Plugins - Extend YASARA with your own functions

Plugins allow you to add your own menu options

A YASARA plugin is a Python script or Yanaconda macro that adds menu options and windows to the YASARA user interface and gets executed when you activate the respective option. Plugins are the ideal solution to add your own favorite functions to YASARA without having to play around with compilation, C and Assembler code.

All plugins are collected in the yasara/plg subdirectory, for latest developments visit the plugin repository at www.yasara.org/plugins.

Plugins can be written in Yanaconda or Python

Even though YASARA has its own built-in macro language, there are some applications where Yanaconda macros cannot help, like web-services and database-interfaces. Python is a well established, user friendly scripting language with excellent support for the above topics and was therefore the first choice.

In either case, YASARA changes the **current working directory** to the location of the plugin before running it, i.e. yasara/plg in Linux and Windows, or YASARA.app/yasara/plg in MacOSX.

Python can be downloaded from www.python.org

If you are a Linux or MacOSX user, Python is already present. If you are running Windows, it depends on your installation. Try to click Help > Install program > Python. If this option is not available, Python is already installed.

If you are new to Python, you can find a tutorial here.

Plugins must stick to format conventions

All plugins must be stored in YASARA's 'yasara/plg' subdirectory (in MacOSX, this folder can be found at YASARA.app/yasara/plg). Look at some of the files present there and pick the one closest to your needs as a programming scaffold.

The header identifies a plugin

The first line of a YASARA Python plugin must always read '# YASARA PLUGIN'. Adapt the remaining fields to describe your plugin. This information will be used if you decide to add it to the YASARA plugin repository at www.yasara.org/plugins in return for YASARA thank-you credits. Go to www.yasara.org/contribute to submit your plugin.

```
# YASARA PLUGIN
# TOPIC:       Database interfaces
# TITLE:       Retrieve a PDB file by FTP
# AUTHOR:      Elmar Krieger
# LICENSE:     GPL (www.gnu.org)
# DESCRIPTION: This plugin asks for a PDB ID and retrieves the file by FTP
#
```

The menu structure is enclosed in a triple-quote comment

After the header, the plugin specifies where to add which entries to YASARA's user interface to make selections and run the plugin. These definitions will be read when YASARA registers the plugin during startup and must be enclosed in triple-quotes ("""), so that Python ignores them. If you make changes here, you must restart YASARA. The indentation has to be a multiple of two spaces, Tabs are not allowed.

Here is an example:

```
"""
MainMenu: File
  PullDownMenu: Load
    SubMenu after PDB file from local PDB: PDB file from I_n_ternet
      TextInputWindow: Download PDB file from the web (IconDOWNLOAD)
        Text: Please input the four letter PDB ID code
        Text: _P_DB ID
      Request: DownloadPDB
"""
```

The example above tells YASARA to:

- Enter the main menu, search for entry 'File' (add it if not found).
- Open a pull down menu, search for entry 'Load' (add it if not found).
- Open a submenu (attached to the pull down menu), search for entry 'PDB file from local PDB', and add the new entry 'PDB file from Internet' immediately afterwards, with the 'n' underlined.
- Open a text input window with the header 'Download PDB file from the web', display a Download icon.
- Within the text input window, print the text 'Please input the four letter PDB ID code'.
- On top of the first (and only) text input box, print the text 'PDB ID' with the 'P' underlined.
- Run the plugin with the request 'DownloadPDB'

The triple-quote comment may contain the following keywords to identify certain menu and window types:

° MainMenu - Add an option to the top menu line

The indentation level for this keyword must be 0. You can use 'after' and 'before' to specify the location of the new option. Underscores surround underlined characters.

Examples:

```
MainMenu: File
MainMenu after Effects: A_n_imations
MainMenu before Analyze: _T_ools
```

° AtomContextMenu - Add an option to the atom context menu

The atom context menu appears when you click on a marked atom with the right mouse button, the indentation level for this keyword must be 0. You can use 'after' and 'before' as known from MainMenu.

The descriptor of the clicked atom can be accessed as yasara.selection[0].atom[0] from Python and as selection(1) from Yanaconda.

Example:

```
AtomContextMenu: _Q_uery mutation effects
```

° ResidueContextMenu - Add an option to the residue context menu

The residue context menu appears when you click on a residue in the sequence selector with the right mouse button, the indentation level for this keyword must be 0. You can use 'after' and 'before' as known from MainMenu.

The descriptor of the clicked residue can be accessed as yasara.selection[0].residue[0] from Python and as selection(1) from Yanaconda.

Example:

```
ResidueContextMenu before Label: _Q_uery mutation effects
```

° SelectionContextMenu - Add an option to the selection context menu

The selection context menu appears when you first select atoms using the interactive box, lasso or sphere selection tools (or directly with the Select command), and then right-click on one of the selected atoms. The indentation level for this keyword must be 0.

The descriptors of the selected atoms can be accessed as yasara.selection[0].atom[i] from Python and as selection(1) from Yanaconda (which simply contains the string 'selected').

Example:

```
SelectionContextMenu after Mass: _B_-factor
```

° HUD*ContextMenu - Add an option to one of the HUD context menus

The head-up-display (HUD) in the top right corner allows to browse the soup and offers context menus for atoms, residues, molecules and objects. You can add options to these menus using the following keywords (whose indentation level must be 0):

Keyword	Python access to selection
HUDAtomContextMenu	yasara.selection[0].residue[0]
HUDResidueContextMenu	yasara.selection[0].residue[0]
HUDMoleculeContextMenu	yasara.selection[0].molecule[0]
HUDObjectContextMenu	yasara.selection[0].object[0]

In Yanaconda, the ID of the selected unit can simply be found in variable 'selection(1)'.

Example:

```
HUDObjectContextMenu: _S_ample
```

° PullDownMenu - Add an option to a pull-down menu

This keyword must only be used after MainMenu with an indentation level of 1.

Example:

```
MainMenu: File
  PullDownMenu after Load: _G_et by FTP
```

° SubMenu - Add an option to a submenu

Submenus are little windows appearing on the side of pull-down menus or context menus. This keyword must therefore only be used after PullDownMenu, AtomContextMenu or ResidueContextMenu, with an indentation level of 1 or 2.

Example:

```
MainMenu: View
  PullDownMenu: Color
    SubMenu: by properties
```

° ObjectSelectionWindow - Add a window allowing to select objects

This keyword adds a standard object selection window, the selections are passed to the Python plugin via object descriptors. The specified text appears as the window header.

In Python, the jth object descriptor in the ith selection window can be accessed as yasara.selection[i].object[j], while Yanaconda stores a simple object selection string in selection(i), which can be passed to YASARA commands directly, e.g. to ListObj to obtain an object list. Note that counting for 'i' starts at 0 in Python and 1 in Yanaconda.

Example:

```
MainMenu: Analyze
  PullDownMenu: _M_CSIS mutations
    ObjectSelectionWindow: Select objects to map mutations stored in the MCSIS
    Request: MapMutations
```

° MoleculeSelectionWindow - Add a window allowing to select molecules

This keyword adds a standard molecule selection window, the selections are passed to the plugin via molecule descriptors. See ObjectSelectionWindow above for an example.

In Python, the jth molecule descriptor in the ith selection window can be accessed as yasara.selection[i].molecule[j], while Yanaconda stores a simple molecule selection string in selection(i), which can be passed to YASARA commands directly, e.g. to ListMol to obtain a molecule list. Note that counting for 'i' starts at 0 in Python and 1 in Yanaconda.

Example:

```
MainMenu: Analyze
  PullDownMenu: _C_ompare amino acid sequence of two molecules
    MoleculeSelectionWindow: Select first molecule to compare sequence
    MoleculeSelectionWindow: Select second molecule to compare sequence
    Request: CompareSequence
```

° ResidueSelectionWindow - Add a window allowing to select residues

Not surprisingly, this keyword adds a residue selection window, the selections are passed to the plugin via residue descriptors. See ObjectSelectionWindow above for an example.

In Python, the jth residue descriptor in the ith selection window can be accessed as yasara.selection[i].residue[j], while Yanaconda stores a simple residue selection string in selection(i), which can be passed to YASARA commands directly, e.g. to ListRes to obtain a residue list. Note that counting for 'i' starts at 0 in Python and 1 in Yanaconda.

Example:

```
MainMenu: Analyze
  PullDownMenu: _P_roscan
    ResidueSelectionWindow: Select residues for ProScan
    Request: Proscan
```

° AtomSelectionWindow - Add a window allowing to select atoms

The last keyword of this type adds an atom selection window, the selections are passed to the plugin via atom descriptors. See ObjectSelectionWindow above for an example.

In Python, the jth atom descriptor in the ith selection window can be accessed as yasara.selection[i].atom[j], while Yanaconda stores a simple atom selection string in selection(i), which can be passed to YASARA commands directly, e.g. to ListAtom to obtain an atom list. Note that counting for 'i' starts at 0 in Python and 1 in Yanaconda.

Example:

```
MainMenu: View
  PullDownMenu: Color
    SubMenu: by force
      AtomSelectionWindow: Select atoms to color by force
      Request: ColorByForce
```

° TextInputWindow - Add a window allowing to input one to four text strings

This keyword adds a window with one to four text input boxes. In addition to the window header, you must specify a general explanation for the user and then one header for each text input box.

The content of the jth text input box in the ith selection window can be accessed as yasara.selection[i].text[j] from Python and as selection(i)text(j) from Yanaconda. Note that counting for 'i' and 'j' starts at 0 in Python and 1 in Yanaconda.

Example for a window with two text input boxes:

```
MainMenu: Options
  PullDownMenu: _R_eport error
    TextInputWindow: Send an error report by e-mail
      Text: Please give a brief description of the problem:
      Text: _D_escription part 1 (header for the first input box)
      Text: _D_escription part 2 (header for the second input box), Default text
```

Note in the last line that a default text can be provided, separated with a comma ','. If the default is not always the same (e.g. a username), you can use 'TextFile:' instead of 'Text:' to read the data from a file as described here. This file must then be created by the 'CheckIfDisabled' startup code.

° NumberInputWindow - Add a window allowing to input one to six numbers

This keyword adds a window with one to six number input boxes. In addition to the window header, you must specify a general explanation text and then one descriptor for each number input box. A number descriptor contains four elements, separated by commas: the title of the number box, the default value, the minimum allowed value and the maximum allowed value. If the default value contains a dot '.', the field accepts floating point numbers, otherwise just integers are allowed.

The content of the jth number input box in the ith selection window can be accessed as yasara.selection[i].number[j] from Python and as selection(i)number(j) from Yanaconda. Note that counting for 'i' and 'j' starts at 0 in Python and 1 in Yanaconda.

Example for a window with one number input box:

```
MainMenu: Edit
  PullDownMenu after Build: Sample
    SubMenu: _O_bject
      ObjectSelectionWindow: Select protein to sample conformational space with CONCOORD
      NumberInputWindow: Select ensemble size
        Text: Number of structures in the CONCOORD ensemble:
        Number: _S_tructures,10,1,98
      Request: SampleObj
```

° RadioButtonWindow - Add a window allowing to switch between two to five options

This keyword adds a window with two to five radio buttons, where exactly one button can be selected. This allows to choose between up to five exclusive options, you must specify a general explanation for the user and then one additional text for every radio button.

The number of the selected radiobutton in the ith selection window can be accessed as yasara.selection[i].radiobutton from Python and as selection(i)radiobutton from Yanaconda. Note that counting for 'i' starts at 0 in Python and 1 in Yanaconda, while counting for 'radiobutton' always starts at 1.

Example for a window with two radiobuttons:

```
MainMenu: Options
  PullDownMenu: _R_eport error
    RadioButtonWindow: Concretize the error
      Text: Did the problem occur right now?
      Text: _Y_es, I did not exit YASARA since then.
      Text: _N_o, just before, I had to restart YASARA to get here.
```

° CheckBoxWindow - Add a window allowing to toggle one to five options

This keyword adds a window with one to five check boxes, that can be activated individually. This allows to toggle up to five independent options, you must specify a general explanation for the user and then one additional text for every check box.

The state of the jth check box in the ith selection window can be accessed as yasara.selection[i].checkbox[j] from Python and as selection(i)checkbox(j) from Yanaconda. Note that counting for 'i' and 'j' starts at 0 in Python and 1 in Yanaconda, the state is either 0 (not checked) or 1 (checked).

Example for a window with two check boxes:

```
MainMenu: NMR
  PullDownMenu: _L_ist restraints
    CheckBoxWindow: List distance and dihedral angle restraints
      Text: Select the type of restraints to list
      Text: Distance restraints
      Text: Dihedral restraints (Checked)
```

By default, all boxes are unchecked. To check a box, add the text '(Checked)' at the end as in the example above.

° ListWindow - Add a window allowing to select from a list

This keyword adds a window with a list of options. Set the 'MultipleSelections' flag to 'Yes' if the user is allowed to select more than one list entry and to 'No' otherwise. The first text is displayed above the list, the other texts are the actual list entries.

The total number of selected list entries can be accessed as yasara.selection[i].listentries from Python and as selection(i)listentries from Yanaconda.

The jth selected list entry in the ith selection window can be accessed as yasara.selection[i].list[j] from Python and as selection(i)list(j) from Yanaconda. Note that counting for 'i' and 'j' starts at 0 in Python and 1 in Yanaconda.

Example:

```
MainMenu: Analyze
```

```
  PullDownMenu: _P_DBFinder2 properties
    ResidueSelectionWindow: Select residues to color by PDBFinder2 properties
    ListWindow: Select PDBFinder2 properties
      MultipleSelections: Yes
      Text: Select more than one list entry to color by the average value
      Text: Nalign - Number of HSSP alignments
      Text: Nindel - Number of insertions and deletions
      Text: Entropy - HSSP sequence entropy
    Request: ColorResidues
```

If the list is long and has dynamic content, you can also read it from disk using the 'TextFile' keyword:

```
ListWindow: Select PDBFinder2 properties
  MultipleSelections: Yes
  Text: Select more than one list entry to color by the average value
  TextFile: options.txt
```

In the above example, the file 'options.txt' is read from the yasara/plg subdirectory, and each line becomes an entry in the list. (This approach works for all keywords, just append 'File' to the keyword name).

° FileSelectionWindow - Add a window allowing to select files

This keyword adds a window with a file browser. Set the 'MultipleSelections' flag to 'Yes' if the user is allowed to select more than one list entry and to 'No' otherwise. The Filename keyword specifies a wildcard with the initial path. Use forward slashes also under Windows.

The total number of selected filenames can be accessed as yasara.selection[i].filenames from Python and as selection(i)filenames from Yanaconda.

The jth selected filename in the ith selection window can be accessed as yasara.selection[i].filename[j] from Python and as selection(i)filename(j) from Yanaconda. Note that counting for 'i' and 'j' starts at 0 in Python and 1 in Yanaconda.

Example:

```
MainMenu: File
  PullDownMenu: Load
    SubMenu after PDB file: _N_MR ensemble
      FileSelectionWindow: Select a PDB file containing an NMR ensemble
        MultipleSelections: No
        Filename: pdb/*.pdb
      Request: LoadEnsemble
```

° ColorSelectionWindow - Add a window allowing to choose a color

This keyword adds a window to select a color. The 'ColorType' parameter decides if a rainbow color ('Bow', used to color atoms) or an RGB/HTML color ('RGB', used to color other things) should be selected. The 'ColorName' parameter sets a name for the color input box, a default color must be provided too (separated with a comma).

The selected color can be accessed as yasara.selection[i].color from Python and as selection(i)color from Yanaconda. Note that counting for 'i' and 'j' starts at 0 in Python and 1 in Yanaconda.

```
MainMenu: View
  PullDownMenu: Color
    SubMenu after Hydrogen bonds: Disulfide bonds
      ColorSelectionWindow: Please choose color for bridged cysteines
        ColorType: Bow
        ColorName: _C_ys color, Yellow
      Request: ColorCys
```

° CustomWindow - Add a window with custom design

The previous examples were predefined windows, with widgets placed automatically depending on the window type and the number of options required. It is also possible to design windows freely, by placing the widgets individually. The code is almost exactly the same as described for the Custom window type of the ShowWin command, that's why only an example is provided here, please refer to ShowWin for a description of the widgets:

```
MainMenu: Window
  PullDownMenu: Show custom window
    CustomWindow: This is a custom window
      Width: 600
      Height: 400
      Text:        X= 20,Y= 48,Text="A simple text at position 20/48, the top left corner"
      TextCen:     X=300,Y= 88,Text="A centered text"
      NumberInput: X= 20,Y= 88,Text="_F_loat",Default=5.0,Min=0,Max=10
      NumberInput: X=450,Y= 88,Text="_I_nteger",Default=25,Min=-100,Max=100
      TextInput:   X= 20,Y=158,Text="_E_nter any text",Width=380,Chars=100
      TextInput:   X=430,Y=158,Text="_P_assword",Width=150,Chars=10
      CheckBox:    X= 20,Y=228,Text="_T_ag this and/or..",Default=No
      CheckBox:    X= 20,Y=280,Text=".._t_his and/or..",Default=Yes
      CheckBox:    X= 20,Y=332,Text="..t_h_is one here.",Default=Yes
      RadioButtons:Options=3,Default=1
                   X=420,Y=228,Text="_S_elect this or.."
                   X=420,Y=268,Text="..this or.."
                   X=420,Y=308,Text="..this one here."
      List:        X=210,Y=238,Text="_C_hoose from a list:"
                   Width=190,Height=128,MultipleSelections=Yes
                   Options=6,  Text="Pick option 1"
                               Text="and/or option 2,"
                               Text="keep Ctrl pressed"
                               Text="to select more"
                               Text="than one option,"
                               Text="up to option 6 here"
      Button:      X=542,Y=348,Text="_O_ K"
    Request: PrintSelection
```

Here is the corresponding Python code to access the selections made:

```
print('The floating point number was %f'%yasara.selection[0].number[0])
print('The integer number was %d'%yasara.selection[0].number[1])
print('Username was %s, password was %s'%(yasara.selection[0].text[0],yasara.selection[0].text[1]))
for i in range(3):
  print('Checkbox %d state was %d'%(i+1,yasara.selection[0].checkbox[i]))
print('Radiobutton %d was selected'%yasara.selection[0].radiobutton)
print('These were the %d selected list entries:'%yasara.selection[0].listentries)
print(yasara.selection[0].list)
```

And here is the corresponding Yanaconda code to access the selections made:

```
print 'The floating point number was (0.000+selection(1)number(1))'
print 'The integer number was (selection(1)number(2))'
print 'Username was (selection(1)text(1)), password was (selection(1)text(2))'
for i=1 to 3
  print 'Checkbox (i) state was (selection(1)checkbox(i))'
print 'Radiobutton (selection(1)radiobutton) was selected'
print 'These were the (selection(1)listentries) selected list entries:'
print (selection(1)list)
```

To keep access to the input compatible with the other selection windows, each custom window may currently contain only a single list, a single radio button selection, and a single button (usually named 'OK'). This limitation does not apply to the ShowWin command.

° The layout for the remaining plugin differs between Python and Yanaconda

When using Yanaconda, the rest of the plugin is a straightforward macro. The variable 'request' contains the specified request, e.g. 'DownloadPDB' and can be used to execute different parts of the macro. Selections can currently not be accessed.

When using Python, the first command must be

```
import yasara
```

Immediately afterwards, you can access the data passed to the plugin as well as call YASARA functions. The predefined variables are listed below, the most important one is yasara.request, a string indentifying the user's request, as specified after 'Request:', e.g. 'DownloadPDB'. When YASARA registers the plugins during startup, it also sends a 'CheckIfDisabled' request, giving the plugin the possibility to exclude itself from registration.

The remaining plugin scaffold therefore looks like that:

```
if (yasara.request=="CheckIfDisabled"):
  # Assign a 1 to yasara.plugin.exitcode if this plugin cannot work and should
  # be disabled (data missing, wrong operating system etc.)
  if (....) yasara.plugin.exitcode=1

elif (yasara.request=="DownloadPDB"):
  # Do the work

# End the plugin, must be the last command
yasara.plugin.end()
```

Note that the 'CheckIfDisabled' costs time since Python has to be run, and therefore slows down YASARA's start. It is therefore avoided if possible. If a plugin does not work in all operating systems, this should be declared in the header instead, using the 'PLATFORMS:' field:

```
# YASARA PLUGIN
# TOPIC:       Molecular Modeling
# TITLE:       Align3D
# AUTHOR:      Mikael Roche & Emmanuel Bettler
# LICENSE:     GPL
# DESCRIPTION: This plugin performs a structural alignment [...]
# PLATFORMS:   Linux,MacOS
```

Also note that you cannot run YASARA commands when handling the 'CheckIfDisabled' request, since this happens during startup, before the user interface is created.

◦ Plugins can be rerun quickly by pressing Alt or Tab

Many YASARA commands can be repeated by holding down Alt or Tab and clicking on an atom. This approach also works with plugins that are hooked into one of the context menus like the AtomContextMenu.

YASARA will display a message at the bottom which is derived by splitting the 'Request' keyword at capital letters.

◦ Plugins can access most YASARA functions

For Yanaconda plugins this is trivial, as they are just macros. For Python plugins, the YASARA functions are wrapped so that they can be accessed with a syntax that matches Python's requirements.

Example: The YASARA command to choose a new 3D font..

```
Font Arial,Height=2,Spacing=1.5,Color=Yellow,Depth=5,DepthCol=Red
```

becomes

```
yasara.Font("Arial",height=2,spacing=1.5,color="Yellow",depth=5,depthcol="Red")
```

Note that argument names are lowercase in Python, because in contrast to Yanaconda, Python's variable names are case-sensitive, and the capitalization is often ambiguous and hard to remember, raising the error rate.

The documentation page of each YASARA command lists the prototype of the corresponding Python function, e.g. the Font command (look at the 'Python:' row in the table at the top of each page).

A few YASARA commands support more than one format with different argument types. This is not possible in Python, the command thus has to be wrapped by different Python functions. The names of these Python functions differ at the end, using either an increasing number or the name of the first argument. More details are available here.

You can of course also access the return values of YASARA commands:

```
# Load a PDB file and color it red
obj = yasara.LoadYOb("1crn")
yasara.ColorObj(obj,"Red")
```

More details about return values can be found here.

It should also be noted that calling a YASARA command from Python is slower than using a Python method, since it involves communication between Python and YASARA. So calls to YASARA commands should be taken out of loops when possible:

```
# Load a PDB file
yasara.LoadPDB("1crn")
# Switch off the console to avoid screen updates
yasara.Console("off")
# Print atom names the slow way
for i in range(yasara.CountAtom("all")[0]):
  name = yasara.NameAtom(i+1)[0]
  print "Atom %d has name %s"%(i+1,name)
# Print atom names the fast way
namelist = yasara.NameAtom("all")
for i in range(len(namelist)):
  print "Atom %d has name %s"%(i+1,namelist[i])
```

If all fails, you can still use the 'run' function to execute any command, also those without a Python wrapper (mostly WHAT IF commands in the Twinset):

```
# Load 1crn, avoiding the Python wrapper 'LoadPDB("1crn")'
yasara.run("LoadPDB 1crn")
# Enter WHATIF's GRAFIC menu
yasara.run("WHATIF")
yasara.run("GRAFIC")
# Show a wire frame
yasara.run("SHOTOT 1 Crambin")
# Go back to YASARA
yasara.run("YASARA")
# List all objects without the Python wrapper 'ListObj("all")'
```

```
yasara.run("ListObj all")
```

Python plugins can access a number of predefined variables

Right after the 'import yasara' statement, the following variables can be accessed. Note that [i] specifies the number of the selection window, counting starts with zero at each chain of selection windows leading to a 'Request' keyword. [j] specifies the number of the selected item, counting starts at 0 in every selection window.

`yasara.request`	The request string sent by YASARA to the plugin
`yasara.opsys`	The current operating system, "Linux", "MacOS" or "Windows"
`yasara.version`	The YASARA version string X.Y.Z
`yasara.serialnumber`	YASARA's serial number
`yasara.stage`	The YASARA stage View, Model, Dynamics or Structure
`yasara.plugin.name`	The name of the plugin (e.g. ftppdb.py)
`yasara.plugin.config`	A Python dictionary with the options from the plugin config file *.cnf
`yasara.plugin.exitcode`	The exit code returned to YASARA when the plugin ends
`yasara.owner.firstname`	Your first name
`yasara.owner.email`	Your e-mail address
`yasara.permissions`	The permissions of the 'yasara' directory, to be propagated to files updated by the plugin
`yasara.workdir`	YASARA's current working directory, which differs from the plugin's working directory (yasara/plg)
`yasara.selection`	A list of selections with one entry for every selection window you defined
`yasara.selection[i].objects`	The number of objects selected in the ith selection window
`yasara.selection[i].object[j]`	The obj_descriptor (see below) for the jth selected object in the ith selection window
`yasara.selection[i].molecules`	The number of molecules selected in the ith selection window
`yasara.selection[i].molecule[j]`	The mol_descriptor (see below) for the jth selected molecule in the ith selection window
`yasara.selection[i].residues`	The number of residues selected in the ith selection window
`yasara.selection[i].residue[j]`	The res_descriptor (see below) for the jth selected residue in the ith selection window
`yasara.selection[i].atoms`	The number of atoms selected in the ith selection window
`yasara.selection[i].atom[j]`	The atom_descriptor (see below) for the jth selected atom in the ith selection window
`yasara.selection[i].texts`	The number of text input boxes in the ith selection window
`yasara.selection[i].text[j]`	The text typed into the jth text input box in the ith selection window
`yasara.selection[i].numbers`	The number of number input boxes in the ith selection window
`yasara.selection[i].number[j]`	The number typed into the jth number input box in the ith selection window
`yasara.selection[i].checkboxes`	The number of checkboxes in the ith selection window
`yasara.selection[i].checkbox[j]`	The state of the jth checkbox in the ith selection window (1=hooked, 0=not hooked)
`yasara.selection[i].radiobutton`	The number of the selected radiobutton in the ith selection window (1=first, 2=second.., None if there was no radiobutton).
`yasara.selection[i].listentries`	The number of selected list entries in the ith selection window
`yasara.selection[i].list[j]`	The jth selected list entry in the ith selection window
`yasara.selection[i].filenames`	The number of selected filenames in the ith selection window
`yasara.selection[i].filename[j]`	The jth selected filename in the ith selection window

Object descriptors identify selected objects

Object descriptors are instances of the class obj_descriptor. Typically, you loop over all object descriptors in the ith selection window:

```
for j in range(yasara.selection[i].objects):
  object=yasara.selection[i].object[j]
```

And then access various object properties:

`object.name`	The name of the object
`object.number.inyas`	

`object.number.inyas`	The unique number/ID of the object in YASARA (a string, starting with 1)
`object.number.inall`	The sequential number of the object in the soup (a string, starting with 1)

You could then color the object red:

```
yasara.ColorObj(object.number.inyas,"Red")
```

◦ Molecule descriptors identify selected molecules

Molecule descriptors are instances of the class mol_descriptor. Typically, you loop over all molecule descriptors in the ith selection window:

```
for j in range(yasara.selection[i].molecules):
  molecule=yasara.selection[i].molecule[j]
```

And then access various molecule properties:

`molecule.name`	The name of the molecule (that's the chain name in the PDB file)
`molecule.number.inyas`	The unique number/ID of the molecule in YASARA (a string). Becomes invalid if atoms are added or deleted.
`molecule.number.inall`	The sequential number of the molecule in the soup (a string, starting with 1)
`molecule.number.inobj`	The sequential number of the molecule in the object (a string, starting with 1)
`molecule.object`	The object descriptor for the object the molecule belongs to

You could then display the molecule as sticks:

```
yasara.StickMol(molecule.number.inyas)
```

Or delete the entire object containing this molecule:

```
yasara.DelObj(molecule.object.number.inyas)
```

◦ Residue descriptors identify selected residues

Residue descriptors are instances of the class res_descriptor. Typically, you loop over all residue descriptors in the ith selection window:

```
for j in range(yasara.selection[i].residues):
  residue=yasara.selection[i].residue[j]
```

And then access various residue properties:

`residue.name3`	The name of the residue in three letter code.
`residue.name1`	The name of the residue in one letter code.
`residue.number.inyas`	The unique number/ID of the residue in YASARA (a string). Becomes invalid if atoms are added or deleted.
`residue.number.inall`	The sequential number of the residue in the soup (a string, starting with 1).
`residue.number.inobj`	The sequential number of the residue in the object (a string, starting with 1).
`residue.number.inmol`	The sequential number of the residue in the molecule (a string, starting with 1).
`residue.number.inpdb`	The number of the residue in the PDB file (a string, last character may be the insertion code).
`residue.object`	The object descriptor for the object the residue belongs to.
`residue.molecule`	The molecule descriptor for the molecule the residue belongs to.

You could then color the residue yellow:

```
yasara.ColorRes(residue.number.inyas,"Yellow")
```

Or display a ribbon for the entire molecule containing this residue:

```
yasara.ShowSecMol(residue.molecule.number.inyas,"Ribbon")
```

◦ Atom descriptors identify selected atoms

Atom descriptors are instances of the class atom_descriptor. Typically, you loop over all atom descriptors in the ith selection window:

```
for j in range(yasara.selection[i].atoms):
  atom=yasara.selection[i].atom[j]
```

And then access various atom properties:

`atom.name`	The name of the atom
`atom.namespaced`	The name of the atom including spaces (always four characters)
`atom.altloc`	The alternate location indicator of the atom
`atom.position`	The position of the atom, a list with three cartesian coordinates
`atom.occupancy`	The occupancy field of the atom in the original PDB file
`atom.bfactor`	The B-factor of the atom
`atom.number.inyas`	The unique number/ID of the atom in YASARA (a string). Becomes invalid if atoms are added or deleted.
`atom.number.inall`	The sequential number of the atom in the soup (a string, starting with 1, the same as .inyas above).
`atom.number.inobj`	The sequential number of the atom in the object (a string, starting with 1, usually the same number as in the PDB file).
`atom.number.inmol`	The sequential number of the atom in the molecule (a string, starting with 1).
`atom.number.inres`	The sequential number of the atom in the residue (a string, starting with 1).
`atom.object`	The object descriptor for the object the atom belongs to.
`atom.molecule`	The molecule descriptor for the molecule the atom belongs to.
`atom.residue`	The residue descriptor for the residue the atom belongs to.

You could then color the atom green:

```
yasara.ColorAtom(atom.number.inyas,"Green")
```

Or delete the entire residue the atom belongs to:

```
yasara.DelRes(atom.residue.number.inyas)
```

Python plugins can access persistent storage in YASARA

If a Python plugin terminates by calling yasara.plugin.end(), the Python interpreter stops executing the plugin and all variables are lost. Complex plugins may like to preserve certain variables until the plugin is run again by the user (for example username and password for a database connection). This can be achieved by using YASARA's persistent storage facility. As shown in the example below, you only need to initialize the variable 'yasara.storage' (this name is fixed) when the plugin is run for the first time, usually by assigning an instance of the trivial 'container' class. All data stored in this container will then be preserved across calls to the plugin, until the user exits YASARA. If you want to preserve data until YASARA is run again, then you need to save it to disk.

```
# YASARA PLUGIN
# DESCRIPTION: Click 'Options > Test plugin' multiple times to show a counter
"""
MainMenu: Options
  PullDownMenu after Stop plugin: Test plugin
    Request: TestPlugin
"""

import yasara,time
from python2to3 import *
from container import *

if (yasara.request=="TestPlugin"):
  if (yasara.storage==None):
    # Plugin is run for the first time, create a persistent storage container
    yasara.storage=container()
    # Store something
    yasara.storage.counter=1
  else:
    # Not the first time, increment counter in persistent storage container
    yasara.storage.counter+=1
  # Display the counter
  yasara.ShowMessage("Test plugin has been run %d times..."%yasara.storage.counter)
  time.sleep(3)
  yasara.HideMessage()
# This must always be the last command
yasara.plugin.end()
```

Yanaconda plugins can access a number of predefined variables

In addition to a large number of predefined variables that can be accessed by all Yanaconda macros, the following additional variables are available to macros run as a plugin:

Note that (i) specifies the number of the selection window, counting starts at 1 at each chain of selection windows leading to a 'Request' keyword. (j) specifies the number of the selected item, counting starts at 1 in every selection window.

`request`	The request string sent by YASARA to the plugin
`selection(i)`	An atom to object selection string for atom to object selection windows and context menus
`selection(i)texts`	The number of text input boxes in the ith selection window
`selection(i)text(j)`	The text typed into the jth text input box in the ith selection window
`selection(i)numbers`	The number of number input boxes in the ith selection window
`selection(i)number(j)`	The number typed into the jth number input box in the ith selection window
`selection(i)checkboxes`	The number of checkboxes in the ith selection window
`selection(i)checkbox(j)`	The state of the jth checkbox in the ith selection window (1=hooked, 0=not hooked)
`selection(i)radiobutton`	The number of the selected radiobutton in the ith selection window (1=first, 2=second..).
`selection(i)listentries`	The number of selected list entries in the ith selection window
`selection(i)list(j)`	The jth selected list entry in the ith selection window
`selection(i)filenames`	The number of selected filenames in the ith selection window
`selection(i)filename(j)`	The jth selected filename in the ith selection window

◦ Plugins can create interactive user interfaces

In addition to the various selection windows that plugins can add to YASARA's user interface via a triple-quote comment, they can also create interactive elements on the fly, while they are running.

There are two different approaches:

- Plugins can use the ShowWin command to open a window and obtain the selections made as a list of return values.
- Plugins can create a custom user interface by printing text, drawing rectangles and showing clickable buttons at three different locations: In the head-up display, in images (those shown directly on screen and those attached to 3D image objects), and in a second window.

  When the user clicks on a button, the plugin is run again with a request that can be provided as the button's 'Action parameter'. Alternatively, the plugin can wait until any button is pressed (which has the drawback that no other plugin can be run while the first is waiting).

  The following example Python plugin shows the details, it creates three buttons that perform various actions. To test the plugin, save it as 'yasara/plg/buttontest.py' (don't change the name), restart YASARA and click Window > Head-up display > Test HUD buttons:

```
# YASARA PLUGIN
# TOPIC:       Database interfaces
# TITLE:       Test interactive buttons in the HUD
# AUTHOR:      Elmar Krieger
# LICENSE:     GPL
# DESCRIPTION: This plugin shows text and some buttons, and handles the button clicks
#
"""
MainMenu: Window
  PullDownMenu: Head-up display
    SubMenu after Off: Test HUD _b_uttons
      Request: ShowTestButtons
"""

import yasara
from python2to3 import *

# CREATE SOME TEST BUTTONS
# ========================
def ShowButtons():
  yasara.Font("Arial",height=25,color="White")
  yasara.PosText(x="50%",y=8,justify="center")
  yasara.Print("Button Test")
  yasara.Font(height=14)
  # Show a simple button, runs this plugin with request=="Showsidechains"
  yasara.ShowButton("Show sidechains",y=100,color="Red")
  # Show a button that opens a browser window
  yasara.ShowButton("Take me to YASARA.org",y=180,color="Green",
                    action="ShowURL http://www.yasara.org")
  # Show a complex button, runs this plugin with request=="ColorCYS" and a selection
  # of all cysteine residues in yasara.selection[0].residue
  yasara.ShowButton("Color cysteines yellow",y=260,color="Blue",
                      action="SavePLIRes CYS,Request=ColorCYS|RunPlugin buttontest.py")

# MAIN PLUGIN
# ===========
if (yasara.request=="ShowTestButtons"):
  # Show example buttons in the HUD
  yasara.PrintHUD()
  ShowButtons()
  # Show example buttons in a moving image
  img=yasara.MakeImage("Buttons",width=512,height=512,topcol="None",bottomcol="None")
  yasara.ShowImage(img,x=-256,y=-128,width=1024,height=768,alpha=100,priority=0)
  yasara.AutoMoveImage(img,x=-256,y=250,width=1024,height=12,alpha=0,steps=400,cycle=1)
  yasara.PrintImage(img)
  ShowButtons()
  # Show example buttons in a 3d image object
  # We simply reuse image 'buttons', which means that button clicks will be visible in both
  obj=yasara.MakeImageObj("Buttons",img,width=40,height=40)
  yasara.AutoRotateObj(obj,y=0.3)
  yasara.MoveObj(obj,x=-20)
  # Direct printing back to console
  yasara.PrintCon()

if (yasara.request=="Showsidechains"):
  # The 'Show sidechains' button has been clicked
  yasara.ShowAtom("Sidechain CA")

if (yasara.request=="ColorCYS"):
  # The 'Color cysteines yellow' button has been clicked, the cysteine
  # residues have been selected via the 'SavePLIRes CYS' command
  if (yasara.selections):
    for i in range(yasara.selection[0].residues):
      yasara.ColorRes(yasara.selection[0].residue[i].number.inyas,"yellow")

# This must always be the last command
yasara.plugin.end()
```

° Python plugins run in a separate thread

While a Python plugin is running, you can continue using YASARA normally. There are in fact two threads working in parallel: YASARA and the plugin. When a plugin runs a YASARA command, this command is passed from the plugin to YASARA and executed as soon as possible. **It is normally NOT guaranteed that YASARA has finished a command when the function call in the plugin returns.** This can lead to potential problems if there is a data dependency between YASARA and the plugin, usually involving files on the hard disk accessed by both.

The solution is to let the plugin wait until YASARA has finished executing the command. This can simply be achieved by using the return value (which can only be known after YASARA finished the command). If you do not need the return value, consider putting the command between brackets [..], which is a good way to indicate that synchronization is requested. Here are three typical examples:

- **The plugin reads a file created by YASARA**: Since this is a common task, all YASARA commands that save data to the hard disk are synchronized automatically. The only exception is the LogAs command:

```
# Log the output of the next command
yasara.LogAs("MyLog")
# List all hydrogen bonds, and assign the return value to a dummy
# variable, so that the plugin is forced to wait until YASARA finished:
dummy = yasara.ListHBoAtom("all","all")
# Or, alternatively with less typing:
[yasara.ListHBoAtom("all","all")]
# Read the log file in Python
log=open("MyLog").readlines()
```

- **YASARA reads a file created by the plugin**: This is normally not a problem, unless the file is a temporary one, and the plugin decides to delete it. In this case, the plugin must wait for YASARA to read the file before deleting it:

```
# Download a certain unofficial PDB file from the web
pdb=urllib2.urlopen(url).readlines()
# Save it temporarily
open(pdbfilename,"w").writelines(pdb)
# Read it in YASARA, and wait until YASARA has finished
[yasara.LoadPDB(pdbfilename)]
# Delete the temporary file only after it has been read by YASARA
os.path.remove(pdbfilename)
```

Instead of waiting for YASARA, one can also let YASARA delete the file, which avoids synchronization issues:

```
# Read it in yasara
yasara.LoadPDB(pdbfilename)
# And delete
yasara.DelFile(pdbfilename)
```

- **Catching a YASARA exception**: Since the plugin does normally not wait until YASARA has finished executing a command, any error YASARA encounters is shown on screen, but cannot be reported back to the plugin. Again, the solution is to force synchronization by using the command's return value or enclosing the command in brackets:

```
# Try to initialize force field parameters, forcing the plugin to wait for YASARA:
try:
  yasara.ShowMessage("Initializing simulation")
  [yasara.Sim("init")]
except:
  yasara.ShowMessage("Could not initialize simulation")
```

Plugins can be speeded up

Normally YASARA executes each command issued by the plugin just as if it had been created via the graphical user interface. This includes an update of the graphics display after each command. If the plugin issues hundreds of commands, this approach may become too slow. In this case resort to the trick used by Yanaconda macros to speed up execution: just switch off the console.

```
yasara.Console("Off")
```

(See the Console command for more details and note that YASARA will not redraw the screen unless you tell it to, and may thus appear frozen. So don't forget to enable the console again with yasara.Console("Hidden") when leaving the performance critical section).

A common performance-critical example is building a large number of atoms. The following example duplicates each atom of PDB file 5tim, at a rate of about 1000 atoms per second:

```
# Make sure that there are no unused objects between others
yasara.RenumberObj("all",1)
# Load the PDB file (becomes the last object), store the number of objects in 'objects'
objects=yasara.LoadPDB("/home/pdb/5tim",model=1)[0]
# Get the global coordinates and chemical elements of all the atoms
poslist=yasara.PosAtom("Obj %d"%objects,coordsys="global")
elementlist=yasara.ElementAtom("Obj %d"%objects)
# Speed up
yasara.Console("off")
for i in range(len(elementlist)):
  # Build a new atom and place it at the right spot
  obj=yasara.BuildAtom(elementlist[i])
  yasara.PosObj(obj,x=poslist[i*3],y=poslist[i*3+1],z=poslist[i*3+2])
  if (not i%100 or i==len(elementlist)-1):
    # As soon as 100 atoms have been built, join them (too many objects slow YASARA down,
    # note also that you need to use i%9 in YASARA View, which supports only 10 objects)
    yasara.JoinObj("not 1-%d"%(objects+1),objects+1)
yasara.Console("on")
```

Plugins can be run from the command line and in console mode

Plugins are normally linked to options in the user interface. Sometimes, it may be helpful to run a plugin directly. This is achieved with the RunPlugin command in connection with SavePLI:

```
# Save a PLugin Input (PLI) File containing a selection of Calpha atoms
# and the request 'MyRequest'
SavePLIAtom CA,MyRequest
# Run the plugin
RunPlugin MyPlugin.py
```

If no selections are required, the plugin can also be run from the command line:

```
yasara PathToMyPlugin/MyPlugin.py MyRequest
```

Without any selections, no graphical user interface is needed, and plugins can also be run in console mode:

```
yasara -con PathToMyPlugin/MyPlugin.py MyRequest
```

If you want to exit YASARA as soon as the plugin has finished, add this line to the end of the plugin:

```
yasara.Exit()
```

Plugins can start additional programs that control YASARA, like a Python module

Some specialized applications may require to control YASARA from an external program. For example, the 3DM system from Bio-Prodict.nl allows to control YASARA from a web browser window. This is a non-trivial task that can be achieved with these steps:

- Write a Python module that opens a socket and listens for YASARA control instructions, which it transforms into YASARA commands.
- Use Java(-Script) on your webpage to check if this Python module is listening, connect to it if yes, and send YASARA control instructions that reflect what the user does in the browser. If the Python module is not listening, your webpage may start YASARA first, e.g. via the MIME-type setting if your browser's security settings don't allow to launch an external program.
- To make sure that your Python module is started together with YASARA, save the module in the 'yasara/pym' folder and create a corresponding Python plugin (saved in the 'yasara/plg' folder) that contains the following launch code:

  ```
  if (yasara.request[:12]=="LaunchModule"):
    # YASARA requests to launch the Python module now. The Python module must be
    # placed in the yasara/pym folder. Since this is a Python plugin, we are currently
    # in the yasara/plg folder, so we need to prepend ../pym/ to the module name.
    # The yasara.request contains additional connection information at the end and
    # must be forwarded to the Python module as command line parameter.
    command='"'+sys.executable+'" ..'+os.sep+'pym'+os.sep+'3dmcommunicator.py '+yasara.request
    subprocess.Popen(command,shell=True)
    # If something went wrong launching, you could tell YASARA with a non-zero exitcode:
    # yasara.plugin.exitcode=1
  ```

  When YASARA starts, it sends a 'LaunchModule' request to your plugin, which uses 'subprocess.Popen' to launch your Python module, in this example 'yasara/pym/3dmcommunicator.py'. Also note that 'yasara.request' needs to be passed to your Python module, in the example as the first command line parameter.

- In your Python module (e.g. yasara/pym/3dmcommunicator.py) you need to connect to the YASARA instance that launched the module, so that you can send it YASARA commands. This is done with the following code:

  ```
  # Import the YASARA Python module yasara/pym/yasara.py
  import yasara

  # Connect to the YASARA that launched us, which is identified via
  # its 'yasara.request' (passed here as command line parameter sys.argv[1])
  yasara.connect(sys.argv[1])
  ```

  So to make sure that your Python module does not launch a new YASARA but instead communicates with the already running YASARA, you need to use the function 'yasara.connect', providing the original yasara.request as argument.

  Then your Python module can already start to issue YASARA commands, e.g.

  ```
  yasara.LoadPDB("1crn",download=1)
  while 1:
   for i in range(46):
     yasara.ColorRes(i+1,"blue")
     time.sleep(2)
  ```

- As a special service, YASARA kills your Python module when the user exits YASARA (since your Python module was started as a separate process with subprocess.Popen, it would normally continue running and crash eventually).

◦ Debugging is done by adding temporary print commands

Programs contain errors, the same is true for plugins. There are two types of errors in Python plugins:

- Errors that occur during the initial plugin registration when YASARA starts up. Most of the time these are simple syntax errors. In Linux and MacOSX, you see the error message in the console from where you started YASARA. Windows can unfortunately not display the error message, but you know that something went wrong because your plugin does not appear in YASARA's user interface. Open a command prompt, go to the yasara\plg directory and run the plugin directly with the Python interpreter to locate the problem:

  ```
  c:\MyPythonInstallationPath\python.exe MyPlugin.py
  ```

  This will show you a traceback. After correcting the error you have to restart YASARA.

- Errors that occur while the plugin is running. YASARA displays the main error message on screen, and a complete traceback in the console which you can bring up by pressing <SPACE>. After correcting the error, you can simply rerun the plugin, you DO NOT have to restart YASARA.

If you want to print debug statements to trace a problem, this is easily done using

```
print "MyMessage"
```

in Yanaconda and

```
yasara.write(WhatEver)
```

in Python plugins. 'WhatEver' does not have to be a string, but just anything you can pass to Python's print function. DO NOT use Python's print function directly, because this fails under Windows unless you also flush the output buffer with sys.stdout.flush()

If your Python plugin hangs in an infinite loop, click on Options > Stop plugin. This will terminate your plugin as soon as it tries to print something or calls a YASARA command. If the plugin does not do any of these things, YASARA will also hang until you kill the Python task manually from the Windows Task Manager, with the Linux 'kill' command or with the MacOSX 'Activity Monitor' (can be found in the Applications/Utilities folder).