained either when the Vitesse program itself creates the view, or by identifying an existing view by its name, or by an identification point.

```
U = KcsVector3D.Vector3D(0, -1, 0) #frame view, looking fore
V = KcsVector3D.Vector3D(0, 0, 1)
⇒ handle = kcs_draft.view_new('FR40', U, V) #create the view 'FR40'
```

In the above example, the function **kcs_draft.view_new()** returns a handle to a new view 'FR40', with the U and V vectors defining the projection of a frame view with the viewer looking fore. The U and V vectors are optional arguments. If omitted, the X and Y axis direction vectors will be used. The new view is always created in the scale 1:1, with the origin at the position (0, 0) in the drawing form. Usually the view is then scaled and displaced appropriately (see section 5.4.2).

① *U and V vectors are located on the view's plane. U points to the right, V points up. Their cross-product $U \times V$ indicates the direction towards the viewer. These vectors should be expressed in the global co-ordinate system.*

If you want to work on a view, that already exists in the drawing, you need to obtain its handle by using the **kcs_draft.view_identify()** or **kcs_draft.view_capture()** functions, described in section 5.3.2.

All the remaining view functions require a proper view handle as their first parameter. If the passed view handle is not valid, an exception will be raised.

When creating views, we have to give 3D vectors U and V, defining the view's orientation. The table below lists the vector co-ordinates for some commonly used views. The W vector is the cross product of U and V.

| | | | |
|----------------|------------------------------------|--|--|
| Y-Z plane view | $U = (0, 1, 0)$ | $V = (0, 0, 1)$ | $W = (1, 0, 0)$ |
| X-Z plane view | $U = (1, 0, 0)$ | $V = (0, 0, 1)$ | $W = (0, -1, 0)$ |
| Y-X plane view | $U = (0, 1, 0)$ | $V = (-1, 0, 0)$ | $W = (0, 0, 1)$ |
| ISO view | $U = (1/\sqrt{2}, -1/\sqrt{2}, 0)$ | $V = (1/\sqrt{6}, 1/\sqrt{6}, 2/\sqrt{6})$ | $W = (-1/\sqrt{3}, -1/\sqrt{3}, 1/\sqrt{3})$ |

① *The view functions often use 2D points and 2D vectors. They are always expressed in the drawing form co-ordinate system (scale 1:1, origin at lower left corner of the drawing form, U-axis pointing to the right, V-axis pointing up), not in the given view's co-ordinate system!*

5.4.2 Transforming the view

The views can be displaced, scaled, reflected and rotated. Below please find an example of displacing and scaling a view, usually performed after creating a view and adding some contents. **centre** is the view's current centre point (Point2D), **target** is the destination point (Point2D), where **centre** should be displaced, **factor** is the calculated relative scaling factor, and **handle** – the view's handle.

```
⇒ kcs_draft.view_scale(handle, factor, centre) #scale the view
vector = KcsVector2D.Vector2D() #build the displacement vector
vector.SetFromPoints(centre, target) #from centre to target
⇒ kcs_draft.view_move(handle, vector)
```

① *The **centre** argument is optional. If omitted, defaults to the centre of the axis-parallel rectangle circumscribing the view's contents.*

Exercise 7: Transforming a view

Indicate a view (identify it), then indicate a rectangle (identify it, and get contour properties). Then transform the view so, that it fits inside the rectangle. For simplicity, assume, that the indicated contour IS a rectangle – do not check the contour's shape, just use the co-ordinates.

Hint: Use the function `kcs_draft.element_extent_get()` (see section 5.12), to obtain the location and size of the view. Then use this information to calculate the proper scaling factor and displacement vector.

The solution to this exercise can be found in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project.

The reflection of the view with respect to the U or V axes and rotation of the view can be done with the functions shown below:

```
kcs_draft.view_reflect(handle, reflect, Centre), and  
kcs_draft.view_rotate(handle, angle, Centre)
```

The **reflect** argument can be either 1 (reflection in the U-axis passing through the centre), or 2 (reflection in the V-axis passing through the centre). The rotation **angle** should be expressed in degrees.

① Again, the **centre** argument is optional. If omitted, defaults to the centre of the axis-parallel rectangle circumscribing the view's contents.

5.4.3 View's projection

The final result of a sequence of view transformations is defined as a Transformation3D object, which can be obtained using the function `kcs_draft.view_projection_get()`

```
... #handle identifies the view  
t = KcsTransformation3D.Transformation3D()  
⇒ kcs_draft.view_projection_get(handle, t) #Get the transformation  
origin = KcsPoint3D.Point3D(t.matrix41, t.matrix42, t.matrix43)
```

In the above example, we additionally derive the location of the origin of the view's co-ordinate system. In a similar manner, we can obtain the U, V, and W vectors of this co-ordinate system.

It is possible to change the projection of an existing view by providing new U and V vectors, and calling the function `kcs_draft.view_projection_set()`. Similar consideration should be taken, regarding these vectors, like for the function `kcs_draft.view_new()`.

```
... #U and V define the original view, identified by handle  
U.BlankProduct(-1.0) #U becomes -U, we change the looking direction  
⇒ kcs_draft.view_projection_set(handle, U, V, 0)
```

The last, optional argument is an integer, determining the draw codes to be used. The value **0** means, that draw codes stored in the model subviews should be used. The value **1** means, that default draw codes should be used. If omitted, defaults to **1**.

For symbolic views (see section 5.4.4) we have another function for retrieving the model's transformation matrix: `kcs_draft.view_symbolic_model_tra()`, which is used exactly in the same way, as the function `kcs_draft.view_projection_get()`.

5.4.4 Symbolic views

It is possible to create symbolic views in Vitesse

```
locPoint = KcsPoint3D.Point3D(20000, 0, 5000) #origin of the projection  
uVector = KcsVector3D.Vector3D(0, 1, 0)  
vVector = KcsVector3D.Vector3D(0, 0, 1) #U and V vectors (projection)  
forward = 500.0  
backward = 100.0 #backward and forward slice depths
```

```

boxOrigin = KcsPoint3D.Point3D(19000, -5000, 2000) #restriction box
box = KcsBox.Box(boxOrigin, uVector, vVector, 10000, 6000, 2000)
⇒ handle = kcs_draft.view_symbolic_new(name, locPoint, uVector, vVector, \
                                       forward, backward, box)

```

The above example creates the symbolic view in the X plane with the cutting plate passing through **locPoint**, slice depths **forward** and **backward**, and the given restriction **box** (X: 19000 – 21000, Y: -5000 – 5000, Z: 2000 – 8000).

The function **kcs_draft.view_symbolic_new()** creates a new symbolic view in the drawing, and returns its handle as the **ElementHandle** class instance. Model items or their parts located outside the **box** will not be drawn, when using the function **kcs_draft.model_draw()**. The box location and dimensions are specified in the ship's co-ordinate system. The U and V vectors must not be parallel, and their cross-product ($U \times V$) defines the direction towards the viewer. If the current drawing already contains a view with the same name, as given by the **name** parameter (if not empty), an exception is raised.

❗ *If the symbolic view is created, as shown above, the view is not drawn yet. You need to draw explicitly model items on this symbolic view to fill up the view with the right contents.*

You can achieve a better control on the creation of the symbolic view by using the instance of the **SymbolicView** class (**KcsInterpretationObject** module) instead of providing all the arguments shown above.

```

sv = KcsInterpretationObject.SymbolicView()
sv.SetViewName(name)           #choose a view name
sv.SetPlaneByX(20000)          #cutting plane at X=20000
sv.SetAutomaticSelection(1)     #request automatic object selection!
sv.SetShellCurves(sv.CURVE_CUT) #request shell curves to be cut
⇒ handle = kcs_draft.view_symbolic_new(sv)

```

❗ *After creation, the symbolic view is not shown in the drawing. Add the following fragment of code to make it appear in the drawing.*

```

transf = KcsTransformation2D.Transformation2D() #identity transformation
kcs_draft.element_transform(handle, transf) #apply the transformation

```

Applying an identity transformation to a drawing element makes it appear in the drawing (the element is redrawn). See section 5.12 for a description of the function **kcs_draft.element_transform()**.

It is possible to obtain the values of the slice depths for a symbolic view (also for a sliced model view) using the function

```

sliceDepthInfo = kcs_draft.view_sliceddepth_get(handle)

```

The function returns a tuple, having the status code as the first item;

```

(0, )           – no slice depth available, the view is not sliced
(1, depth)      – sliced model view with the given slice depth
(2, forward, backward) – symbolic view with the given forward and backward slice depths

```

The slicing planes can be retrieved using the function **kcs_draft.view_slice_planes_get()**

```

... #handle identifies a symbolic view (or sliced model view)
⇒ plane1, plane2 = kcs_draft.view_slice_planes_get(handle)
   #Get the origin points and vectors normal to the planes
   origin1, normal1 = plane1.Point, plane1.Normal
   origin2, normal2 = plane2.Point, plane2.Normal

```

The result is a tuple of two **Plane3D** class instances, that have the information about the plane (the origin point and the normal vector)

5.4.5 Miscellaneous

By calling the function **kcs_draft.view_hl_remove()** with the **view handle** as an argument, it is possible to remove hidden lines on the view.

The function **kcs_draft.point_transform()**, mentioned already in section 3.7 (page 43), translates the 3D point in a view to the corresponding 2D point in the drawing's co-ordinate system.

```

... #get modelPoint (Point3D)
dwgPoint = KcsPoint2D.Point2D()
⇒ kcs_draft.point_transform(handle, modelPoint, dwgPoint)

```

The function converts the given **modelPoint** to the UV co-ordinates of the view identified by **handle**, storing the result in the **dwgPoint** parameter. Additionally it returns **dwgPoint** as the result.

The function **kcs_draft.view_restriction_area_get()** returns a **Rectangle2D** class instance defining the restriction area declared in the drawing form for the given **view**, whose handle is passed as an argument. If the drawing form does not set restriction area for the given **view**, the function raises an exception. The rectangle is expressed in the drawing's co-ordinate system.

5.5 Model Handling Functions

In all model handling functions a Model class instances are used. This requires the presence of the statement **import KcsModel** in the program. Additional **import** statements may be necessary if the program uses functions having other 'geometrical' Vitesse classes as parameters.

5.5.1 Drawing model objects

Model objects can be drawn on the views using the function **kcs_draft.model_draw()**. There are two variants of its usage:

```

model = KcsModel.Model("plane panel", "ES123-AY012")
⇒ kcs_draft.model_draw(model, viewHandle)
and
assCrit = KcsModelDrawAssyCriteria.ModelDrawAssyCriteria('-B01')
assCrit.SetRecursive(1) #Set recursive lookup mode
assCrit.EnableModelType('CurvedPanel', 0) #disable curved panels
⇒ kcs_draft.model_draw(assCrit, viewHandle)

```

In the first variant, a single **model** object or its part (**Model** class instance) is drawn. In order to draw the whole model, provide the **Model** class instance with the correct **Type** and **Name** attributes, and with **PartId** set to **0** (default). If the **PartId** attribute is not zero, only the given part of the model is drawn.

The second variant draws all model objects fulfilling the assembly criteria specified by the provided **ModelDrawAssyCriteria** class instance. If **viewHandle** argument is provided, drawing occurs on the given view only. If **viewHandle** is omitted, the selected models are drawn on all model views,

i In the case of plane panels, the parts like the holes and cutouts cannot be drawn separately.

5.5.2 Identifying and collecting model objects

The function **kcs_draft.model_identify()** looks around the given **idPoint**, and identifies the closest model object, filling the provided **model** argument with the information about the identified model object.

```

... #idPoint is some point in the drawing
model = KcsModel.Model()
⇒ res = kcs_draft.model_identify(idPoint, model)
kcs_ui.message_confirm("%s '%s' found!" % (model.Type, model.Name))

```

This function will raise an exception, if it is unable to find a model object in the neighbourhood. The model argument can define the expected model type, and then the function will look for the given type of models only.

i The function **getModel()** in our **VTBasic** module combines an indication of the model with its identification. Additionally, the identified model can be confirmed by the user before continuing.

The function **kcs_draft.model_identify()** returns the tuple containing:

| | | |
|------------------|---------|--|
| model | Model | type & name of the model item (Model class instance) |
| subView | integer | handle to the model subview |
| component | integer | handle to the model component (part ID = model.PartId) |

These handles can be used, for example for highlighting the given model or its component before asking for confirmation, removing the given model from the drawing, etc.

Exercise 8: Displaying the weight of indicated structures

Indicate a structure object, extract its weight and display it in a message window.

The solution to this exercise can be found in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project.

The function `kcs_draft.model_capture()`, described in section 5.3.2, returns the list of handles to model objects captured in the **region** given as an argument.

i Please note, that this function returns a list of handles, not **Model** class instances!

Fortunately, the program can get an information about each handle in the list by calling the function `kcs_draft.model_properties_get()`, described in section 5.3.3

Tribon Vitesse provides also the reverse operation. It is possible to get handles of all subviews (components) containing the given model (its part) by calling the function `kcs_draft.model_handle_get()`, providing the **model** class instance as an argument. If **model.PartId** is zero, the function returns the list of all handles to the subviews containing the given model. If **model.PartId** is not zero, the resulting list contains all handles to the component subpictures containing the given part. The example below removes the given **model** from all views, using the function `kcs_draft.element_delete()`, described in details in section 5.12:

```
... #model describes a model object to be deleted
⇒ handleList = kcs_draft.model_handle_get(model)
   for handle in handleList:
       kcs_draft.element_delete(handle)
```

5.5.3 Deleting model objects

Fortunately, there is a simpler way to remove model objects from the drawing. In order to remove the given model object from the view (or from the whole drawing), the function `kcs_draft.model_delete()` should be used. The above example could be replaced by a single line:

```
kcs_draft.model_delete(model)
```

If you want to remove a model object from the given view only, provide the view handle as the second argument - then the other occurrences of the given model object will not be removed.:

```
kcs_draft.model_delete(model, viewHandle)
```

5.5.4 Modifying model object's modal properties

Each subview containing a model object has the following properties: the colour, and the layer. They can be changed using the functions

```
kcs_draft.model_colour_set(model, colour, viewHandle), and
kcs_draft.model_layer_set(model, layer, viewHandle)
```

where **model** is a Model class instance, describing the model object itself, **colour** is the Colour class instance, describing its new colour, **layer** is the Layer class instance describing its new layer, and **viewHandle** is the handle to the view, on which the given update should be made. **viewHandle** is an optional argument, and the change will be made to ALL views, if this argument is omitted.

5.5.5 Tribon Data Management functions

Model objects can now have revisions, if TDM is enabled. Tribon Vitesse handles them with the class **ModelObjectRevision** and the following functions:

```
kcs_draft.model_object_revision_save(model, revision, viewHandle, \
                                     saveSpools)
```

The function saves the given **revision** of the model object **model** on the view identified by **viewHandle**. Additionally, if **model** refers to a pipe or ventilation object, we can request saving the spools together with the model by setting **saveSpools** to **1**. The value of **0** turns off the saving of spools.

```
revisionList = []
⇒ kcs_draft.model_object_revision_get(model, viewHandle, revisionList)
```

The function fills up the **revisionList** with available revisions of the given **model** in the view identified by **viewHandle**.

```
kcs_draft.model_object_revision_set(model, revision, viewHandle)
```

The function displays the given **revision** of the **model** in the view identified by **viewHandle**.

The **ModelObjectRevision** class holds the information about: the revision name, its remark, creator, creation date, modifier, and modification date.

Exercise 9: Create model views

Ask for a plane panel name and create a new drawing named 'PANEL_XXX' (where XXX stands for the panel name) using the drawing form 'A1'. In this drawing, create four views: 'X' – in Y-Z plane, 'Y' – in X-Z plane, 'Z' – in Y-X plane, and 'ISO' – 3D isometric view (see the note on page 64), and draw the panel on each of them. Scale and translate the views so that they fit inside the quarters of the A1 drawing frame (within the rectangle between (0,0) and (810,580)). When creating the drawing, make sure that any existing drawing with the same name is deleted from the databank, and that the current drawing (if any) is closed.

Hints: The function **math.sqrt()** in the **math** module returns the square root of the argument. Use it for defining the U and V vectors for the isometric view.

Use also the function **kcs_draft.element_extent_get()** returning the 2D rectangle that the given view occupies in the drawing. It will help you determine the proper scale factor and displacement vector when locating the views in the quarters of the drawing frame.

If you set **model.ReflCode** to **2**, both the panel and its starboard image will be drawn, if the panel is symmetric.

The solution to this exercise can be found in the 'Vitesse Basic Training' folder under SB_PYTHON in the training project.

5.6 Basic Geometry Entities

The functions in this section provide the means of creating the basic geometry elements in the current drawing, highlighting them, and setting the general modal properties of these elements. The **kcs_draft** Vitesse module provides functions for setting and reading the modal properties such as: colour, line type, layer. Please note, that the colour and line type are represented by Python classes, which require the corresponding Python module to be imported in the program...

```
import KcsColour
```

...for dealing with the colours, and...

```
import KcsLineType
```

...for dealing with the line types.

Each geometry element is defined in Python language as a Python class. Before using any of these functions, the corresponding **import** statement including the class definition file must be used in the program. Every geometry element, when created, obtains an associated handle (ElementHandle class instance), which is used then for all manipulation of the given element. All geometry elements are drawn using current values of the modal properties.

5.6.1 Modal properties

The modal properties are handled by a family of get/set functions, reading the current values and setting new ones. The example below manages the modal colour:

```

    orgColour = KcsColour.Colour()
⇒ kcs_draft.colour_get(orgColour) #store the original colour
    kcs_ui.message_confirm("Original colour: " + orgColour.GetName())
    newColour = KcsColour.Colour("NavyBlue")
⇒ kcs_draft.colour_set(newColour) #set the new modal colour

```

and this one – the modal line type

```

    orgLinetype = KcsLinetype.Linetype()
⇒ kcs_draft.linetype_get(orgLinetype) #store the original line type
    kcs_ui.message_confirm("Original line type: " + orgLinetype.Name())
    newLinetype = KcsLinetype.Linetype("DashedWide")
⇒ kcs_draft.linetype_set(newLinetype) #set the new modal line type

```

The functions below manage the display settings of various line types (line thickness, patterns, etc.). They use an instance of the **LinetypeDisplaySettings** class.

```

⇒ settings = kcs_draft.linetype_display_settings_get()
    thinWidth = settings.GetThinWidth() #width of the THIN line
    settings.SetThinWidth(2*thinWidth) #make it twice as wide
⇒ kcs_draft.linetype_display_settings_set(settings) #apply the settings

```

The last modal attribute is the layer. The example below shows the functions managing the modal layer.

```

⇒ orgLayer = kcs_draft.layer_get() #store the original modal layer number
    kcs_ui.message_confirm("Original layer: %d" % orgLayer)
⇒ kcs_draft.layer_set(1000) #set a new modal layer number

```

5.6.2 Creation of basic geometry entities

The functions listed below draw the basic geometry entities given as an argument, and return the handle to the created entity as the **ElementHandle** class instance. The entity is PERMANENT, i.e. it will stay in the drawing until removed explicitly. All geometric entities are drawn using current values of the general modal properties.

| Function | geometric entities | Vitesse class |
|--|--------------------|---------------|
| <code>kcs_draft.arc_new(arc)</code> | arcs | Arc2D |
| <code>kcs_draft.circle_new(arc)</code> | circles | Circle2D |
| <code>kcs_draft.conic_new(conic)</code> | conic segments | Conic2D |
| <code>kcs_draft.contour_new(contour)</code> | contours | Contour2D |
| <code>kcs_draft.ellipse_new(ellipse)</code> | ellipses | Ellipse2D |
| <code>kcs_draft.line_new(line)</code> | line segment | Rline2D |
| <code>kcs_draft.point_new(point)</code> | points | Point2D |
| <code>kcs_draft.rectangle_new(rect, radius)</code> | rectangles | Rectangle2D |
| <code>kcs_draft.spline_new(points)</code> | spline curves | Polygon2D |

The **kcs_draft.arc_new()** function raises an exception and sets **kcs_draft.error** to 'kcs_AmplitudeTooBig', if the amplitude defined in the **Arc2D** class instance is too big (greater than half of the distance between the arc's endpoints). The same does the function **kcs_draft.contour_new()**, when one of the contour arc segments has an amplitude too big.

The **kcs_draft.rectangle_new()** function draws the axis-parallel rectangle, optionally with rounded corners, if the **radius** argument is provided.

The **kcs_draft.spline_new()** function draws the spline curve without any tangent restrictions at the node points.

☞ See 'Example10.py' for a demonstration of creating a circle basing on user-indicated centre point and the radius. If the radius is not valid, the program asks for a periphery point, and derives the radius from the distance between the centre and the periphery point.

Thanks to the use of **kcs_draft.dwg_current()** function in 'Example 10.py', we are sure that there is a current drawing. Since we validate the centre, and the radius, we should not expect any exceptions to be raised. That's why this example is free of **try ... except ...** constructions.

Note that the essential part of the code (creating the circle) consists only of three lines following the comment '# construct the circle'. The rest deals with the user interface, and preparation of safe data for the circle creation. This is a necessary feature if the program wants to be interactive and safe from any errors resulting from incorrect user input. Of course, the program size will be significantly bigger due to the added user interface and safeguarding code.

i The circle radius, if keyed in by the user, does NOT take into account the scale factor of the given view, but uses the scale 1:1. The program must handle the proper scaling by itself.

All created entities are attached to the **current subpicture**. See the functions in section 5.11, which let you create subpictures and set the current subpicture for adding new geometry entities.

5.6.3 Entity properties

The geometry elements, whose creation is described in the last section, are drawn using the current values of modal attributes: colour, line type, and layer. Fortunately, there is a way to retrieve this information from already existing elements, and update it. In the examples below, **handle** refers to the drawing element, whose properties are manipulated. This is how we manipulate element layer ...

```
layer = KcsLayer.Layer()
⇒ kcs_draft.element_layer_get(handle, layer)
newLayer = KcsLayer.Layer(1000) #new layer
⇒ kcs_draft.element_layer_set(handle, newLayer)
```

... colour ...

```
colour = KcsColour.Colour()
⇒ kcs_draft.element_colour_get(handle, colour)
newColour = KcsColour.Colour("Red") #new colour
⇒ kcs_draft.element_colour_set(handle, newColour)
```

... and line type ...

```
linetype = KcsLinetype.Linetype()
⇒ kcs_draft.element_linetype_get(handle, linetype)
newLinetype = KcsLinetype.Linetype("DashedWide") #new linetype
⇒ kcs_draft.element_linetype_set(handle, newLinetype)
```

i The functions given here can be used not only with handles referring to the basic geometry entities, but also to other drawing elements, like: texts, symbols, notes, position number, hatchings, dimensions, views, subviews, components, etc.

5.6.4 Highlighting elements

The Tribon system uses highlighting of the drawing entities for indication of the selected entity (transformations, deletions, etc.). The same possibility is available to the Vitesse programmer through the highlighting functions described below...

| Function | geometric entities | Vitesse class |
|---|--------------------|---------------|
| kcs_draft.arc_highlight(object) | arcs | Arc2D |
| kcs_draft.circle_highlight(object) | circles | Circle2D |
| kcs_draft.conic_highlight(object) | conic segments | Conic2D |
| kcs_draft.contour_highlight(object) | contours | Contour2D |
| kcs_draft.ellipse_highlight(object) | ellipses | Ellipse2D |
| kcs_draft.line_highlight(object) | line segments | RLine2D |
| kcs_draft.point_highlight(object, type) | points | Point2D |
| kcs_draft.rectangle_highlight(object) | rectangles | Rectangle2D |
| kcs_draft.spline_highlight(object) | splines | Polygon2D |

where **object** is an instance of the corresponding geometry class, containing the definition of the entity to draw highlighted. All the above functions draw a new highlighted entity and return a special identifier of the entity (this is NOT an **ElementHandle** class instance!). This identifier should be saved, because it is required, when highlighting is turned off. Turning off the highlight removes the entity from the drawing.

① *Note, that contrary to the `kcs_draft.XXX_new()` functions (see section 5.6.2), which draw PERMANENT basic geometry entities, the highlighting functions draw TEMPORARY basic geometry entities, which disappear automatically from the drawing, when their highlight is turned off.*

The function `kcs_draft.point_highlight()` accepts an optional second argument, which determines the point's shape: the value of `1` causes a small cross to be drawn, whereas other values (default) draw the point as a small circle.

Additionally there is a function, which highlights arbitrary drawing's elements, given their handles, not definition. The argument can be either a single **ElementHandle** class instance, or a list of such elements.

```
kcs_draft.element_highlight(handle), or  
kcs_draft.element_highlight(handleList)
```

The examples below highlights a model object identified by the `kcs_draft.model_identify()` function, which provides the handle to the subpicture of the indicated model object.

```
... #idPoint is a point in the drawing, model is a Model class instance  
res = kcs_draft.model_identify(idPoint, model)  
⇒ ident = kcs_draft.element_highlight(res[1]) #highlight model object  
... #ask the user for confirmation of the indicated model object  
⇒ kcs_draft.highlight_off(ident) #turn off the highlight
```

This function highlights an already existing entity identified by its handle (**ElementHandle** class instance), and returns the highlight's identifier (integer), which must be used for turning the highlight off. In this case, however, turning off the highlight does not remove the entity from the drawing, because the entity had existed already before it has been highlighted, but rather it causes the entity to be redrawn in its normal colours.

① *The function `kcs_draft.element_highlight()` can be used for highlighting not only basic geometric entities, but in fact every drawing item identifiable by a handle (model items and their parts, views, texts, symbols, notes, position numbers, hatch patterns, etc.)*

When highlighting is to be turned off, there is a general function to do this.

```
kcs_draft.highlight_off(highlightIdent)
```

This function turns off the highlighting of the given entity defined by its highlight's identifier. To turn the highlight from ALL highlighted items, call

```
kcs_draft.highlight_off(0)
```

The entities highlighted using the function `kcs_draft.element_highlight()` are simply redrawn in their normal colours, whereas all other highlighted entities disappear from the drawing.

① *The functions from the `XXX_highlight()` family return standard integer numbers as highlight's identifiers, not **ElementHandle** class instances!*

5.7 Texts

The Vitesse program can easily create text in the current drawing with full control over such text modal properties as height, rotation and slanting angle, aspect ratio, text & vector font numbers, and interline spacing factor. The current general modal properties like the colour, line type, and layer are also taken into account (see section 5.6.1)

5.7.1 Modal properties

The text drawing elements have the following modal properties: text height, rotation angle, slanting angle, aspect ratio, font number (both ASCII and Vector), and or multi-line texts – the inter-line spacing. For every text modal attribute, there is a pair of functions: one setting the new value of the attribute (**set**), and another – reading the current value of the attribute (**get**).

```
⇒ height = kcs_draft.text_height_get()  
kcs_draft.text_height_set(4.5) #text height as a real number  
⇒ rotation = kcs_draft.text_rotation_get()  
kcs_draft.text_rotation_set(30) #text rotation angle in degrees
```

```

⇒ aspectRatio = kcs_draft.text_aspect_get()
   kcs_draft.text_aspect_set(0.8)      #set narrower text
⇒ slant = kcs_draft.text_slant_get()
   kcs_draft.text_slant_set(70)        #set slanted text (70 degrees)
⇒ asciiFont = kcs_draft.text_ascii_font_get()
   kcs_draft.text_ascii_font_set(77)   #set user-defined ascii text font
⇒ vectorFont = kcs_draft.text_vector_font_get()
   kcs_draft.text_vector_font_set(77)  #set user-defined vector text font
⇒ ILSP = kcs_draft.text_ilsp_get()
   kcs_draft.text_ilsp_set(1.5)        #set inter-line spacing factor

```

Text height is given in the 1:1 scale (text size on the unscaled print-out). Angles are given in degrees. Aspect ratio of 1.0 means the standard proportions of the letters. Larger values mean the wider text, lower values – narrower text. Vector fonts are used in multi-byte languages (e.g. Japanese). Inter-line spacing factor values can be negative – then the consecutive lines of text will be positioned in the opposite direction (upwards).

❶ To select a TrueType font (e.g. Arial) for drawing text elements, use the function `kcs_draft.default_value_set()`

and set the **TEXT_FONT** Drafting default to the proper font name (see section 5.10).

Note, that the text height represents the height of the capital letter (e.g. 'M'). This does not take into account the additional space occupied by characters drawn below the text baseline (e.g. 'p', 'j', 'q', etc.).

5.7.2 Creation of texts

In a similar way as the basic geometric entities, the Vitesse program can create text in the drawing. The text is created using the current values of the modal text properties. The text string may contain the '\n' characters, denoting a line break. This allows multi-line texts also to be handled by the program.

```

   s = "This is an example\nof a multi-line\ntext."
   point = KcsPoint2D.Point2D(150, 100)
⇒ handle = kcs_draft.text_new(s, point)
or
   text = KcsText.Text("This is an example\nof a multi-line\ntext.")
   text.SetPosition(KcsPoint2D.Point2D(150, 100))
⇒ handle = kcs_draft.text_new(text) #Use of the Text class instance

```

❶ The **Text** class handles the attributes of the given text drawing element, including the string and position on the drawing.

The `kcs_draft.text_new()` function creates a new text element in the current drawing, and returns its handle as the **ElementHandle** class instance. In order to use this function, the program must import the **KcsPoint2D** module.

The use of the **Text** class allows us to obtain easily the length of the text. This can be useful for determining the column width of a table placed in the drawing, sizing the rectangle to be drawn around the text, etc.

```

   text = KcsText.Text("Panel name") #A text element
   p1 = KcsPoint2D.Point2D(150,100)
   text.SetPosition(p1)               #placed at (150,100)
⇒ textLen = kcs_draft.text_length(text)
   p2 = KcsPoint2D.Point2D(p1.X + textLen, p1.Y + text.GetHeight() + 1)
   #Rectangle is drawn around the text
   handle = kcs_draft.rectangle_new(KcsRectangle2D.Rectangle2D(p1, p2))

```

❶ For multi-line text elements, the function `kcs_draft.text_length()` returns the length of the **FIRST** text line (up to the first '\n'), also when one of the next text lines is longer than the first. For computing the width of the multi-line text element we have to split the text into single lines, calculate their lengths, and select the biggest one.

If your drawing contains the drawing rules (e.g. \$2000), it is possible to set the text for these rule text elements, as shown below:

```

   #Set or update the "Drawn by" drafting rule ($2102)
⇒ handle = kcs_draft.rule_text_new("John Smith", 2102)

```

5.8 Symbols

The Vitesse program is able to put symbols in the drawing, and control their height and rotation angle.

5.8.1 Modal properties

The examples below demonstrate, how to control the symbol's height ...

```
symbHeight = kcs_draft.symbol_height_get()
⇒ kcs_draft.symbol_height_set(2.0*symbHeight)
```

... and rotation angle ...

```
symbRotation = kcs_draft.symbol_rotation_get()
⇒ kcs_draft.symbol_rotation_set(30.0)
```

Symbol height is the symbol size on the printout (unscaled). Rotation angle is expressed in degrees.

5.8.2 Creation of symbols

Tribon Vitesse provides the function `kcs_draft.symbol_new()`, which draws a new symbol in the drawing and returns its handle. Possible ways of using this function are shown below

```
point = KcsPoint2D.Point2D(100, 100) #symbol's position
fontNo, symbolNo = 21, 1
⇒ handle = kcs_draft.symbol_new(fontNo, symbolNo, point)
```

or

```
fontNo, symbolNo = 21, 1
symbol = KcsSymbol.Symbol(fontNo, symbolNo)
symbol.SetPosition(KcsPoint2D.Point2D(100, 100)) #Set position
symbol.SetRotation(30) #Set rotation angle (... and other attributes)
⇒ handle = kcs_draft.symbol_new(symbol)
```

The first variant uses the current settings of modal attributes, whereas the second one uses the attributes stored in the Symbol class instance (rotation angle, height, visibility, detectability, reflection, etc.)

5.9 Drawing Components

Tribon supports creating four kinds of drawing components: hatchings, notes, position numbers, and dimensions. All these components are created as level 3 subpictures attached to the closest subview.

5.9.1 Hatching

The modal hatch pattern can be defined in three ways, as in any Tribon application:

- as one of the standard hatch patterns,
- as the angle and distance between the hatch pattern lines,
- as the hatch pattern from the Standard Hatch Pattern Book

All these methods of defining the modal hatch pattern are supported by the functions in the Vitesse module `kcs_draft`. The selected pattern is then used for adding hatching to the contours, which requires importing of the `KcsContour2D` Vitesse module since the class `Contour2D` is used.

5.9.1.1 Modal properties

The only modal property specific to hatching elements is the hatching pattern. Of course, the hatching functions take also the current modal colour, line type and layer into consideration. The function shown below sets one of the standard hatch patterns

```
kcs_draft.std_hatch_pattern_set(Type)
```

where **Type** takes one of the following values:

- 1 – single line hatching (HATCH_ANG_PAT1)
- 2 – single line hatching (HATCH_ANG_PAT2)
- 3 – cross hatching (HATCH_ANG_CROSS)

If other inclination angles or distances between the lines are required, we can use the function

```
kcs_draft.hatch_pattern_set(angle, distance)
```

where we provide proper values for the line inclination angle (in degrees) and the distance between the lines (in mm). Finally, Tribon system supports the standard book of hatching patterns, where we can store subpictures to be used as hatching patterns. Up to 8 patterns can be stored on a single page in this book. When calling the function

```
kcs_draft.userdef_hatch_pattern_set(page, detail)
```

we have to provide the proper page and pattern number from the Standard Hatch Pattern Book.

5.9.1.2 Adding hatching to the contours

Like all the geometric elements, the hatch pattern components are also recognised by the handle that is returned by the functions performing the hatching of the contours.

```
p = KcsPoint2D.Point2D(100, 100)
cont = KcsContour2D.Contour2D(p)
p.X = 300 #Move by 200 to the right
cont.AddLine(p) #add a straight segment
p.SetCoordinates(200,200)
cont.AddArc(p, -30) #add an arc segment
p.SetCoordinates(100, 100) #go back to the starting point
cont.AddArc(p, 30) #add an arc segment
kcs_draft.hatch_pattern_set(30.0, 2.0) #Set the hatching pattern
⇒ hatchHandle = kcs_draft.hatch_new(cont) #draw the hatching
```

The above example draws the hatching itself. If you want to see the contour too, just add the statement

```
contHandle = kcs_draft.contour_new(cont)
```

It is also possible to call the function **kcs_draft.hatch_new()** with **contHandle** as an argument.

```
contHandle = kcs_draft.contour_new(cont)
⇒ hatchHandle = kcs_draft.hatch_new(contHandle) #draw the hatching
```

We can remove the part of the given hatch pattern by providing original hatching handle and the island's contour or its handle.

```
p.SetCoordinates(170, 120)
island = KcsContour2D.Contour2D(p)
p1 = KcsPoint2D.Point2D(170, 160)
island.AddArc(p1, 20)
island.AddArc(p, 20)
islandHandle = kcs_draft.contour_new(island)
⇒ hatchHandle = kcs_draft.hatch_island_new(hatchHandle, island)
```

The function **kcs_draft.hatch_island_new()** removes the part of the given hatch pattern that lies inside a given island, defined by a closed 2D contour, or its handle, and returns the handle to the modified hatch pattern component as the **ElementHandle** class instance.

5.9.2 Notes and Position Numbers

The **kcs_draft** module provides functions for creating notes and position numbers, as well as for defining the modal properties of their symbols.

5.9.2.1 Modal properties

Tribon Vitesse provides functions to control the note and position number symbols

```
noteSymbol = kcs_draft.note_symbol_get()
kcs_draft.note_symbol_set(35)    #new note symbol from font 21
posnoSymbol = kcs_draft.posno_symbol_get()
kcs_draft.posno_symbol_set(65)  #new position number symbol from font 21
```

Symbols used for notes and position numbers are restricted to the system font 21. The valid note symbols are in a range 31 – 60 (or –1 to suppress the note symbol). For position numbers, symbols in the range 61-80 can be used.

```
oldHeight = kcs_draft.posno_height_get()
kcs_draft.posno_height_set(2.0*oldHeight)
```

- ❗ If the given symbol does not fall in the right range, the exception is raised. Tribon M3 provides an alternative symbol font for note and position number symbols (font #41). It will be used instead of the system font 21, if the Drafting default keywords `NOTE_SYMB_EXTENDED_RANGE` or `POSNO_SYMB_EXTENDED_RANGE` are set. Then the interval 1-400 is foreseen for note symbols, 401-500 for note "start" symbols, 501-600 for note "end" symbols, and 601-999 for position number symbols.

5.9.2.2 Creating notes and position numbers

The functions given below create notes and position numbers using current values of the modal properties. Since instances of the Polygon2D class are used, the program must contain the statement `import KcsPolygon2D` in order to use these functions.

```
p = KcsPoint2D.Point2D(100,100)
refLine = KcsPolygon2D.Polygon2D(p)
p.SetCoordinates(120,130)
refLine.AddPoint(p)
p.X += 10
refLine.AddPoint(p)
⇒ handle = kcs_draft.note_new("Note text", refLine)
```

In order to put a position number instead of a note, just replace the last line with the following one ...

```
handle = kcs_draft.posno_new("31", refLine)
```

Exercise 10: Weight of a structure in a note

It is a modification of Exercise 8. Instead of displaying a message window, put a note on the drawing with the information about the weight of the indicated structure.

Hint: Use the `withPoint` argument of the function `VTBasic.getModel()` to obtain the point, at which the given structure has been indicated.

🔗 The solution to this exercise can be found in the 'Vitesse Basic Training' folder under `SB_PYTHON` in the training project.

It is possible to move the reference symbol of an existing note or position number to the new position using the function `kcs_draft.reference_move()`

```
... #handle refers to the note or position number
newPos = KcsPoint2D.Point2D(150,150)
kcs_draft.element_visibility_set(handle, 0)
⇒ kcs_draft.reference_move(newPos)
kcs_draft.element_visibility_set(handle, 1)
```

- ❗ The function `kcs_draft.element_visibility_set()`, described in section 5.12, toggles the visibility of a note or position number on or off. Repainting the drawing (using `kcs_draft.dwg_repaint()`), after moving the reference symbol, would have a similar effect, but it is a less efficient solution.

5.9.3 Dimensioning

This section describes the functions used to create dimensioning components. The **kcs_draft** module provides the functions creating linear, angle, radius and diameter dimensioning components. The current values of the General Design defaults are taken into account. These functions take Vitesse 'geometric' class instances as parameters (Point2D, Vector2D, Point3D, Vector3D, etc.), and return a handle to the created component as an **ElementHandle** class instance.

5.9.3.1 2D dimensioning components

2D linear dimensions are created, as shown below:

```
p = KcsPoint2D.Point2D(x1, y1)
points = KcsPolygon2D.Polygon2D(p)
p = KcsPoint2D.Point2D(x2, y2)
points.AddPoint(p) #... add more measure points, if necessary
direction = KcsVector2D.Vector2D(1, 0) #example: horizontal dimension
position = KcsPoint2D.Point2D(measureX, measureY)
⇒ handle = kcs_draft.dim_linear_new(points, 1, direction, position)
```

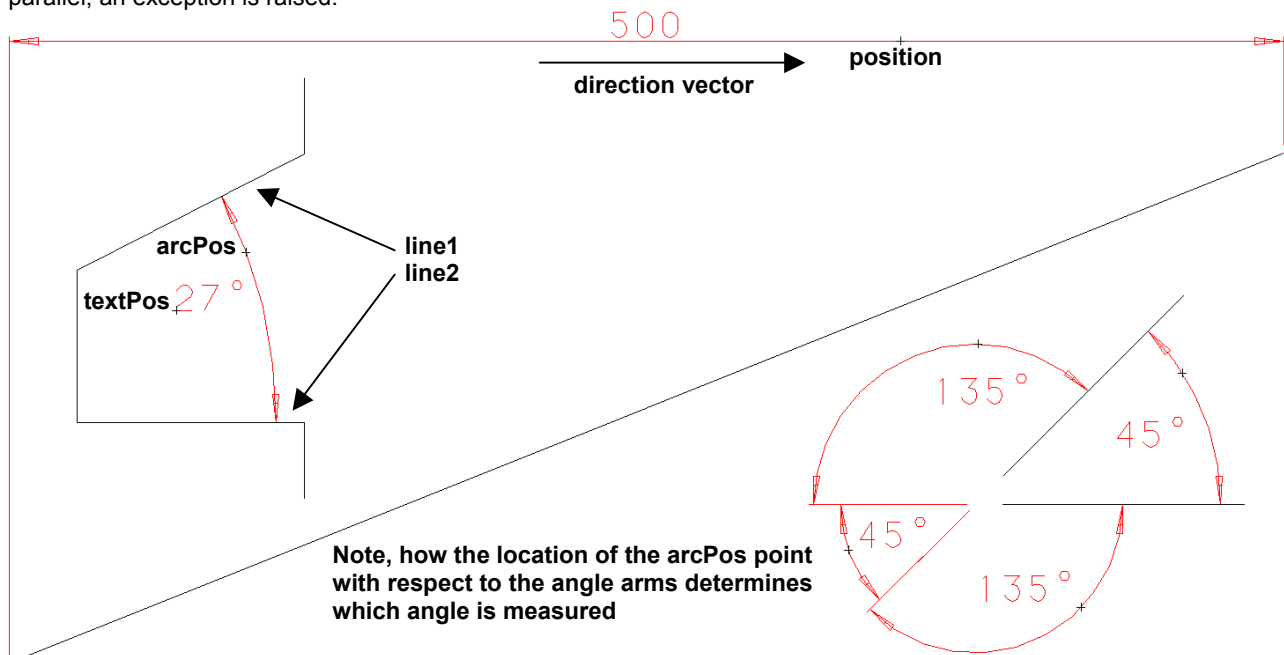
The **Polygon2D** class instance **points** is a collection of measure points; **direction** determines, whether the dimension is horizontal (1, 0), vertical (0, 1), or parallel (other – parallel to the dimension line); **position** defines the location of the measuring line. The second argument determines the type of a dimension:

- 1 – normal,
- 2 – chain,
- 3 – staircase

The function creates the linear 2D dimensioning component, given the set of measure points. The dimension line will be parallel to **direction** vector, and will pass through the **position** point. In a similar way, we can define and draw an angle dimension

```
p0 = KcsPoint2D.Point2D(x0, y0)
p1 = KcsPoint2D.Point2D(x1, y1)
line1 = KcsRline2D.Rline2D(p0, p1) #first arm of the angle
p2 = KcsPoint2D.Point2D(x2, y2)
line2 = KcsRline2D.Rline2D(p0, p2) #second arm of the angle
arcPos = KcsPoint2D.Point2D(arcPosX, arcPosY) #measure arc location
textPos = KcsPoint2D.Point2D(textPosX, textPosY) #measure text location
⇒ handle = kcs_draft.dim_angle_new(line1, line2, arcPos, textPos)
```

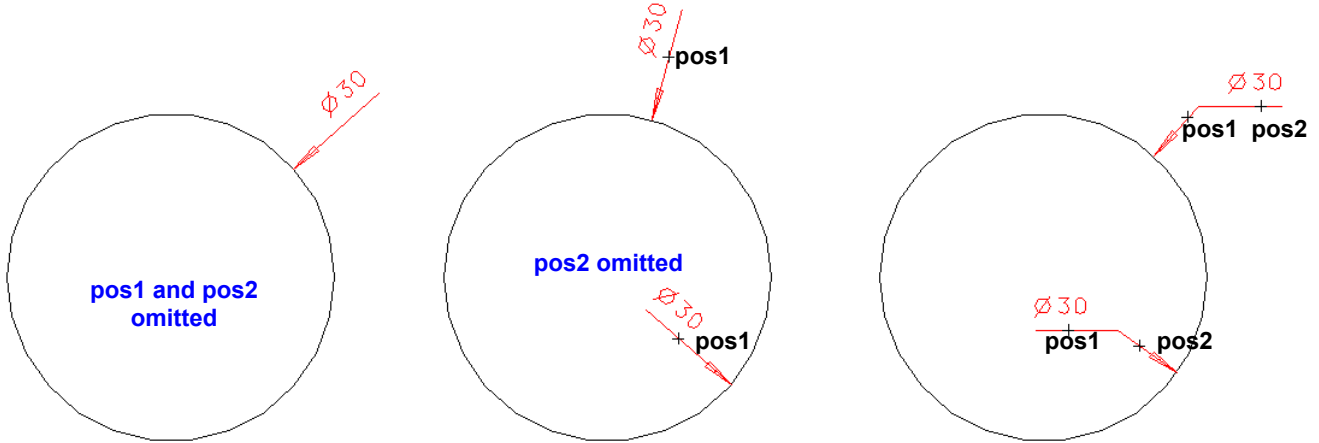
The function creates the 2D angular dimensioning component, given two non-parallel lines. If **line1** and **line2** are parallel, an exception is raised.



The diameter measures can be created by calling the function

```
handle = kcs_draft.dim_diameter_new(circle, pos1, pos2), or
handle = kcs_draft.dim_diameter_new(arc, pos1, pos2)
```

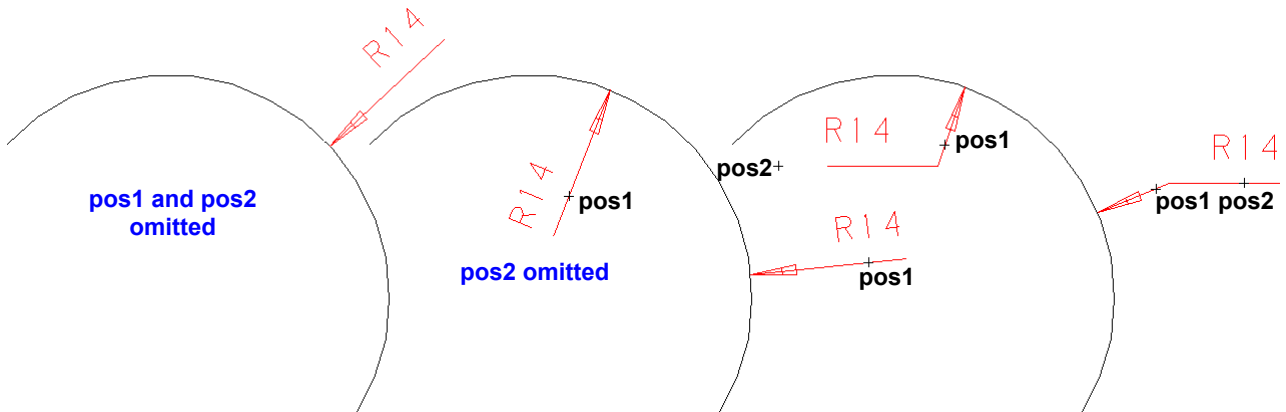
where the first argument is always either a **Circle2D** or **Arc2D** class instance, defining the circle or arc to be measured. The remaining arguments are the optional **Point2D** class instances, defining the location and shape of the measure (see examples below).



The function **kcs_draft.dim_diameter_new()** creates the diameter-dimensioning component, given either a circle or an arc. By specifying both **pos1** and **pos2** you create a 'knuckled' dimensioning component, whose reference line passes through **pos1** and then, horizontally, through **pos2**. You can omit the **pos2** (or both **pos2** and **pos1**) parameter and the function takes the default location of the reference line (straight, not 'knuckled'). A more detailed description can be found in the Tribon Vitesse User's Guide. In a similar way, we can create radius-dimensioning components:

```
handle = kcs_draft.dim_radius_new(circle, pos1, pos2), or
handle = kcs_draft.dim_radius_new(arc, pos1, pos2)
```

The **pos1** and **pos2** points are used in a similar way as in the previous function.



5.9.3.2 3D dimensioning components

Vitesse is able also to create 3D dimensioning components. The function below creates and draws the 3D point co-ordinates dimension.

```
handle = kcs_draft.dim_point_3d(point3D, locPoint2D, height, rotation, \
                                annotation, modelSubview)
```

where the coordinates of the point **point3D** are displayed in the drawing at the point **locPoint2D**. The dimension is created with the given text **height** and **rotation**. If not empty, the given **annotation** is added to the dimension. The argument **modelSubview** is optional. If given, defines the handle to the model subview, where the dimension is placed.

Next function creates and draws the 3D linear dimension.

```
handle = kcs_draft.dim_linear_new(points, type, projDir, locPoint2D, \
                                  witnDir, modelSubview, basepoint)
```

where the arguments have the following meaning:

| | |
|---------------------|---|
| points | (Polygon3D) list of 3D measure points |
| type | (integer) type of measure: 1 – normal, 2 – chain, 3 – staircase |
| projDir | (Vector3D) projection direction vector |
| locPoint2D | (Point2D) location in the drawing, where the dimension should be placed |
| witnDir | (Vector3D) direction vector for witness lines |
| modelSubview | (optional) handle to the model subview, to which the dimension component will be attached |
| basepoint | (integer, optional) zero-based index of the measure point used to define the dimension element plane (default: first point) |

i If **modelSubview** is not provided, the appropriate parent subpicture is chosen automatically.

The last function in this section creates a 3D dimensioning component, consisting of distances from the first object to the second one, along the third. This is used to measure distances along the shell model objects (longitudinals, transversals, hull curves, seams and stiffeners), which have to intersect the same surface.

```
handleList = kcs_draft.dim_shell_new(viewHandle, From, Along, To, \
                                     Type, Colour)
```

The following arguments have to be provided:

| | |
|------------------------|---|
| viewHandle | handle to the model view containing the objects to be measured |
| From, Along, To | (list) lists of Model class instances, describing groups of model objects. The measure will be calculated from the From group along the Along group to the To group. |
| Type | (integer) type of measure: 0 – only the dimension text will be created, 1 – dimension text with the arrow at the end point and circle at the start point, 2 – the above, including the dimension trace |
| Colour | (Colour) the colour of the dimension |

This function returns a list of handles to the created dimensioning components. Dimensions are created individually for each triple of objects from the groups **From**, **To**, and **Along**.

5.9.4 Other drawing components

Tribon Vitesse can also generate other drawing components, which are stored as Level 3 subpictures: pipe restriction symbols ...

```
... #subView is a handle to the subview, where the primitive is placed
start = KcsPoint2D.Point2D(100, 100) #start point
end = KcsPoint2D.Point2D(100, 120)   #end point
⇒ handle = kcs_draft.pipe_restr_symbol_new(subView, start, end)
```

... general restriction symbols ...

```
... #replace the last line of the above example with ...
⇒ handle = kcs_draft.general_restr_symbol_new(subView, start, end, True)
```

... where the last argument determines, how soft the symbol will be. The value of **True** indicates, that a spline (soft corners) should be used. Otherwise the symbol will be drawn using a contour (sharp corners).

Then we have yet clouds (based on rectangles or general polygons) ...

```
p1 = KcsPoint2D.Point2D(100,100)
p2 = KcsPoint2D.Point2D(150,130)
rect = KcsRectangle2D.Rectangle2D(p1, p2)
⇒ handle = kcs_draft.cloud_new(subView, rect) #or a Polygon2D
```

... crosses ...

```
p1, p2 = KcsPoint2D.Point2D(10,10), KcsPoint2D.Point2D(90,90)
p3, p4 = KcsPoint2D.Point2D(10,90), KcsPoint2D.Point2D(90,10)
line1, line2 = KcsRline2D.Rline2D(p1,p2), KcsRline2D.Rline2D(p3,p4)
text = KcsText.Text("Text in the cross")
⇒ handle = kcs_draft.cross_new(subView, text, line1, line2)
```

... rulers ...

```
p = KcsPoint2D.Point2D(10,50)
text = KcsText.Text("")
text.SetHeight(7.0)
⇒ handle = kcs_draft.ruler_new(subView, p, 20.0, 0, 200, 5, text)
```

... where we produce a ruler component in the given **subView**, starting at point **p**, with the ticks from **0** to **200**, displaced by **20.0** (in drawing's co-ordinate system. In scale 1:50 this produces ticks at every metre in the ship's co-ordinate system). The labels are drawn at every **5th** tick, using the text attributes from **text** variable. Finally, we can also produce standard position rulers

```
... #viewHandle refers to the view, on which the position ruler is drawn
start = KcsPoint2D.Point2D(10, 100)
end = KcsPoint2D.Point2D(300, 100)
⇒ handle = kcs_draft.position_ruler_new(3, viewHandle, start, end)
```

where we produce the Frame ruler (3) between the indicated points. The first argument indicates the type of the ruler to draw, and can take the following values:

- 1 – Base Line ruler
- 2 – Centre Line ruler
- 3 – Frame ruler
- 4 – Longitudinal Horizontal ruler
- 5 – Longitudinal Vertical ruler

5.10 Drafting Default Values

The appearance of the dimensioning components, as well as many other low-level aspects of Tribon Drafting are controlled through a number of parameters stored in the Drafting default file (SBD_DEF1 Tribon environment variable). The **kcs_draft** module provides two functions for reading and setting the Drafting defaults.

```
⇒ oldFont = kcs_draft.default_value_get("TEXT_FONT") #store original font
⇒ kcs_draft.default_value_set("TEXT_FONT:Arial")      #set new font
point = KcsPoint2D.Point2D(100,100)
handle = kcs_draft.text_new("This text is in Arial font", point)
kcs_draft.default_value_set(oldFont) #restore original font
```

The value returned by the function **kcs_draft.default_value_get()** has the format "**keyword** : **value**". This is also the required format of the argument of the function **kcs_draft.default_value_set()**.

Both functions will raise an exception if the default keyword is not recognised. Additionally the function **kcs_draft.default_value_set()** will raise an exception, if the value of the keyword is not valid, or if the argument does not have the right format.

5.11 Subpicture Managing Functions

We have already learned some functions creating subpictures. This includes the functions creating views (**view_new()**, **view_symbolic_new()**), subviews (**model_draw()**), and components (**note_new()**, **posno_new()**, **hatch_new()**, **dim_linear_new()**, etc.). All they were related either to the creation of model views or of the standard drawing components. It was not possible to create arbitrary subpictures 'on demand'. Until now ...

```
handle = kcs_draft.subview_new(name)
handle = kcs_draft.component_new(name)
```

These functions create a new subpicture of the appropriate level under the current parent subpicture, assign the given **name** and return the handle to the created subpicture. They cannot be used, if current subpicture is set to 'automatic'. Current subpicture setting can be controlled by the functions

```
currentSubList = kcs_draft.subpicture_current_get()
kcs_draft.subpicture_current_set(subpictureHandle)
```

The first one returns a 3-element list, consisting of the handles to the current view, current subview, and current component. If the current setting is 'automatic', the returned list is EMPTY!

❗ **NOTE:** If you use the function `kcs_draft.subpicture_current_get()`, when the current subpicture settings are not set to 'automatic' and there are no views, subviews or components, the system will create them and set them as current. Handles of created subpictures will be returned.

The second function sets a subpicture to be current for the creation of new geometries. If a component subpicture handle is given, this will be set as current. If a handle to view or subview is given, the component (and subview) will be chosen by the system. If no argument is given, the subpicture will be automatically chosen each time geometry is created ('automatic' mode – see restrictions above).

❗ If you try to set a view or a subview, which has no children as current, the system will create the component (and subview) for you and set it current.

It is possible to manipulate the names of the subpictures, using the functions

```
name = kcs_draft.subpicture_name_get(subpictureHandle)
kcs_draft.subpicture_name_set(subpictureHandle, name)
```

It is not possible to change the name of the drawing form view, or set to a subpicture a name, that is already occupied. Tribon Vitesse supports the exchange of subpictures between the drawing and the subpicture databank with the functions

```
kcs_draft.subpicture_save(subpictureHandle)
handle = kcs_draft.subpicture_insert(subpictureName, parentHandle, \
                                     databank)
```

The first function saves the subpicture identified by the given **subpictureHandle** to the standard subpicture databank. The subpicture must have a non-empty name, before this function can be called. The second function fetches a copy of the given subpicture from the subpicture databank (or the databank provided as the third argument – valid values are `kcs_draft.kcsSBD_PICT`, and `kcs_draft.kcsSBD_STD`) and inserts it in the drawing under the given parent subpicture, which must be of a correct level. If the inserted subpicture is a view (Level 1 subpicture), the **parentHandle** MUST NOT be given. The handle to the inserted subpicture is returned to the program.

Finally, we have a function for packing all components at the given subpicture (see also the function `kcs_draft.dwg_pack()` in section 5.2.4).

```
kcs_draft.element_pack(subpictureHandle)
```

If **subpictureHandle** is not given, all components in the drawing will be packed. The main goal of this function is to save system resources.

❗ After calling this function, the element handles belonging to the given subpicture are no longer valid.

5.12 Drawing Element Handling

This section describes functions dealing with the drawing elements, regardless of their type. Each of these functions refers to the drawing element by its handle. For removing a drawing element, we have a function

```
kcs_draft.element_delete(handle)
```

❗ This function is used not only for deleting basic geometry elements, but also for all other kinds of drawing elements, like: text elements, symbols, notes, position numbers, hatchings, views, subviews, components, etc.

NOTE: For some kinds of objects there are specialised functions doing a better job, like e.g. `kcs_draft.model_delete()`, which is able to remove the model object (a subview) from all views, where it is present.

If we want to delete multiple drawing elements, confined within a region (or located outside the region), the following function can be used:

```
... #define closed contour (Contour2D)
kcs_draft.delete_by_area(handleList, kcs_draft.kcsDEL_INSIDE, contour)
```

or

```
kcs_draft.delete_by_area(handleList, kcs_draft.kcsDEL_OUTSIDE, contour)
```

The elements, whose handles are in the **handleList**, that are located INSIDE or OUTSIDE (see the second argument) the contour are removed from the drawing.

An existing drawing element can be transformed using the function **kcs_draft.element_transform()**, which uses the **Transformation2D** class instance to define the transformation. See the example below ...

```
... #we have indicated a contour, and obtained its handle
trans = KcsTransformation2D.Transformation2D()
vector = KcsVector2D.Vector2D(0, 100) #displacement vector
trans.Translate(vector)                #apply the displacement
centre = KcsPoint2D.Point2D(200, 200) #rotation centre
trans.Rotate(centre, 30) #rotate around centre by 30 degrees
⇒ kcs_draft.element_transform(handle, trans) #apply the transformation
```

It is also possible to obtain the current transformation of any drawing element by calling the function **kcs_draft.element_transformation_get()**. See an example below, how to obtain the current view scale factor ...

```
... #handle refers to a view
trans = KcsTransformation2D.Transformation2D()
⇒ kcs_draft.element_transformation_get(handle, trans)
scaleDef = trans.GetScale()
xScaleFactor, yScaleFactor = scaleDef
```

The **GetScale()** method of the **Transformation2D** class returns a tuple, consisting of the X and Y scale factors. Usually, they will be the same, so it should not matter, which one you choose.

Instead of transforming an element, we can also define a new 2D transformation, and assign it to the element.

```
trans = KcsTransformation2D.Transformation2D()
... #define the transformation - translate, rotate, etc.
⇒ kcs_draft.element_transformation_redefine(handle, trans)
```

The ability to transform elements is especially useful in a combination with the function copying elements. The copy, when created, is located at the same position as the original. Then we have to move somehow the copy, so that we can distinguish it from the original.

```
copyHandle = kcs_draft.element_copy(handle, targetHandle)
```

This function copies an element and places it under the given **target** subpicture. If **target** is not given, the copy will be placed under the same parent subpicture as the original. As a result, it returns the handle to the created copy of the element. If the element is a subpicture and the target parent subpicture already has a subpicture with this name, the name of the copy will be blanked. You can then set a new name for the copy subpicture, if necessary.

When copying and subsequently transforming an element, the original element will be temporarily erased from the display. Please call the **kcs_draft.dwg_repaint()** function (or apply an identity transformation) to repaint the display so that the original element appears again.

It is possible to obtain the axis-parallel rectangle circumscribing the given element by calling the function

```
rectangle = kcs_draft.element_extent_get(handle)
```

The function returns the **Rectangle2D** class instance defining the area occupied by the element defined by **handle**. If **handle** is not specified, the returned rectangle defines the area occupied by the whole drawing. It is possible, that the given element is empty (e.g. empty subpicture). In this case the **IsEmpty()** method of the **Rectangle2D** class will return **1**, otherwise the result is **0**. This function is especially useful for fitting an element into the specified area in the drawing.

The visibility of the drawing elements can be controlled through the functions

```
Visible = kcs_draft.element_visibility_get(handle), and
kcs_draft.element_visibility_set(handle, Visible)
```

The value of **1** indicates, that the element is visible, and the value of **0**, that it is hidden. This is sometimes useful to turn off the visibility, make a change in the element, and turn the visibility back on. See the example on page 76.

5.13 Shading functions

Tribon system supports shading of model views. Only one shaded view can exist at a time. In order to create a shaded view from the given model view, pass its handle to the function

```
kcs_draft.shd_new(viewHandle)
```

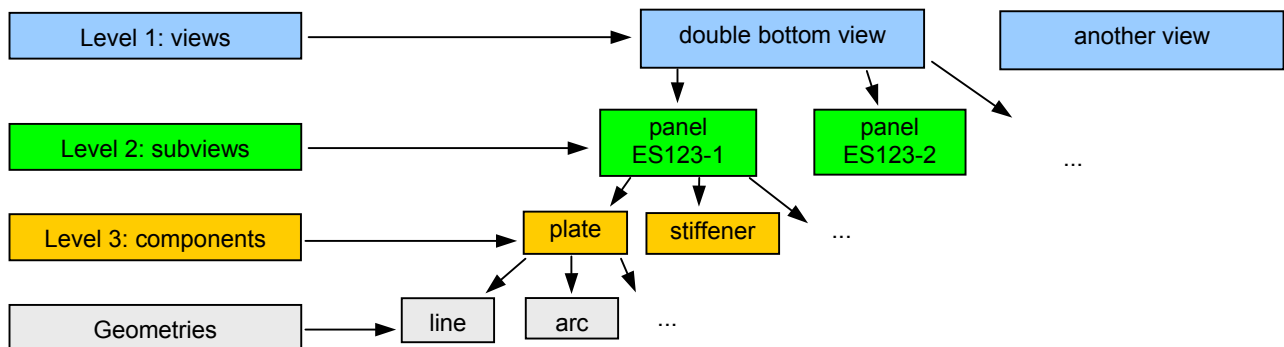
If a shaded view exists, an exception will be raised. It is possible to set the projection of the shaded view by setting up the Transformation3D class instance and passing it to the function `kcs_draft.shd_projection_set()`

```
origin = KcsPoint3D.Point3D(0,0,0)
U = KcsVector3D.Vector3D(1,0,0) #X axis
V = KcsVector3D.Vector3D(0,0,1) #Z axis
transf = KcsTransformation3D.Transformation3D()
transf.SetFromPointAndTwoVectors(origin, U, V)
⇒ kcs_draft.shd_projection_set(transf)
```

The existing shaded view can be automatically scaled using the function `kcs_draft.shd_autoscale()`, and zoomed using the function `kcs_draft.shd_zoom_box(p1, p2)`, where **p1**, and **p2** are the opposite corners (Point3D class instances) of the axis-parallel zooming box.

5.14 Traversing the drawing's structure

As we have already mentioned, the drawing's structure is hierarchical. Each drawing's element can have only one parent element, with the exception of views that do not have any parent. Each drawing's element can have one or more child elements, with the exception of geometry elements that do not have any child element. The picture below illustrates this structure.



In this example, the arrows indicate parent-child relationship. For the element entitled '**panel ES123-1**' we can say, that the element '**double bottom view**' is its parent, the elements '**plate**' and '**stiffener**' are its child elements, and the element '**panel ES123-2**' is its sibling. By descending down to the child elements and exploring the siblings on the given level, we can traverse the whole drawing's structure.

The Vitesse functions making this possible are described below.

```
parentHandle = kcs_draft.element_parent_get(handle)
```

This function returns the handle (**ElementHandle** class instance) to the parent element of the element identified by **handle**. Since the view does not have a parent, the function will raise an exception (`kcs_draft.error` set to '`kcs_NotFound`'), if a view handle is passed to this function.

```
childHandle = kcs_draft.element_child_first_get(subpictureHandle)
```

This function returns the handle (**ElementHandle** class instance) to the first child under the subpicture identified by **subpictureHandle**. If **subpictureHandle** is not given, a handle to the first view in the drawing is returned. Since the geometry elements do not have child elements, the function will raise an exception, if a geometry element handle is passed to this function.

```
siblingHandle = kcs_draft.element_sibling_next_get(handle)
```

This function returns the handle (**ElementHandle** class instance) to the next sibling of the element identified by **handle** (next element on the same level). When the list of elements on the given level is exhausted (no next sibling), the function will raise an exception, setting **kcs_draft.error** to '**kcs_NotFound**'.

🔗 The script 'Example 11.py' demonstrates how to analyse the contents of the views on the drawing using the above functions.