The Programmer's Guide is under construction.

A working knowledge of Python is required for the Programmer's Guide to be useful. The Python Tutorial is recommended for programmers who want a quick introduction to Python. Non-programmers should consult the Beginner's Guide to Python for some suggestions on where to begin.

If all you want to do is loop through a series of data files and perform a series of Chimera commands on them (possibly including writing out results), this extremely basic primer covers how to do that. For anything more complicated please keep reading.

The Chimera Programmer's Guide main component is an Examples section demonstrating various useful Chimera programming interfaces. There are also example scripts for performing various tasks that you may be able to use as a jumping off point for developing your own script. We recommend starting with the Examples, and then either using an example script or examining the Chimera source code for an extension that does something similar to what you want to do and working from there. You can also use Chimera's IDLE programming shell with the dir(obj) and help(obj) functions to find out more about the attributes and methods of particular objects. The FAQ (below) can also be helpful and questions to the chimera-dev mailing list are usually quickly answered.

Also provided:

- For convenience, the help() output for the main Chimera classes:
  - Molecule
  - Residue
  - Atom
  - Bond
- an FAQ with brief answers to common programming questions
- how to create surface models
- making tools scene- and animation-aware
- Chimera's C++ source code online in the SVN repository or packaged for download.
- Handouts from the summer 2008 Chimera programming class describing how to write MD Movie scripts and custom MultAlign Viewer headers and detailing the most generically useful molecular data attributes and Chimera module functions.
- a set of guidelines for Chimera menu/widget text

Please see the Chimera documentation index and Chimera home page for other types of information.

This locally installed Chimera documentation can be searched using "Search Documentation" in the Chimera Help menu.

Looping Through Data Files and Running Chimera Commands on Them

...and not much else

Scenario

This primer covers the scenario where you have a set of data files in a folder and want to perform a series of Chimera commands on each one in turn, writing out some kind of results file (***e.g.*** image; structure measurements; modified data file) for each. Though this primer provides all the Python code needed to carry out the scenario tasks, it would still be beneficial (though not strictly necessary) for you to read the Python Tutorial (at least the first few sections) in order to understand the code better and to be able to modify it if needed in ways this primer doesn't cover.

Broad Outline

You will take the code below, modified for your needs, and place it in a file ending with a '.py' suffix. The '.py' suffix will indicate to Chimera that the file should be interpreted as Python code. You could then run the Python code by opening the file with Chimera's **File→Open** dialog or with the open command. If you want to suppress the Chimera interface from appearing during your script processing, you can start Chimera using the `--nogui` option (***e.g.*** `chimera --nogui processData.py`). Note that if your script creates images then you ***must*** start the Chimera interface unless you've downloaded the "headless" version of Chimera (see the download page).

An important fact to know is that any Chimera command can be executed in Python using the `runCommand()` call. For instance, to color all models red and surface them:

```
from chimera import runCommand
runCommand("color red")
runCommand("surf")
```

This makes it simple to perform actions in your Python script as long as you know the equivalent Chimera command.

Scripting Approach

The general scheme used in the script will be to enter the folder containing your data files, gather the names of the files, and then loop through them one by one, performing a series of commands on each. In the example below the data files are PDB files (suffix: .pdb), each of which has a ligand and a receptor. The script focuses on the ligand, attempts to ensure that the receptor isn't obscuring the ligand, surfaces the receptor, and saves an image.

The Script

There are a lot of comments in the script describing the code. Python comments are introduced by the **#** character (as long as it's not inside a quoted string of course).

```
import os
from chimera import runCommand as rc # use 'rc' as shorthand for runCommand
from chimera import replyobj # for emitting status messages

# change to folder with data files
os.chdir("/Users/pett/data")

# gather the names of .pdb files in the folder
file_names = [fn for fn in os.listdir(".") if fn.endswith(".pdb")]

# loop through the files, opening, processing, and closing each in turn
for fn in file_names:
        replyobj.status("Processing " + fn) # show what file we're working on
        rc("open " + fn)
        rc("align ligand ~ligand") # put ligand in front of remainder of molecule
        rc("focus ligand") # center/zoom ligand
        rc("surf") # surface receptor
        rc("preset apply publication 1") # make everything look nice
        rc("surftransp 15") # make the surface a little bit see-through
        # save image to a file that ends in .png rather than .pdb
        png_name = fn[:-3] + "png"
        rc("copy file " + png_name + " supersample 3")
        rc("close all")
# uncommenting the line below will cause Chimera to exit when the script is done
#rc("stop now")
# note that indentation is significant in Python; the fact that
# the above command is exdented means that it is executed after
# the loop completes, whereas the indented commands that
# preceded it are executed as part of the loop.
```

Actual Script File

Here is a link to an actual file containing the script so that you can download it and use it as a starting point for your own script — and save yourself some typing.

Introduction to Examples

The Examples section of the Chimera Programmer's Guide consists of a series of example code, with associated description, that illustrate how to use various Python interfaces exported by Chimera. The target audience for the section is users who are familiar with Python programming; users who wish to learn more about Python can start with the Python Beginner's Guide.

The list of examples includes:

- Chimera's Object Model
- Molecular Editing Using Python
- Creating Molecules Using Python
- Toolbar Buttons
- Packaging an Extension
- Working with the Chimera Extension Manager
- Adding Command-line Commands
- Extension-Specific User Interface
- Colors and Color Wells
- Trigger Notifications
- Selections
- Session Saving
- Preferences
- Help
- Textures and Surfaces
- Registering Selectors
- Atomic Measurements
- Running a Background Process
- Writing a C/C++ extension

Each example starts with a short description of the functionality that it demonstrates, followed by sample code and detailed commentary, and ends with instructions on how to execute the sample code.

```
Help on class Molecule in module _molecule:

class Molecule(_chimera.Model)
 |  Molecule() -> Molecule
 |
 |  Method resolution order:
 |      Molecule
 |      _chimera.Model
 |      _chimera.Selectable
 |      libwrappy2.WrapPy
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __hash__(...)
 |      x.__hash__() <==> hash(x)
 |
 |  __init__(...)
 |      x.__init__(...) initializes x; see help(type(x)) for signature
 |
 |  addPDBHeader(...)
 |      addPDBHeader(k: unicode, h: unicode)
 |
 |  allRings(...)
 |      allRings(crossResidues: bool, allSizeThreshold: int) -> set of Ring
 |
 |  atomCoordinatesArray(...)
 |      atomCoordinatesArray() -> object
 |
 |  atomGroups(...)
 |      atomGroups(numAtoms: int) -> list of list of Atom
 |
 |  computeIdatmTypes(...)
 |      computeIdatmTypes()
 |
 |  computeSecondaryStructure(...)
 |      computeSecondaryStructure(energyCutoff: float = -0.5, minHelixLength: int = 3, minStrandLength: int
= 3, info: __unknown__ = None)
 |
 |  decrHyds(...)
 |      decrHyds()
 |
 |  deleteAtom(...)
 |      deleteAtom(element: Atom)
 |
 |  deleteBond(...)
 |      deleteBond(element: Bond)
 |
 |  deleteCoordSet(...)
 |      deleteCoordSet(element: CoordSet)
 |
 |  deleteResidue(...)
 |      deleteResidue(element: Residue)
 |
 |  findAtom(...)
 |      findAtom(i: int) -> Atom
 |
 |  findBond(...)
 |      findBond(i: int) -> Bond
 |
 |  findBondRot(...)
 |      findBondRot(bond: Bond) -> BondRot
 |
 |  findCoordSet(...)
 |      findCoordSet(i: int) -> CoordSet
 |
 |  findResidue(...)
```

```
|      findResidue(i: int) -> Residue
|      findResidue(rid: MolResId, type: (str|None) = None) -> Residue
|
|  incrHyds(...)
|      incrHyds()
|
|  indexedAtoms(...)
|      indexedAtoms(indices: object) -> list of Atom
|
|  loadAllFrames lambda self
|
|  metalComplexGroup = _getMetalPbg(mol, **kw)
|
|  minimumRings(...)
|      minimumRings(crossResidues: bool = false) -> set of Ring
|
|  moveResAfter(...)
|      moveResAfter(from: Residue, to: Residue)
|
|  newAtom(...)
|      newAtom(n: unicode, e: Element, coordIndex: int = -1) -> Atom
|
|  newBond(...)
|      newBond(a0: Atom, a1: Atom) -> Bond
|
|  newCoordSet(...)
|      newCoordSet(key: int) -> CoordSet
|      newCoordSet(key: int, size: int) -> CoordSet
|
|  newResidue(...)
|      newResidue(t: unicode, rid: MolResId, neighbor: (Residue|None) = None, after: bool = true) -> Residue
|      newResidue(t: unicode, chain: unicode, pos: int, insert: str, neighbor: (Residue|None) = None,
after: bool = true) -> Residue
|
|  numHyds(...)
|      numHyds() -> int
|
|  primaryAtoms(...)
|      primaryAtoms() -> list of Atom
|
|  printComponents(...)
|      printComponents(os: writable file-like)
|
|  pruneShortBonds(...)
|      pruneShortBonds()
|
|  pseudoBondMgr(...)
|      pseudoBondMgr(cs: (CoordSet|None) = None) -> PseudoBondMgr
|
|  reorderResidues(...)
|      reorderResidues(residues: list of Residue)
|
|  residueAfter(...)
|      residueAfter(r: Residue) -> Residue
|
|  residueBefore(...)
|      residueBefore(r: Residue) -> Residue
|
|  ribbonCoordinates(...)
|      ribbonCoordinates(a: Atom) -> chimera.Point
|
|  rootForAtom(...)
|      rootForAtom(a: Atom, ignoreBreakPoints: bool) -> Root
|
|  roots(...)
|      roots(ignoreBreakPoints: bool) -> list of Root
|
|  sequence = getSequence(molecule, chainID, **kw)
```

```
|      Get the Sequence of the specified chain
|
|      Uses the getSequences function (below) and accepts the same
|      keywords.  Throws KeyError if the specified chain isn't found,
|      and AssertionError if there are multiple chains with the
|      specified ID.
|
|  sequences = getSequences(molecule, asDict=False)
|      return all non-trivial sequences in a molecule
|
|      This function is also available as molecule.sequences(...)
|
|      returns a list of sequences for the given molecule, one sequence per
|      multi-residue chain.  The sequence name is "Chain X" where X is
|      the chain ID, or "Principal chain" if there is no chain ID.
|
|      The 'residues' attribute of each sequence is a list of the
|      residues for that sequence, and the attribute 'resmap' is a
|      dictionary that maps residue to sequence position (zero-based).
|      The 'residues' attribute will self-delete if the corresponding
|      model is closed.
|
|      If 'asDict' is true, return a dictionary of Sequences keyed on
|      chain ID (can throw AssertionError if multiple chains have same ID),
|      otherwise return a list.
|
|  setAllPDBHeaders(...)
|      setAllPDBHeaders(hs: dict of (unicode, list of unicode))
|
|  setPDBHeader(...)
|      setPDBHeader(k: unicode, v: list of unicode)
|
|  traverseAtoms(...)
|      traverseAtoms(root: Root) -> list of Atom
|
|  traverseBonds(...)
|      traverseBonds(root: Root) -> list of Bond
|
|  updateRibbonData(...)
|      updateRibbonData() -> bool
|
|  useAsRoot(...)
|      useAsRoot(newRoot: Atom)
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|
|  activeCoordSet
|      CoordSet
|
|  aromaticColor
|      chimera.Color
|
|  aromaticDisplay
|      bool
|
|  aromaticLineType
|      int
|
|  aromaticMode
|      int
|
|  atoms
|      list of Atom
|
|  atomsMoved
```

```
|      set of Atom
|
|  autochain
|      bool
|
|  ballScale
|      float
|
|  bonds
|      list of Bond
|
|  coordSets
|      dict of (int, CoordSet)
|
|  idatmValid
|      bool
|
|  lineType
|      int
|
|  lineWidth
|      float
|
|  lowerCaseChains
|      bool
|
|  mol2comments
|      list of unicode
|
|  mol2data
|      list of unicode
|
|  numAtoms
|      unsigned int
|
|  numBonds
|      unsigned int
|
|  numResidues
|      unsigned int
|
|  pdbHeaders
|      dict of (unicode, list of unicode)
|
|  pdbVersion
|      int
|
|  pointSize
|      float
|
|  residueLabelPos
|      int
|
|  residues
|      list of Residue
|
|  ribbonHidesMainchain
|      bool
|
|  ribbonInsideColor
|      chimera.Color
|
|  ribbonSmoothing
|      int
|
|  ribbonStiffness
|      float
|
```

```
|  ribbonType
|      int
|
|  showStubBonds
|      bool
|
|  stickScale
|      float
|
|  structureAssigned
|      bool
|
|  surfaceColor
|      chimera.Color
|
|  surfaceOpacity
|      float
|
|  vdwDensity
|      float
|
|  wireStipple
|      (int, int)
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  CentroidAll = 0
|
|  CentroidBackbone = 1
|
|  Circle = 0
|
|  DefaultBondRadius = 0.20000000298023224
|
|  DefaultOffset = -1e+99
|
|  Disk = 1
|
|  PrimaryAtom = 2
|
|  RSM_COIL = 2
|
|  RSM_STRAND = 1
|
|  RT_BSPLINE = 0
|
|  RT_CARDINAL = 1
|
|  __new__ =
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  ----------------------------------------------------------------------
|  Methods inherited from _chimera.Model:
|
|  __str__ = _labelFunc(item)
|
|  addAssociatedModel(...)
|      addAssociatedModel(model: Model, propagateAttrs: bool = true)
|
|  associatedModels(...)
|      associatedModels() -> list of Model
|
|  bbox(...)
|      bbox() -> bool, BBox
|
|  bsphere(...)
|      bsphere() -> bool, Sphere
```

```
|
|  destroy(...)
|      destroy()
|
|  frontPoint(...)
|      frontPoint(p1: Point, p2: Point) -> bool, float
|
|  intersects(...)
|      intersects(p: Plane) -> bool
|
|  removeAssociatedModel(...)
|      removeAssociatedModel(model: Model)
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from _chimera.Model:
|
|  clipPlane
|      Plane
|
|  clipThickness
|      float
|
|  color
|      Color
|
|  display
|      bool
|
|  id
|      int
|
|  name
|      unicode
|
|  openState
|      OpenState
|
|  silhouette
|      bool
|
|  subid
|      int
|
|  useClipPlane
|      bool
|
|  useClipThickness
|      bool
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from _chimera.Model:
|
|  selLevel = 1
|
|  ----------------------------------------------------------------------
|  Methods inherited from _chimera.Selectable:
|
|  oslChildren(...)
|      oslChildren() -> list of Selectable
|
|  oslIdent(...)
|      oslIdent(start: int = SelDefault, end: int = SelDefault) -> unicode
|
|  oslLevel(...)
|      oslLevel() -> int
|
|  oslParents(...)
|      oslParents() -> list of Selectable
```

```
|
|  oslTestAbbr(...)
|      oslTestAbbr(a: OSLAbbreviation) -> bool
|
|  ----------------------------------------------------------------------
|  Static methods inherited from _chimera.Selectable:
|
|  count(...)
|      count() -> int
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from libwrappy2.WrapPy:
|
|  __destroyed__
|      true if underlying C++ object has disappeared
```

```
Help on class Residue in module _molecule:

class Residue(_chimera.Selectable)
 |  Not instantiable from Python
 |
 |  Method resolution order:
 |      Residue
 |      _chimera.Selectable
 |      libwrappy2.WrapPy
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __cmp__(...)
 |      x.__cmp__(y) <==> cmp(x,y)
 |
 |  __eq__(...)
 |      x.__eq__(y) <==> x==y
 |
 |  __ge__(...)
 |      x.__ge__(y) <==> x>=y
 |
 |  __gt__(...)
 |      x.__gt__(y) <==> x>y
 |
 |  __hash__(...)
 |      x.__hash__() <==> hash(x)
 |
 |  __le__(...)
 |      x.__le__(y) <==> x<=y
 |
 |  __lt__(...)
 |      x.__lt__(y) <==> x x!=y
 |
 |  __str__ = _labelFunc(item)
 |
 |  addAtom(...)
 |      addAtom(element: Atom)
 |
 |  atomNames(...)
 |      atomNames() -> set of unicode
 |
 |  bestAltLoc(...)
 |      bestAltLoc() -> str
 |
 |  bondedResidues(...)
 |      bondedResidues() -> list of Residue
 |
 |  bondedTo(...)
 |      bondedTo(r: Residue) -> bool
 |
 |  currentLabelOffset(...)
 |      currentLabelOffset() -> chimera.Vector
 |
```

```
|  findAtom(...)
|      findAtom(name: unicode) -> Atom
|      findAtom(name: unicode, altLoc: str) -> Atom
|
|  findRangeAtoms(...)
|      findRangeAtoms(name: unicode) -> dict of (unicode, list of Atom))
|
|  hasRibbon(...)
|      hasRibbon() -> bool
|
|  hasSurfaceCategory(...)
|      hasSurfaceCategory(category: unicode) -> bool
|
|  labelCoord(...)
|      labelCoord() -> chimera.Point
|
|  removeAtom(...)
|      removeAtom(element: Atom)
|
|  ribbonBinormals(...)
|      ribbonBinormals() -> GeometryVector
|
|  ribbonCenters(...)
|      ribbonCenters() -> GeometryVector
|
|  ribbonFindStyle(...)
|      ribbonFindStyle() -> RibbonStyle
|
|  ribbonFindStyleType(...)
|      ribbonFindStyleType() -> int
|
|  ribbonFindXSection(...)
|      ribbonFindXSection(mode: int) -> RibbonXSection
|
|  ribbonNormals(...)
|      ribbonNormals() -> GeometryVector
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  getDefaultRibbonStyle(...)
|      getDefaultRibbonStyle(ss: int) -> RibbonStyle
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|
|  altLocs
|
|  atoms
|      list of Atom
|
|  atomsMap
|      dict of (unicode, list of Atom))
```

```
|
|  chi1
|
|  chi2
|
|  chi3
|
|  chi4
|
|  fillColor
|      chimera.Color
|
|  fillDisplay
|      bool
|
|  fillMode
|      int
|
|  hasNucleicAcidSugar
|      bool
|
|  heavyAtomCount
|      int
|
|  id
|      MolResId
|
|  isHelix
|      bool
|
|  isHet
|      bool
|
|  isIsolated
|      bool
|
|  isMetal
|      bool
|
|  isSheet
|      bool
|
|  isStrand
|      bool
|
|  kdHydrophobicity
|      object
|
|  label
|      unicode
|
|  labelColor
|      chimera.Color
|
|  labelOffset
```

```
 |      returns 3-tuple (or None if not set)
 |      accepts 3-tuple, Vector, or None
 |
 |  molecule
 |      Molecule
 |
 |  numAtoms
 |      int
 |
 |  phi
 |
 |  psi
 |
 |  ribbonColor
 |      chimera.Color
 |
 |  ribbonData
 |      RibbonData
 |
 |  ribbonDisplay
 |      bool
 |
 |  ribbonDrawMode
 |      int
 |
 |  ribbonResidueClass
 |      RibbonResidueClass
 |
 |  ribbonStyle
 |      RibbonStyle
 |
 |  ribbonXSection
 |      RibbonXSection
 |
 |  ssId
 |      int
 |
 |  type
 |      unicode
 |
 |  uniprotIndex
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  RS_ARROW = 3
 |
 |  RS_HELIX = 1
 |
 |  RS_NUCLEIC = 4
 |
 |  RS_SHEET = 2
 |
 |  RS_TURN = 0
 |
```

```
|  Ribbon_2D = 0
|
|  Ribbon_Custom = 3
|
|  Ribbon_Edged = 1
|
|  Ribbon_Round = 2
|
|  Thick = 1
|
|  Thin = 0
|
|  __new__ =
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  selLevel = 2
|
|  ----------------------------------------------------------------------
|  Methods inherited from _chimera.Selectable:
|
|  oslChildren(...)
|      oslChildren() -> list of Selectable
|
|  oslIdent(...)
|      oslIdent(start: int = SelDefault, end: int = SelDefault) -> unicode
|
|  oslLevel(...)
|      oslLevel() -> int
|
|  oslParents(...)
|      oslParents() -> list of Selectable
|
|  oslTestAbbr(...)
|      oslTestAbbr(a: OSLAbbreviation) -> bool
|
|  ----------------------------------------------------------------------
|  Static methods inherited from _chimera.Selectable:
|
|  count(...)
|      count() -> int
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from libwrappy2.WrapPy:
|
|  __destroyed__
|      true if underlying C++ object has disappeared
```

```
Help on class Atom in module _molecule:

class Atom(_chimera.Selectable)
 |  Not instantiable from Python
 |
 |  Method resolution order:
 |      Atom
 |      _chimera.Selectable
 |      libwrappy2.WrapPy
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __hash__(...)
 |      x.__hash__() <==> hash(x)
 |
 |  __str__ = _labelFunc(item)
 |
 |  addBond(...)
 |      addBond(element: Bond)
 |
 |  addPseudoBond(...)
 |      addPseudoBond(element: PseudoBond)
 |
 |  allLocations(...)
 |      allLocations() -> list of Atom
 |
 |  allRings(...)
 |      allRings(crossResidues: bool = false, sizeThreshold: int = 0) -> list of Ring
 |
 |  associated(...)
 |      associated(otherAtom: Atom, category: unicode) -> bool
 |
 |  associations(...)
 |      associations(category: unicode, otherAtom: (Atom|None) = None) -> list of PseudoBond
 |
 |  clearVdwPoints(...)
 |      clearVdwPoints()
 |
 |  connectsTo(...)
 |      connectsTo(a: Atom) -> Bond
 |
 |  coord(...)
 |      coord() -> chimera.Point
 |      coord(cs: CoordSet) -> chimera.Point
 |
 |  coordination(...)
 |      coordination(valueIfUnknown: int = 0) -> int
 |
 |  currentLabelOffset(...)
 |      currentLabelOffset() -> chimera.Vector
 |
 |  findBond(...)
 |      findBond(a: Atom) -> Bond
 |
 |  findPseudoBond(...)
```

```
|      findPseudoBond(i: int) -> PseudoBond
|
|  haveBfactor(...)
|      haveBfactor() -> bool
|
|  haveOccupancy(...)
|      haveOccupancy() -> bool
|
|  labelCoord(...)
|      labelCoord() -> chimera.Point
|
|  minimumRings(...)
|      minimumRings(crossResidues: bool = false) -> list of Ring
|
|  primaryBonds(...)
|      primaryBonds() -> list of Bond
|
|  primaryNeighbors(...)
|      primaryNeighbors() -> list of Atom
|
|  removeBond(...)
|      removeBond(element: Bond)
|
|  removePseudoBond(...)
|      removePseudoBond(element: PseudoBond)
|
|  revertDefaultRadius(...)
|      revertDefaultRadius()
|
|  rootAtom(...)
|      rootAtom(ignoreBreakPoints: bool) -> Atom
|
|  setCoord(...)
|      setCoord(c: chimera.Point)
|      setCoord(c: chimera.Point, cs: CoordSet)
|
|  shown(...)
|      shown() -> bool
|
|  shownColor(...)
|      shownColor() -> chimera.Color
|
|  traverseFrom(...)
|      traverseFrom(ignoreBreakPoints: bool) -> Atom
|
|  vdwPoints(...)
|      vdwPoints() -> list of tuple(chimera.Point, chimera.Vector)
|
|  xformCoord(...)
|      xformCoord() -> chimera.Point
|      xformCoord(cs: CoordSet) -> chimera.Point
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  getIdatmInfoMap(...)
|      getIdatmInfoMap() -> dict of (unicode, Atom.IdatmInfo)
|
```

```
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|
|  altLoc
|      str
|
|  anisoU
|      object
|
|  bfactor
|      float
|
|  bonds
|      list of Bond
|
|  bondsMap
|      dict of (Atom, Bond)
|
|  color
|      chimera.Color
|
|  coordIndex
|      unsigned int
|
|  defaultRadius
|      float
|
|  display
|      bool
|
|  drawMode
|      int
|
|  element
|      Element
|
|  hide
|      bool
|
|  idatmIsExplicit
|      bool
|
|  idatmType
|      unicode
|
|  label
|      unicode
|
|  labelColor
|      chimera.Color
|
|  labelOffset
|      returns 3-tuple (or None if not set)
|      accepts 3-tuple, Vector, or None
|
|  minimumLabelRadius
```

```
|      float
|
|  molecule
|      Molecule
|
|  name
|      unicode
|
|  neighbors
|      list of Atom
|
|  numBonds
|      unsigned int
|
|  occupancy
|      float
|
|  pseudoBonds
|      list of PseudoBond
|
|  radius
|      float
|
|  residue
|      Residue
|
|  serialNumber
|      int
|
|  surfaceCategory
|      unicode
|
|  surfaceColor
|      chimera.Color
|
|  surfaceDisplay
|      bool
|
|  surfaceOpacity
|      float
|
|  vdw
|      bool
|
|  vdwColor
|      chimera.Color
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  Ball = 3
|
|  Dot = 0
|
|  EndCap = 2
|
|  IdatmInfo =
|      IdatmInfo() -> IdatmInfo
```

```
|      IdatmInfo(_x: Atom.IdatmInfo) -> IdatmInfo
|
|  Ion = 0
|
|  Linear = 2
|
|  Planar = 3
|
|  Single = 1
|
|  Sphere = 1
|
|  Tetrahedral = 4
|
|  UNASSIGNED = 4294967295
|
|  __new__ =
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  selLevel = 3
|
|  ----------------------------------------------------------------------
|  Methods inherited from _chimera.Selectable:
|
|  oslChildren(...)
|      oslChildren() -> list of Selectable
|
|  oslIdent(...)
|      oslIdent(start: int = SelDefault, end: int = SelDefault) -> unicode
|
|  oslLevel(...)
|      oslLevel() -> int
|
|  oslParents(...)
|      oslParents() -> list of Selectable
|
|  oslTestAbbr(...)
|      oslTestAbbr(a: OSLAbbreviation) -> bool
|
|  ----------------------------------------------------------------------
|  Static methods inherited from _chimera.Selectable:
|
|  count(...)
|      count() -> int
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from libwrappy2.WrapPy:
|
|  __destroyed__
|      true if underlying C++ object has disappeared
```

```
Help on class Bond in module _molecule:

class Bond(_chimera.Selectable)
 |  Not instantiable from Python
 |
 |  Method resolution order:
 |      Bond
 |      _chimera.Selectable
 |      libwrappy2.WrapPy
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __hash__(...)
 |      x.__hash__() <==> hash(x)
 |
 |  __str__ = _labelFunc(item)
 |
 |  allRings(...)
 |      allRings(crossResidues: bool = false, sizeThreshold: int = 0) -> list of Ring
 |
 |  contains(...)
 |      contains(a: Atom) -> bool
 |
 |  currentLabelOffset(...)
 |      currentLabelOffset() -> chimera.Vector
 |
 |  findAtom(...)
 |      findAtom(i: int) -> Atom
 |
 |  labelCoord(...)
 |      labelCoord() -> chimera.Point
 |
 |  length(...)
 |      length() -> float
 |
 |  minimumRings(...)
 |      minimumRings(crossResidues: bool = false) -> list of Ring
 |
 |  otherAtom(...)
 |      otherAtom(a: Atom) -> Atom
 |
 |  shown(...)
 |      shown() -> bool
 |
 |  sqlength(...)
 |      sqlength() -> float
 |
 |  traverseFrom(...)
 |      traverseFrom(ignoreBreakPoints: bool) -> Bond
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
```

```
|  __dict__
|
|  atoms
|      2-tuple of Atom
|
|  color
|      chimera.Color
|
|  display
|      int
|
|  drawMode
|      int
|
|  halfbond
|      bool
|
|  label
|      unicode
|
|  labelColor
|      chimera.Color
|
|  labelOffset
|      returns 3-tuple (or None if not set)
|      accepts 3-tuple, Vector, or None
|
|  molecule
|      Molecule
|
|  radius
|      float
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  Always = 1
|
|  Never = 0
|
|  Smart = 2
|
|  Spring = 2
|
|  Stick = 1
|
|  Wire = 0
|
|  __new__ =
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  selLevel = 4
|
|  ----------------------------------------------------------------------
|  Methods inherited from _chimera.Selectable:
```

```
|
|  oslChildren(...)
|      oslChildren() -> list of Selectable
|
|  oslIdent(...)
|      oslIdent(start: int = SelDefault, end: int = SelDefault) -> unicode
|
|  oslLevel(...)
|      oslLevel() -> int
|
|  oslParents(...)
|      oslParents() -> list of Selectable
|
|  oslTestAbbr(...)
|      oslTestAbbr(a: OSLAbbreviation) -> bool
|
|  ----------------------------------------------------------------------
|  Static methods inherited from _chimera.Selectable:
|
|  count(...)
|      count() -> int
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from libwrappy2.WrapPy:
|
|  __destroyed__
|      true if underlying C++ object has disappeared
```

Chimera Programming Frequently Asked Questions

Last updated May 10, 2010. The most recent copy of the FAQ is at http://www.cgl.ucsf.edu/chimera/docs/ProgrammersGuide/faq.html

1) Where is the programming documentation?

The Programming Examples are a good source of information. More information can be gleamed from the C++ header files for the Chimera objects. Those header files are available in the source code download. Another source of object info is the `help()` function in the Chimera's IDLE Python shell (under General Controls). For example, `help(chimera.Atom)` will show (C++) methods and attributes of Atom objects. Even more information is available via chimera developer's mailing list, chimera-dev@cgl.ucsf.edu. The archived mailing list is at http://www.cgl.ucsf.edu/pipermail/chimera-dev.

2) How to emulate command-line functionality.

Commands available at the type-in command line are almost all implemented in the Midas module's __init__.py file. You can use the commands for convenience in implementing the same functionality in your extension. For example, to color iron atoms red:

```
import Midas
Midas.color('red', '@/element=Fe')
```

A few commands related to processing command text (***e.g.***handling files of commands) are in Midas.midas_text. One in particular, makeCommand(), allows you to use command-line syntax directly instead of determining the proper arguments to a Midas module function. So the above example of coloring atoms red would look like this using runCommand():

```
from chimera import runCommand
runCommand("color red @/element=Fe")
```

Note that if the command text may contain errors (***e.g.***it is based on user input), `runCommand()` can raise `MidasError` (defined in the Midas module) so in such cases you may want to embed the `runCommand()` in a `try/except` block.

In pre-1.0 build 2107 versions of Chimera, the `runCommand()` convenience function doesn't exist, so you'd have to use the functionally identical `makeCommand()` as follows:

```
import Midas
from Midas.midas_text import makeCommand
makeCommand("color red @/element=Fe")
```

3) Passing arguments to scripts.

Use the `--script` option to invoke a Python script after all of the other arguments have been processed. If more than than one script option is given, the scripts are executed in the order given. Each script is executed using the arguments enclosed along with it in quotes. Any data files specified in the shell command line are opened before the script is called. For example:

```
chimera --nogui --nostatus --script "script.py -r 2.3 -- -1.pdb" -- -4.pdb
```

Chimera would open the `-4.pdb` file, and invoke `script.py` with the runscript command so `sys.argv` would be set to `['script.py', '-r', '2.3', '--', '-1.pdb']`. The `--` argument terminates the options list and is only necessary if the next non-option argument has a leading dash.

3a) Turning scripts into programs.

To make your Python script look like any other shell program, you could provide an executable shell script as shown below. The shell script accepts a subset of chimera options and options for the Python script, adds in a chimera option to show the Reply Log at startup, then packages the Python script options into a single chimera option, and invokes chimera with those options.

```
#!/bin/bash
PYSCRIPT=PATH-TO-PYTHON_SCRIPT.py
CHIMERA=PATH-TO-CHIMERA
# Parse arguments to decide which are script arguments
# and which are chimera arguments.  In this case, --debug,
# --stereo, -n, --nogui, --nostatus, and --silent are chimera
# arguments.  And -r, and --radius are the script arguments.
# Note that accepting --argument options in shell scripts
# depends on having a newer version of getopt.

if `getopt -T >/dev/null 2>&1` ; [ $? = 4 ]
then
        TEMP=`getopt -o nr: --long radius:,debug,stereo:,nogui,nostatus,silent -n "$0" -- "$@"`
else
        TEMP=`getopt nr: "$@"`
fi
if [ $? != 0 ]
then
        printf "Usage: %s: [-r|--radius value] args\n" $0
        exit 2
fi
eval set -- "$TEMP"

# set initial chimera arguments, in this case always show the Reply Log
```

```
CHARGS=(--start "Reply Log")
while [ $1 != -- ]
do
        case "$1" in
        -r|--radius)
                PYSCRIPT="$PYSCRIPT $1 '$2'"
                shift 2;;
        -n|--debug|--nogui|--nostatus|--silent)
                CHARGS[${#CHARGS[@]}]=$1
                shift;;
        --stereo)
                CHARGS[${#CHARGS[@]}]=$1
                CHARGS[${#CHARGS[@]}]="$2"
                shift 2;;
        esac
done
shift # skip --
$CHIMERA "${CHARGS[@]}" --script "$PYSCRIPT" "$@"
```

And the Python script would parse its arguments with:

```
import getopt
try:
        opts, args = getopt.getopt(sys.argv[1:], 'r:', ['radius='])
except getopt.error, message:
        raise chimera.NonChimeraError("%s: %s" % (__name__, message))
radius = 1.0
for o in opts:
        if o[0] in ("-r", "--radius"):
                radius = o[1]
assert(len(args) == 0)
```

3b) Installing Python packages into Chimera.

Installing a binary distribution

If you don't have a compiler, ***e.g.*** on Microsoft Windows, or if you want to use a binary distribution of the Python package, you will need to do the binary installation into the system Python and copy the appropriate files into Chimera's Python.

Installing a source distribution

> The following assumes that (1) CHIMERA is replaced by the path to the Chimera installation, (2) you can run chimera from a shell or terminal window (Command Prompt in Microsoft Windows), (3) you have a source distribution of the Python package, (4) if the Python package has C or C++ code, you have the appropriate compiler, and (5) that the working directory is the Python package's source directory.

Chimera only includes the default Python packaging mechanisms, so Python packages should be installed using setup.py as follows:

```
CHIMERA/bin/chimera --nogui --silent --script "setup.py install"
```

Sometimes the Python package prefers to be installed with easy_install or pip. In those cases, you would install

easy_install first by getting the source code (the setuptools-VERSION.tar.gz file, instead of one of the .egg files), and installing as shown above. Then to use easy_install, you would:

```
CHIMERA/bin/chimera --nogui --silent --script "CHIMERA/bin/easy_install install"
```

To install and use pip, you would first install pip with easy_install as shown above. And then to use pip, you would analogously:

```
CHIMERA/bin/chimera --nogui --silent --script "CHIMERA/bin/pip install"
```

4) How to rotate/translate the camera.

Camera always points in -z direction. There is no way to rotate it. Instead, rotate all of the models.

```
>>> v = chimera.viewer
>>> c = v.camera
>>> print c.center
(5.9539999961853027, -2.186500072479248, 10.296500205993652)
>>> c.center = (3, 2.5, 10)     # to translate camera
>>> v.scaleFactor = 1.5         # to zoom camera
```

5) How to rotate/translate individual models.

The Xform object model.openState.xform retrieves a copy of the rotation and translation transformation for a model.

```
>>> om = chimera.openModels
>>> mlist = om.list()
>>> m = mlist[0]
>>> axis = chimera.Vector(1, 0, 0)
>>> angle = 90                  # degrees
>>> xf = chimera.Xform.rotation(axis, angle)
>>> print m.openState.xform     # 3x3 rotation matrix
                                # last column is translation
0.982695 0.121524 0.139793 -1.07064
0.0250348 0.660639 -0.750287 6.83425
-0.183531 0.740803 0.646164 6.35578
>>> m.openState.globalXform(xf)
```

Another method to change the transform

```
>>> curxform = m.openState.xform # get copy (not reference)
>>> xf.premultiply(curxform)     # changes xf
>>> m.openState.xform = xf       # set it
```

To rotate relative to model's data axes use

```
>>> m.openState.localXform(xf)
```

or

```
>>> curxform = m.openState.xform # get copy (not reference)
>>> xf.multiply(curxform)       # changes xf
>>> m.openState.xform = xf      # set it
```

6) How to save the current view as an image.

```
import Midas
Midas.copy(file='/home/goddard/hoohoo.png', format='PNG')
# format can be 'PNG', 'JPEG', 'TIFF', 'PS', 'EPS'
```

7) Some attributes return a copy of an object.

```
>>> xf = model.openState.xform  # xf is a copy of the model's Xform matrix.
>>> xf.zRotate(45)              # This will not rotate the model.
```

```
>>> c = model.atoms[0].color    # c is the MaterialColor object for the atom
>>> c.ambientDiffuse = (1,0,0)  # The Atom color changes immediately to red.
```

Some Chimera objects returned as attributes are always copies, some are always references to the original object. Objects that are always copied include Xform, Vector, Point, Sphere, Element, MolResId, Coord, ... Objects that are never copied include Atom, Bond, PseudoBond, CoordSet, Molecule, Residue, RibbonStyle, ... Object that can be copied have a __copy__ method. In order to know if an object type is passed by value is to look at the Chimera C++ header files. Classes without a WrapPy base class are always copied. This base class is part of the C++ to Python interface generation.

8) Explanation of transparency for Surface_Model, MSMSModel and VRMLModel.

Volume viewer isosurfaces are Surface_Model objects defined by the _surface.so C++ module. The Surface_Model interface is given in the surfmodel.h source code file. By default these surfaces use OpenGL (1,1-alpha) blending. This means the color for a triangle is added to an image plus the transparency (= 1-alpha) times the color from triangles in back of this one. As the transparency becomes greater, the brightness of the triangle does not diminish. In fact, more shows through from behind the triangle so it appears brighter. The specular highlights stay bright even if the triangle is black and fully transparent. In Chimera 1730 a Surface_Model attribute transparency_blend_mode was added to allow the more common (alpha,1-alpha) blend mode. This is like the above mode but the triangle color is multiplied by alpha before being added to the image. For highly transparent triangles alpha is close to zero, and the triangle and specular highlights become dim.

MSMS molecular surface models use (alpha, 1-alpha) blending. They also use a 2 pass algorithm which is faster than sorting all the triangles by depth, but gives the strictly correct appearance only when the viewer looks through at most 2 surface layers.

VRML models use (alpha,1) blending. That is they multiply triangle color by alpha, but add in all of the color from triangles further back without reducing it by the transparency factor. This is like complete transparency, where the triangle colors are just scaled by the alpha value. This is horrible looking unless you want complete transparency.

9) Transparency displayed incorrectly when 2 or more models are transparent.

The Chimera architecture only correctly displays transparency when at most one transparent model is shown. Any number of opaque models can also be shown. If two transparent models are shown, one will be drawn after the other. This will make one appear as if it is drawn entirely on top of another even if they in actuality intersect. The rearmost model, may in fact appear to be in front. This is because Chimera sorts the triangles by depth within a single model

as needed for rendering transparency, but is not able to sort triangles by depth across multiple models.

10) Explanation of openState attribute.

The `openState` attribute of a Model controls whether that model is active for motion (`'.active'`), and contains the model's transformation matrix (`'.xform'`) and center of rotation (`'.cofr'`). Since some models must move in synchrony (***e.g.***a molecule and its surface), OpenState instances may shared among multiple models. If you create a model that needs a shared `openState` with another model, then when adding your model to the list of open models with `chimera.openModels.add()`, you should use the `'sameAs'` keyword to specify that other model.

11) Memory leaks in scripts.

In the 1700 release, Chimera uses a substantial amount of memory to hold molecular data, but does not have any large memory leaks to the best of our knowledge (***i.e.*** as structures are closed their memory usage will be reclaimed). However, the memory is reclaimed by a task that runs during idle times, so therefore scripts that loop through many structures, opening and closing them, will have their memory use grow continuously until the script finishes. This will often cause the script to fail as the Chimera process runs out of memory.

You can cause the memory-reclamation task to run during your script by calling:

```
chimera.update.checkForChanges()
```

You would want to call this after closing models in your script. Since the checkForChanges() routine also allows triggers to fire, you might want to also put it after any code that opens models. For example, the code that assigns surface categories to models runs from a trigger callback, so adding a molecular surface may not work as expected if checkForChanges() is not called after a molecule is opened.

In post-1700 releases and snapshots, checkForChanges() is called automatically when models are opened or closed, so you will not need to insert these calls into your code.

12) Getting the size of a volume data set.

Here's some code to find the size of a volume data set.

```
import VolumeViewer
d = VolumeViewer.volume_dialog()
d0 = d.data_sets[0]     # There may be more than one data set opened
                        # You could look at each one's name (= file name)
                        # to find the one you want.
data = d0.data          # This is a Grid_Data object defined in
                        # VolumeData/griddata.py
xsize, ysize, zsize = data.size
```

13) How to write out a PDB file containing the crystal unit cell.

Here's code that creates the copies of a PDB molecule needed to fill out a crystallographic unit cell, then writes all the copies to a new PDB file. This code uses the standard Chimera 1.0 build 1892 PDBmatrices module, which uses the PDB SMTRY remarks or CRYST1 record to determine the needed transformations.

You can run it without a graphical user interface as follows

```
% chimera --nogui myfile.pdb writeunitcell.py
```

where the writeunitcell.py file contains the script given below.

```
# Get the Molecule that has already been opened
import chimera
m = chimera.openModels.list()[0]

# Get the symmetry matrices
import PDBmatrices
tflist = PDBmatrices.crystal_symmetry_matrices(m.pdbHeaders)

# Get center of bounding box
import UnitCell
center = UnitCell.molecule_center(m)

# Get CRYST1 line from PDB headers
cryst1 = m.pdbHeaders['CRYST1'][0]

# Getting crystal parameters
from PDBmatrices import crystal
cp = crystal.pdb_cryst1_parameters(cryst1)
a,b,c,alpha,beta,gamma = cp[:6]

# Adjust matrices to produce close packing.
cpm = PDBmatrices.close_packing_matrices(tflist, center, a, b, c, alpha, beta, gamma)

# Apply transformations to copies of Molecule
mlist = []
from PDBmatrices import matrices
path = m.openedAs[0]                    # Path to original PDB file
for tf in cpm:
    xf = matrices.chimera_xform(tf)     # Chimera style transform matrix
    m.openState.globalXform(xf)
    mlist.append(m)
    m = chimera.openModels.open(path)[0]        # Open another copy

# Write PDB file with transformed copies of molecule
import Midas
out_path = 'unitcell.pdb'
Midas.write(mlist, None, out_path)
```

14) How to access files within the Chimera app on Macs.

One way to access the Python code and other Chimera files on a Mac is to right-click on the Chimera application icon and choose "Show Package Contents" from the resulting pop-up menu. Another way is to use Terminal.app and the command "cd" to navigate into the Chimera.app directory and its subdirectories. Most files of interest can be found under Contents/Resources/share/.

Description of _surface module

Chimera version 1.2540 (July 9, 2008)

The _surface module is for displaying surfaces in Chimera. It was developed for displaying isosurfaces of volume data. Chimera has a second way to display surfaces: VRML models.

The _surface module defines SurfaceModel and SurfacePiece objects that are available from Python. These objects are defined in the C++ header file surfmodel.h.

Surface Specification

A surface is described as a set of triangles that are defined in two arrays. One array contains the vertex xyz positions. If there are N vertices the array is of size N by 3. The second array gives 3 indices for each triangle. These are indices into the vertex xyz position array. This method of representing the surface saves some space since each vertex is used in about 6 triangles in typical triangulated surfaces.

Here is an example that makes surface model with 2 triangles.

```
import _surface
m = _surface.SurfaceModel()

# Xyz vertex positions.
v = [(0,0,0), (1.5,0,0), (0,1.2,0), (0,0,2.3)]

# Define two triangles each defined by 3 indices into vertex list.
vi = [(0,1,2), (3,0,2)]

# Color specified as red, green, blue, opacity (0-1), partially transparent red.
rgba = (1,0,0,.5)

m.addPiece(v, vi, rgba)

import chimera
chimera.openModels.add([m])
```

The vertex position array must contain floating point values. The index array must contain integer values. The arrays can be nested tuples, or lists, or numpy arrays can be used.

Surface Pieces

A surface model can display several sets of triangles. The sets are called surface pieces. The SurfaceModel.addPiece() method creates a SurfacePiece. Surface pieces can be individually added and removed from a SurfaceModel. They do not share vertex xyz arrays or any other properties. Their original purpose was for showing multiple isosurfaces of volume data in a single Chimera model.

Surface Piece Features

A Surface_Piece has methods from controlling its display.

- Show or hide.
- Solid (filled triangles) or Mesh (just edges of triangles) or Dot display styles.
- Single color for whole surface, or separate colors for all the vertices.
- Lighting on/off.
- Two sided or one sided lighting in with Solid style. Mesh style only allows one sided lighting since that is all that OpenGL provides.
- Surface triangles can be changed.
- Save pieces in session files (Chimera 1.4 and newer).

Caution. When defining a triangulated surface it is important to specify the 3 triangle vertices in a consistent order, so that the normal vectors all point towards the same side of the surface. Otherwise the shading produced by lighting will not look right. For volume surfaces, the data gradient direction is used for lighting normals. In other cases surface normals can be calculated for each vertex by adding up the normals of the triangles that share that vertex, then scaling to make the sum have unit length. The orientation of the triangle normals is based on the ordering of the 3 vertices that make up a triangle, and the normal orientation surface piece attribute.

Example

```
import _surface
m = _surface.SurfaceModel()

import chimera
chimera.openModels.add([m])

# For minimum memory use and maximum speed use NumPy arrays for vertices
# and triangles.
from numpy import array, single as floatc, intc

# Xyz vertex positions.
v = array(((0,0,0), (1.5,0,0), (0,1.2,0), (0,0,2.3)), floatc)

# Define two triangles each defined by 3 indices into vertex list.
vi = array(((0,1,2), (3,0,2)), intc)

# Color specified as red, green, blue, opacity (0-1), partially transparent red.
rgba = (1,0,0,.5)

p = m.addPiece(v, vi, rgba)             # This returns a Surface_Piece

p.display = False                       # hide the piece
p.display = True                        # show the piece

p.displayStyle = p.Mesh
p.displayStyle = p.Solid                # default

p.color = (0,0,1,1)                     # change color to blue

# Set the 4 vertices to be red, green, blue and white.
```

```
# These values must be floating point.
# This overrides the previous setting of the piece to blue.
p.vertexColors = [(1.,0,0,1),(0,1,0,1),(0,0,1,1),(1,1,1,1)]

# Clear the vertex colors.  Piece will be blue again.
p.vertexColors = None

p.useLighting = False           # Lighting off
p.useLighting = True            # Lighting on (default)

p.twoSidedLighting = False      # Light one side
p.twoSidedLighting = True       # Light both sides (default)

p.save_in_session = True        # Include when saving sessions

v2 = array(((1.,0,0), (2,0,0), (0,3,0), (0,0,4)), floatc)
vi2 = array(((0,1,2), (3,1,0)), intc)
p.geometry = v2, vi2            # Change the surface for this piece

v3 = array(((0.,1,2), (0,2,3), (0,5,0), (1,1,0)), floatc)
vi3 = array(((1,0,2), (3,2,0)), intc)
rgba3 = (1,0,1,1)
p3 = m.addPiece(v3, vi3, rgba3) # Make another piece
```

Making Tools Scene- and Animation-aware

Chimera version 1.7 (November 5, 2012)

The Animate package implements both scene and animation functionality. Core attributes such as model names, residue types and atom positions are handled automatically by the Animate package. Tools are responsible for saving, restoring and animating objects and attributes that they introduce. Notification of scene and animation events happen through the standard Chimera trigger mechanism.

Scene Triggers

There are two triggers associated with scenes:

`chimera.SCENE_TOOL_SAVE`
: This trigger is fired when a scene is created or updated. The core attributes have already been saved. The trigger data argument to registered handlers is the scene object.

`chimera.SCENE_TOOL_RESTORE`
: This trigger is fired when a scene is restored (and just before an animation sequence completes, see below). The core attributes have already been restored. The trigger data argument to registered handlers is the scene object.

Each scene object has a dictionary, `tool_settings`, for managing tool-specific data. Tools are responsible for creating their own keys and values for storing and accessing data in the dictionary. Singleton tools such as **2DLabels** can use fixed strings such as "2D Labels (gui)" as their keys; multi-instance tools will need to come up with unique keys and make sure that they are not overwriting data for some other tool. Typically, data is saved in `SCENE_TOOL_SAVE` and accessed in `SCENE_TOOL_RESTORE` handlers.

Animation Triggers

`Animate` updates the graphics display during transitions. There are three types of transitions:

key frame transition
: When the user plays the animation, a transition occurs between each sequential pair of timeline items. If the ending item is a key frame, then a playback transition results. To transform from the starting state to the ending key frame state, data values are interpolated over the number of steps associated with the end key frame (the value is stored in the `frames` attribute of the key frame). There is a static "transition" for the initial frame if it has a `non-zero frames` value.

scene transition
: When the user selects a scene (either from the **Scenes** list or by double-clicking on a key frame in the time line) , a transition with default parameters is used. Currently, the "transition" is a one-frame update to the selected scene.

action transition
: When an animation is played and one of the timeline items is an **action** rather than a **key frame**, an action transition results. None of the triggers listed below are fired for action transitions because the target state is unknown.

There are four triggers associated with key frame and scene transitions:

`chimera.ANIMATION_TRANSITION_START`
: This trigger is fired when a transition from one key frame to the next starts. No core attributes have been updated yet, so Chimera data represents the initial state of the transition. The trigger data argument to registered handlers is the transition object.

`chimera.ANIMATION_TRANSITION_STEP`
: This trigger is fired for each step of the transition from one key frame to the next. The core attributes have already been updated. The trigger data argument to registered handlers is the transition object.

`chimera.ANIMATION_TRANSITION_FINISH`
: This trigger is fired after all steps of the transition from one key frame to the next have been completed. The trigger data argument to registered handlers is the transition object.

`chimera.SCENE_TOOL_RESTORE`
: This trigger is fired just prior to the `ANIMATION_TRANSITION_FINISH`. The purpose for firing this trigger is to simplify writing tools that do not need step-by-step animation yet want to restore state when a scene state is reached. These tools can just register for the `SCENE_TOOL_RESTORE` trigger and completely ignore the `ANIMATION_TRANSITION_*` triggers. Tools that wants to handle both step-by-step animation and scene restoration will need to avoid getting the `SCENE_TOOL_RESTORE` trigger twice by deregistering for it in `ANIMATION_TRANSITION_START` and reregistering for it in `ANIMATION_TRANSITION_FINISH`.

The transition object received by handlers has several useful attributes and methods:

`frames`
: The total number of frames in this transition.

`frameCount`
: The step number of the current frame in the transition. For `ANIMATION_TRANSITION_START`, this value should always be zero. For `ANIMATION_TRANSTION_STEP`, this value ranges from 1 to `frames`, inclusive. For `ANIMATION_TRANSITION_FINISH`, this value should be the same as `frames`.

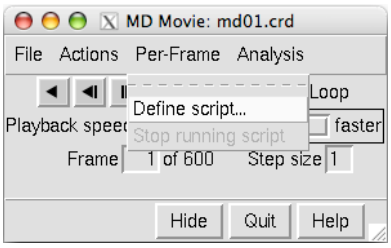
`target()`
: The end point for this transition, usually an instance of `Animate.Keyframe.Keyframe` for movie playback or `Animate.Scene.Scene` for scene restoration using the default transition.

`scene()`
: The scene at the end point for this transition, or None if the target is neither a `Keyframe` nor a `Scene`.

`tool_settings`
: A dictionary for managing tool-specific data in the same manner as scene objects. Unlike the dictionary in scene objects, the `tool_settings` in transition objects are transient. When a transition completes, the dictionary is automatically removed and any stored data will be lost.

MD Movie Scripting

1. Bring up script dialog with Per-Frame➔Define script...

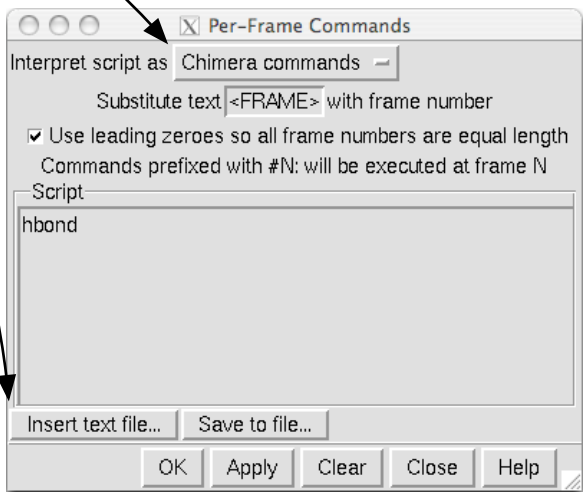


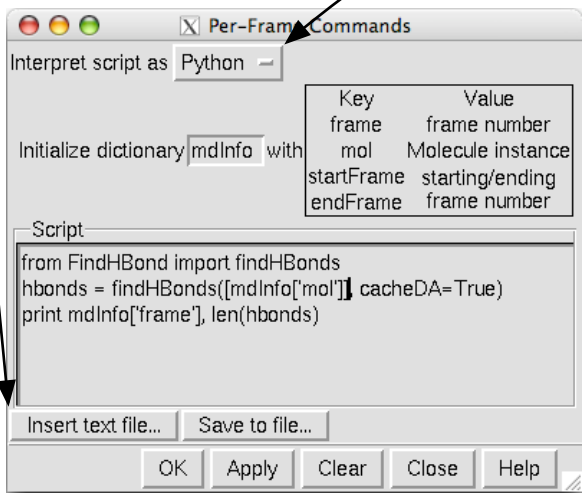
2. Choose scripting language

3. Type in script or read from file

4. Click Apply/OK (executes for current frame)

5. Play/record movie (may want to disable Loop)





Custom MAV Headers

Static Headers

The ability to add pre-computed headers to your alignment is covered well in the MultAlign Viewer documentation.

Dynamic Headers

Read MAVHeader/ChimeraExtension.py for example header definitions. You may also want to look through MultAlignViewer/HeaderSequence.py for useful methods to use/override.

Create a directory containing a ChimeraExtension.py with your header definitions. Don't put the directory inside the Chimera distribution since it will be lost if you upgrade Chimera. To get Chimera to use the headers, add the directory above the directory you created to the Locations list in the Tools preference category.

Your header class should subclass from **DynamicHeaderSequence** or **DynamicStructureHeaderSequence** (the latter if your header values depend on what structures are associated with the alignment). You use the *registerHeaderSequence* function to notify Multalign Viewer of the existence of your header class. The *defaultOn* keyword arg controls whether the header defaults to being shown initially.

The alignment sequences will be available as *self.mav.seqs* .

The one method you absolutely must define yourself is *evaluate(pos)*, which returns the value of the header at alignment position *pos* (indexing starts at zero). The value should be whatever is appropriate for the header, *e.g.* Conservation Percentage would be a number in the range 0-100.

Two methods that you most likely will want to override are *colorFunc(line, pos)* and *depictionVal(pos)*. *colorFunc* returns the color to use at *pos*. The color should be a string that Tk accepts as a color, such as any of the normal color names in Chimera or an "#rrrgggbbb" string. The *line* argument is essentially another copy of *self* and can be ignored in this context. The *depictionVal* method should return a value to use to depict the header at *pos*, either a character, a number in the range 0-1, or None. The *histInfinity* method can be useful for converting an unbounded range of numbers to the range 0-1.

Resources

- Programmer's Examples (www.cgl.ucsf.edu/chimera/docs/ProgrammersGuide)
  - Example scripts: socrates2.cgl.ucsf.edu/trac/chimera/wiki/Scripts
- IDLE (Tools➔General Controls): help(*object*), dir(*object*)
- Python language/modules: www.python.org/doc/current/
- Numpy examples: www.scipy.org/Numpy_Example_List_With_Doc
- Chimera developer mailing list: chimera-dev@cgl.ucsf.edu
- C++ source code: browse SVN atsocrates2.cgl.ucsf.edu/trac/chimera/browser or download from www.cgl.ucsf.edu/chimera/sourcecode.html
  - Python source code included with distribution

Chimera Molecular Data

- *chimera.openModels.list()*: list of open models
  - *modeltypes=[chimera.Molecule]*: restrict list to Molecules
- *m.residues* / *m.atoms* / *m.bonds*: a Molecule's residues / atoms / bonds

Residues

- *type*: LYS, HEM, *etc.*
- *id.position* / *id.chainId* / *id.insertionCode:* number / chain ID / insertion code
- *molecule*: parent Molecule
- *atoms*: list of atoms
- *atomsMap*: dict of atom-name ➔ list of atoms
- *isHelix* / *isStrand*: in helix / strand

Atoms

- *name*: name
- *coord()* / *xformCoord()*: untransformed / transformed coordinates
- *residue* / *molecule*: parent Residue / Molecule
- *bonds*: list of bonds
- *neighbors*: list of bonded atoms
- *primaryBonds()* / *primaryNeighbors()*: same as above but only primary altlocs
- *bondsMap*: dict of bonded-atom ➔ bond
- *color*: Color
- *display*: True if shown
- *drawMode*: one of chimera.Atom.X with X being Dot, Sphere, EndCap, or Ball
- *element*: chemical element (type chimera.Element, settable with string or number)
- *label*: label shown in graphics window
- *radius*: VdW radius

Bonds

- *atoms*: 2-tuple of atoms
- *otherAtom(a)*: [*a* is one of the bond's atoms] other atom in bond
- *drawMode*: one of chimera.Bond.Y with Y being Wire or Stick
- *label*: label shown in graphics window
- *molecule*: parent Molecule
- *length*(): length

Useful Chimera modules/functions

Molecular Measurements

chimera module

functions use Points, which are returned by Atom's *coord()* or *xformCoord()* methods

- *distance* / *sqdistance*
  - *also: a1.coord()*.[*sq*]*distance(a2.coord())* [similar for *xformCoord*]
- *angle* — in degrees
- *dihedral* — in degrees

Molecular Editing

chimera.molEdit module

- *addAtom*
  - if adding in bulk, make sure to specify optional *serialNumber* keyword
- *addBond*
- *addDihedralAtom* – add atom given a bond length / angle / dihedral

look in BuildStructure/__init__.py for examples of creating new Molecules and Residues

Setting/Querying The Selection

chimera.selection module

- *currentAtoms* / *currentBonds* / *currentResidues* / *currentMolecules*: currently selected Atoms / Bonds / Residues / Molecules
- *setCurrent*: set current selection to given items
- *addCurrent* / *addImpliedCurrent*: add given items to current selection
  - the "implied" version also selects endpoint Atoms of added Bonds and connecting Bonds of added Atoms
- *removeCurrent*: remove items from current selection, if present

Miscellaneous

chimera module

- *runCommand*: execute any command-line command (arg is a string)
  - direct Python equivalent usually in Midas module

chimera.colorTable module

- *getColorByName*: get a Color by name

OpenSave module

- *osOpen*: open a file or HTTP URL, with or without compression

chimera.extension module

- *manager.instances*: running dialogs listed at end of Tools menu

Chimera Menu/Widget Text Guidelines

I. Goals

- to promote consistent text usage in Chimera's user interfaces
- to provide guidelines for developers and programmers
- to promote awareness and discussion

II. Font

The default font type and size should be used.

III. Menus

IIIa. General Scheme

```
Primary (one word, noun or verb, capitalized)
  Secondary (words capitalized as in a title)
    tertiary or lower (numeral or lowercase,
                       except proper nouns)
```

Proper nouns include atom types, elements, and extension (tool) names.

Examples:

```
Select
  Chemistry
    element
      other
        Fe-Hg
          Fe
          (etc.)
  Residue
    amino acid category
      aliphatic
      (etc.)
  Selection Mode (replace)
    append
    (etc.)
Actions
  Atoms/Bonds
    wire width
      1
      (etc.)
  Ribbon
    show
    (etc.)
```

```
Tools
  Utilities
    Browser Configuration
    (etc.)
```

IIIb. Usage of Ellipses

The ellipsis string "..." should indicate a menu item that opens an additional interface which requires user input to accomplish the function described by its name (one-shot panels, as opposed to those intended to be left up for a series of user interactions). For now, **Tools** are exempted from this guideline.

We decided that "..." should not indicate a menu item which simply opens an additional interface, since practically all items would then require it.

There also needs to be consistency in whether "..." is preceded by a space; we recommend no space.

Finally, should "..." appear on buttons as well as menu items? If so, the same criteria should apply.

IV. Widgets (GUIs)

This is the broadest grouping, and thus less amenable to standardization. It includes panels and dialog boxes generated by built-in functions as well as extensions to Chimera. General recommendations:

- **Title of Widget**
  - one or more words to appear on the top bar, capitalized as a title, no colon or period at the end; should be the same text as the invoking menu item or button (except sans any "...")
- **Brief Header** for a section
  - capitalized as a title, optional colon at the end (but no colon when sections are treated as "index cards")
- **Longer description of a section**
  - first word capitalized, subsequent words not capitalized unless proper nouns or acronyms; optional colon at the end, no period
- **Instructive statement**
  - first word capitalized, subsequent words not capitalized unless proper nouns or acronyms; no period
  Example:
  Ctrl-click on histogram to add or delete thresholds in the **Volume Viewer Display** panel
- **[box] Description next to a checkbox**
  - first word capitalized, subsequent words not capitalized unless proper nouns or acronyms; no period or question mark
    - exception: when the checkbox indicates a section to be expanded/compacted, the text may be capitalized as a title (instead of only the first word being capitalized).
- **Item name: [blank, color well, slider, pulldown menu or checkbox list]**

or (especially if there are many of these in the widget)
item name: [blank, color well, slider, pulldown menu or checkbox list]
- first word of item name optionally capitalized, subsequent words not capitalized unless proper nouns or acronyms; colon separating the item name from the value(s); options in a pulldown menu or checkbox list not capitalized unless proper nouns or acronyms
    - exception: when the item name and pulldown option together describe a section, both should be capitalized and the colon is optional
      Examples:
      Inspect [Atom/etc.] in the **Selection Inspector**
      Category: [New Molecules/etc.] in the **Preferences Tool**

- **Phrase with [blank, color well, pulldown menu, or checkbox list] embedded**
  - first word capitalized, no colon, period or question mark; the blank (etc.) should not start the phrase
- **Phrase with [button] embedded**
  - 1-2 words actually on the button, others trailing and/or preceding; the first word should be capitalized whether or not on the button; no colon, period or question mark; the button may start the phrase
- buttons marked **OK**, **Apply**, **Cancel**, **Help**
  - common but optional
- widget-specific buttons
  - 1-2 words, each capitalized if the button brings up another panel, at least the first word capitalized otherwise; if another panel is evoked, consider using "..."

UCSF Computer Graphics Laboratory / November 2004

```
import os
from chimera import runCommand as rc # use 'rc' as shorthand for runCommand
from chimera import replyobj # for emitting status messages

# change to folder with data files
os.chdir("/Users/pett/data")

# gather the names of .pdb files in the folder
file_names = [fn for fn in os.listdir(".") if fn.endswith(".pdb")]

# loop through the files, opening, processing, and closing each in turn
for fn in file_names:
	replyobj.status("Processing " + fn) # show what file we're working on
	rc("open " + fn)
	rc("align ligand ~ligand") # put ligand in front of remainder of molecule
	rc("focus ligand") # center/zoom ligand
	rc("surf") # surface receptor
	rc("preset apply publication 1") # make everything look nice
	rc("surftransp 15") # make the surface a little bit see-through
	# save image to a file that ends in .png rather than .pdb
	png_name = fn[:-3] + "png"
	rc("copy file " + png_name + " supersample 3")
	rc("close all")
# uncommenting the line below will cause Chimera to exit when the script is done
#rc("stop now")
# note that indentation is significant in Python; the fact that
# the above command is exdented means that it is executed after
# the loop completes, whereas the indented commands that
# preceded it are executed as part of the loop.
```

Introduction to Examples

The Examples section of the Chimera Programmer's Guide consists of a series of example code, with associated description, that illustrate how to use various Python interfaces exported by Chimera. The target audience for the section is users who are familiar with Python programming; users who wish to learn more about Python can start with the Python Beginner's Guide.

The list of examples includes:

- Chimera's Object Model
- Molecular Editing Using Python
- Creating Molecules Using Python
- Toolbar Buttons
- Packaging an Extension
- Working with the Chimera Extension Manager
- Adding Command-line Commands
- Extension-Specific User Interface
- Colors and Color Wells
- Trigger Notifications
- Selections
- Session Saving
- Preferences
- Help
- Textures and Surfaces
- Registering Selectors
- Atomic Measurements
- Running a Background Process
- Writing a C/C++ extension

Each example starts with a short description of the functionality that it demonstrates, followed by sample code and detailed commentary, and ends with instructions on how to execute the sample code.

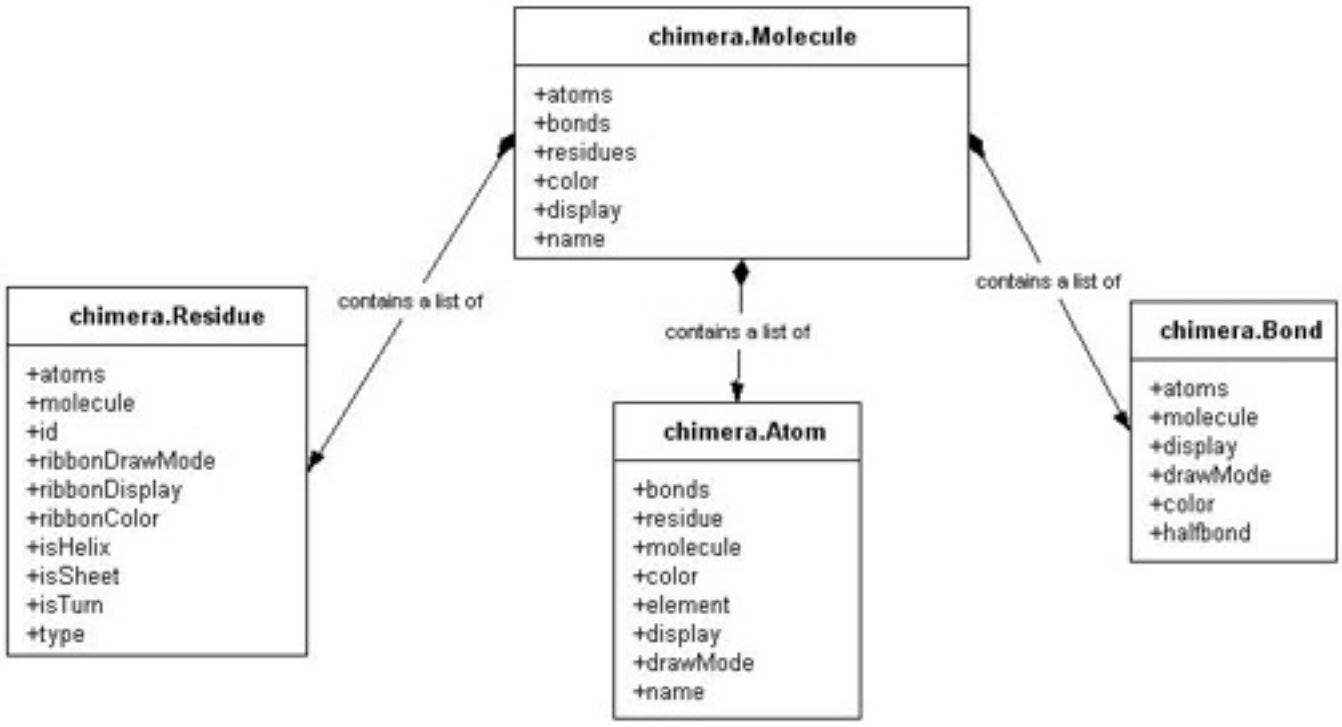
Chimera's Object Model

Purpose

Provide an introduction to the Chimera's object model.

Introduction

The first step to programming in the Chimera environment is to understand its ***object model*** . Chimera uses a hierarchy of objects to represent actual chemical components - atoms, bonds, residues, and so on. Most of Chimera is written in Python, an easy-to-learn, strongly object-oriented scripting language. The use of Python (for flexibility) in combination with C++ (for speed) has led to a highly structured, yet easily navigable, object model that simulates all the chemical components necessitated by a full-featured, intelligent molecular modeling system.
This example will provide a foundation for some of the more complicated programming examples by explaining both the makeup of Chimera's objects and their relationship to one another.



The accompanying image illustrates the general relationship between some of Chimera's fundamental objects - `Atoms`, `Bonds`, `Residues`, and `Molecules`. While this diagram shows several attributes for each object, some attributes have been left out for the sake of simplicity. Only the attributes that are discussed in this tutorial (the basics, and therefore, in some sense the most important) have been shown. You can always do a

```
>>>help(object)
```

in Chimera's IDLE window (essentially a Python interpreter built in to Chimera) to see documentation for **all** the contents of any Chimera object. The attributes associated with these objects can be divided into

roughly two categories: those that contain chemical/structural information (e.g. `Molecule.atoms` or `Atom.idAtmType`) and those that control the visual representation of the data (e.g. `Molecule.color` and `Atom.drawMode`). The following example will demonstrate the use of attributes in the first category. A discussion of those in the second category is saved for another example.

Note on semantics for the following examples:

- In general, anything written in `this font` (fixed width) is referring specifically to some element of code.
- There is sometimes a need to talk specifically about either an ***instance*** of an object, or its ***class***. To distinguish between these two situations, lowercase is used to refer to an instance ("`atom`") and uppercase is used to refer to the class ("`Atom`"). If the reference to an object is ***not*** written in `this font`, then the implication is obvious enough and/or there is no need to make this distinction.
- When discussing the color of a component, there can be some confusion differentiating between the color an object appears to be (how it appears on the screen), and what is assigned to that object's `color` attribute (on the programmatic level). "color" will be used when referring to the former, and "`color`" will be used in dealing with the latter. Example: "Due to Chimera's color hierarchy, the color of an atom (the color it appears to be) may not always reflect its `color` attribute (the color assigned to that atom in the code)"
- Unlike `Atoms` and `Bonds`, `Residues` are not actually visible in and of themselves. It is only when they are drawn as ribbons that they are visible in a model. Thus, when residue color or display is mentioned in the following discussion, it is actually referring to the color/display of the ribbon portion which represents that residue.

Examples in this guide are typically laid out as a downloadable link to a Python script followed by a line-by-line explanation of the script. You may want to read through the explanation in its entirety, or look through the script and refer back to the detailed explanation for the parts you don't understand.

To execute the script, either open the script file with the File→Open menu item or with the open command.

Example *writeMol2.py*

Import Chimera modules used in this example.

```
import chimera
```

First, we'll open up a model to work with. This molecule (4fun) is very small, comprised of just a couple residues, but it is perfect for illustrating some important components of Chimera's object model. For more information on how to open/close models in Chimera, see the "Basic Model Manipulation" Example in the Programmer's Guide (coming soon). For now, just understand that this code opens up any molecules stored in the file 4fun.pdb and returns a list of references to opened models. (Put 4fun.pdb on your desktop or change the path in the command below.)

```
opened = chimera.openModels.open('~/Desktop/4fun.pdb')
```

Because only one molecule was opened, opened is a list with just one element. Get a reference to that element (which is a `Molecule` instance) and store it in `mol`

```
mol = opened[0]
```

Now that we have a molecule to work with, an excellent way of examining its data structures is to flatten it

out and write it to a file. We'll write this file in the mol2 format, a free-format ascii file that describes molecular structure. It is not necessary to have any prior knowledge of the mol2 format to understand this example, just a basic comprehension of file formats that use coordinate data. Check out the finished product. It should serve as a good reference while you're going through the example. Get a reference to a file to write to:

```
f = open("4fun.mol2", 'w')
```

mol2 uses a series of Record Type Indicators (RTI), that indicate the type of structure that will be described in the following lines. An RTI is simply an ASCII string which starts with an asterisk (@), followed by a string of characters, and is terminated by a new line. Here, we define some RTI's that we will use througout the file to describe the various parts of our model:

```
MOLECULE_HEADER = "@MOLECULE"
ATOM_HEADER = "@ATOM"
BOND_HEADER = "@BOND"
SUBSTR_HEADER = "@SUBSTRUCTURE"
```

The chimera2sybyl dictionary is used to map Chimera atom types to Sybyl atom types. See section below on writing out per-atom information.

```
chimera2sybyl = {

      'C3' : 'C.3', 'C2' : 'C.2', 'Car' : 'C.ar', 'Cac' : 'C.2',
      'C1' : 'C.1', 'N3+' : 'N.4', 'N3' : 'N.3', 'N2' : 'N.2',
      'Npl' : 'N.pl3', 'Ng+' : 'N.pl3', 'Ntr' : 'N.2', 'Nox' : 'N.4',
      'N1+' : 'N.1', 'N1' : 'N.1', 'O3' : 'O.3', 'O2' : 'O.2',
      'Oar' : 'O.2', 'O3-' : 'O.co2', 'O2-' : 'O.co2', 'S3+' : 'S.3',
      'S3' : 'S.3', 'S2' : 'S.2', 'Sac' : 'S.O2', 'Son' : 'S.O2',
      'Sxd' : 'S.O', 'Pac' : 'P.3', 'Pox' : 'P.3', 'P3+' : 'P.3',
      'HC' : 'H', 'H' : 'H', 'DC' : 'H', 'D' : 'H',
      'P' : 'P.3', 'S' : 'S.3', 'Sar' : 'S.2', 'N2+' : 'N.2'

}
```

Writing Out per-Molecule Information

The "<TRIPOS>MOLECULE" RTI indicates that the next couple of lines will contain information relevant to the molecule as a whole. First, write out the Record Type Indicator (RTI):

```
f.write("%s\n" % MOLECULE_HEADER)
```

The next line contains the name of the molecule. This can be accessed through the mol.name attribute. (Remember, mol is a reference to the molecule we opened). If the model you open came from a pdb file, name will most often be the name of the file (without the .pdb extension). For this example, mol.name is "4fun".

```
f.write("%s\n" % mol.name)
```

Next, we need to write out the number of atoms, number of bonds, and number of substructures in the model (substructures can be several different things; for the sake of simplicity, the only substructures we'll worry about here are residues). This data is accessible through attributes of a molecule object: mol.atoms, mol.bonds, and mol.residues all contain lists of their respective components. We can determine how

many atoms, bonds, or residues this molecule has by taking the `len` of the appropriate list. save the list of references to all the atoms in `mol`:

```
ATOM_LIST = mol.atoms
```

save the list of references to all the bonds in `mol`:

```
BOND_LIST = mol.bonds
```

save the list of references to all the residues in `mol`:

```
RES_LIST = mol.residues
```

```
f.write("%d %d %d\n" % ( len(ATOM_LIST), len(BOND_LIST), len(RES_LIST)) )
```

type of molecule

```
f.write("PROTEIN\n")
```

indicate that no charge-related information is available

```
f.write("NO_CHARGES\n")
```

```
f.write("\n\n")
```

Writing Out per-Atom Information

Next, write out atom-related information. In order to indicate this, we must first write out the atom RTI:

```
f.write("%s\n" % ATOM_HEADER)
```

Each line under the `ATOM` RTI consists of information pertaining to a single atom. The following information about each atom is required: an arbitrary atom id number, atom name, x coordinate, y coordinate, z coordinate, atom type, id of the substructure to which the atom belongs , name of the substructure to which the atom belongs.

You can look at each atom in the molecule by looping through its `atoms` attribute. Remember, `ATOM_LIST` is the list of atoms stored in `mol.atoms`. It's more efficient to get the list once, and assign it to a variable, then to repeatedly ask for `mol.atoms`.

```
for atom in ATOM_LIST:
```

> Now that we have a reference to an atom, we can write out all the necessary information to the file. The first field is an arbitrary id number. We'll just use that atom's index within the `mol.atoms` list.
>
> ```
> f.write("%d " % ATOM_LIST.index(atom) )
> ```
>
> Next, we need the name of the atom, which is accessible via the `name` attribute.
>
> ```
> f.write("%s " % atom.name)
> ```
>
> Now for the x, y, and z coordinate data. Get the atom's `xformCoord` object. This is essentially a wrapper that holds information about the coordinate position (x,y,z) of that atom. `xformCoord.x`, `xformCoord.y`, and `xformCoord.z` store the x, y, and z coordinates, respectively, as floating point integers. This information comes from the coordinates given for each atom specification in the input file

```
coord = atom.xformCoord()
f.write("%g %g %g " % (coord.x, coord.y, coord.z) )
```

The next field in this atom entry is the atom type. This is a string which stores information about the chemical properties of the atom. It is accessible through the `idatmType` attribute of an atom object. Because Chimera uses slightly different atom types than SYBYL (the modeling program for which .mol2 is the primary input format), use a dictionary called chimera2sybyl (defined above) that converts Chimera's atom types to the corresponding SYBYL version of the atom's type.

```
f.write("%s " % chimera2sybyl[atom.idatmType])
```

The last two fields in an atom entry pertain to any substructures to which the atom may belong. As previously noted, we are only interested in residues for this example. Every atom object has a `residue` attribute, which is a reference to the residue to which that atom belongs.

```
res = atom.residue
```

Here, we'll use `res.id` for the substructure id field. `res.id` is a string which represents a unique id for that residue (a string representation of a number, i.e. "1" , which are sequential, for all the residues in a molecule).

```
f.write("%s " % res.id)
```

The last field to write is substructure name. Here, we'll use the `type` attribute of `res`. the `type` attribute contains a string representation of the residue type (e.g. "HIS", "PHE", "SER"...). Concatenate onto this the residue's `id` to make a unique name for this substructure (because it is possible, and probable, to have more than one "HIS" residue in a molecule. This way, the substructure name will be "HIS6" or "HIS28")

```
f.write("%s%s\n" % (res.type, res.id) )
```

```
f.write("\n\n")
```

Writing Out per-Bond Information

Now for the bonds. The bond RTI says that the following lines will contain information about bonds.

```
f.write("%s\n" % BOND_HEADER)
```

Each line after the bond RTI contains information about one bond in the molecule. As noted earlier, you can access all the bonds in a molecule through the `bonds` attribute, which contains a list of bonds.

```
for bond in BOND_LIST:
```

each bond object has an `atoms` attribute, which is list of length 2, where each item in the list is a reference to one of the atoms to which the bond connects.

```
a1, a2 = bond.atoms
```

The first field in a mol2 bond entry is an arbitrary bond id. Once again, we'll just use that bond's index in the `mol.bonds` list

```
f.write("%d " % BOND_LIST.index(bond) )
```

The next two fields are the ids of the atoms which the bond connects. Since we have a reference to both these atoms (stored in a1 and a2), we can just get the index of those objects in the mol.atoms list:

```
f.write("%s %s " % (ATOM_LIST.index(a1), ATOM_LIST.index(a2)) )
```

The last field in this bond entry is the bond order. Chimera doesn't currently calcuate bond orders, but for our educational purposes here, this won't be a problem. The mol2 format expects bond order as a string: "1" (first-order), "2" (second-order), etc., so just write out "1" here (even though this may not be correct).

```
f.write("1\n")
```

```
f.write("\n\n")
```

Writing Out per-Residue Information

Almost done!!! The last section contains information about the substructures (i.e. residues for this example) You know the drill:

```
f.write("%s\n" % SUBSTR_HEADER)
```

We've already covered some of these items (see above):

```
for res in RES_LIST:
```

residue id field

```
f.write("%s " % res.id )
```

residue name field

```
f.write("%s%s " % (res.type, res.id) )
```

the next field specifies the id of the root atom of the substructure. For the case of residues, we'll use the alpha-carbon as the root. Each residue has an atomsMap attribute which is a dictionary. The keys in this dictionary are atom names (e.g. C, N, CA), and the values are lists of references to atoms in the residue that have that name. So, to get the alpha-carbon of this residue:

```
alpha_carbon = res.atomsMap['CA'][0]
```

and get the id of alpha_carbon from the mol.atoms list

```
f.write("%d " % ATOM_LIST.index(alpha_carbon) )
```

The final field of this substructure entry is a string which specifies what type of substructure it is:

```
f.write("RESIDUE\n")
```

```
f.write("\n\n")
f.close()
```

And that's it! Don't worry if you didn't quite understand all the ins and outs of the mol2 file format. The

purpose of this exercise was to familiarize yourself with Chimera's object model; writing out a mol2 file was just a convenient way to do that. The important thing was to gain an understanding of how Chimera's atoms, bonds, residues, and molecules all fit together.

Display Properties

The goal of any molecular modeling system is to enable researchers to visualize their data. Equally important as the attributes that describe chemical structure, are those that control how the structures are actually represented on-screen. In fact, an extensive object model is worthless unless the objects can be represented in a suitable manner! The display of Chimera's core objects is governed by a few key concepts:

Color Hierarchy

Chimera uses a hierarchical system to color fundamental chemical components. This hierarchy is composed of two levels: 1) individual atoms/bonds/residues and 2) the model as a whole. The `color` assigned to an individual atom/bond/residue will be visible over the `color` assigned to the model as a whole. When a model is initially opened, each atom/bond/residue `color` is set to None, and the model-level `color` is determined by a configurable preference (by default, Chimera automatically assigns a unique model-level `color` to each new molecule that is opened). Because all the components' (atoms/bonds/residues) `color` attributes are initially set to `None`, they (visually) inherit their color from the model-level `color`. However, setting any particular atom's `color`, or issuing a command such as 'color blue' (which is the same as setting each individual atom's `color` to blue) will result in the model appearing blue (because either of those actions affect an individual atoms' `color`, which takes visual precedence over the model-level `color`). See here for more information.

Display Hierarchy

Each of Chimera's objects has an attribute which determines if it is displayed (visible) or not. For atoms, bonds, and molecules this is called `display`, while residues have a `ribbonDisplay` attribute (residues are represented visually as ribbons). A value of `True` means that the component is displayed, while `False` means it is not displayed. An atom/bond/residue will only be displayed if the model to which it belongs is displayed. This means that even if an atom/bond/residue's respective display attribute is set to `True`, if the molecule to which that atom belongs is undisplayed (i.e. the molecule's `display` is set to `False`), then that atom/bond/residue will still not be visible. See here for more information.

Draw Modes

Each Chimera object can be drawn in one of several representations ('draw modes'), specific to that object. atoms and bonds each have an attribute named `drawMode` that controls this characteristic, while residues' (because they are represented as ribbons) corresponding attribute is called `ribbonDrawMode`. The value of this attribute is a constant which corresponds to a certain type of representation specific to that object. For example, `chimera.Atom.Dot`, `chimera.Atom.Sphere`, `chimera.Atom.EndCap` and `chimera.Atom.Ball` are constants that each define a different draw mode for atoms. There is a different set of constants that define draw modes for bonds and residues (see below for more information).

Example *displayProp.py*

```
import chimera
```

open up a molecule to work with:

```
opened = chimera.openModels.open('3fx2', type="PDB")
mol = opened[0]
```

Molecule Display Properties

the `color` attribute represents the model-level color. This color can be controlled by the midas command `modelcolor`. The `color` assigned to a newly opened model is determined by a configurable preference (see discussion above). Programmatically, the model color can be changed by simply assigning a `MaterialColor` to `molecule.color`. Molecules also have a `display` attribute, where a value of `True` corresponds to being displayed, and a value of `False` means the molecule is not displayed. So to make sure the molecule is shown (it is by default when first opened):

```
mol.display = True
```

To color the molecule red, get a reference to Chimera's notion of the color red (returns a `MaterialColor` object)

```
from chimera.colorTable import getColorByName
red = getColorByName('red')
```

and assign it to `mol.color`.

```
mol.color = red
```

Note that the model will appear red at this point because all the atoms/bonds/residues `color` attributes are set to `None`

Atom Display Properties

Each atom in a molecule has its own individual color, accessible by the `color` attribute. Upon opening a molecule, each atom's `color` is set to `None`; it can be changed by assigning a new `MaterialColor` to `atom.color`. So, if we wanted to color all the alpha-carbon atoms blue, and all the rest yellow, get references to the colors:

```
blue = getColorByName('blue')
yellow = getColorByName('yellow')
```

get a list of all the atoms in the molecule

```
ATOMS = mol.atoms
for at in ATOMS:
```

check to see if this atom is an alpha-carbon

```
    if at.name == 'CA':
```

```
        at.color = yellow
```

```
    else:
        at.color = blue
```

Now, even though `mol.color` is set to red, the molecule will appear to be blue and yellow. This is because each individual atom's `color` is visible over `mol.color`.

Like molecules, atoms also have a `display` attribute that controls whether or not the atom is shown. While `atom.display` controls whether the atom can be seen at all, `atom.drawMode` controls its visual representation. The value of `drawMode` can be one of four constants, defined in the `Atom` class. Acceptable values for `drawMode` are `chimera.Atom.Dot` (dot representation), `chimera.Atom.Sphere` (sphere representation), `chimera.Atom.EndCap` (endcap representation), or `chimera.Atom.Ball` (ball representation). So, to represent all the atoms in the molecule as "balls":

```
for at in ATOMS:

    at.drawMode = chimera.Atom.Ball
```

Bond Display Properties

Bonds also contain `color`, and `drawMode` attributes. They serve the same purposes here as they do in atoms (`color` is the color specific to that bond, and `drawMode` dictates how the bond is represented). `drawMode` for bonds can be either `chimera.Bond.Wire` (wire representation) or `chimera.Bond.Stick` (stick representation). The `bond.display` attribute accepts slightly different values than that of other objects. While other objects' `display` can be set to either `False` (not displayed) or `True` (displayed), `bond.display` can be assigned a value of `chimera.Bond.Never` (same as `False` - bond is not displayed), `chimera.Bond.Always` (same as `True` - bond is displayed), or `chimera.Bond.Smart` which means that the bond will only be displayed if both the atoms it connects to are displayed. If not, the bond will not be displayed. The heuristic that determines bond color is also a little more complicated than for atoms. Bonds have an attribute called `halfbond` that determines the source of the bond's color. If `halfbond` is set to `True`, then the bond derives its color from the atoms which it connects, and ignores whatever `bond.color` is. If both those atoms are the same color (blue, for instance), then the bond will appear blue. If the bonds atoms are different colors, then each half of the bond will correspond to the color of the atom on that side. However, if `bond.halfbond` is set to `False`, then that bond's color will be be derived from its `color` attribute, regardless of the `colors` of the atoms which it connects (except in the case `bond.color` is `None`, the bond will derive its color from one of the atoms to which it connects). To set each bond's display mode to "smart", represent it as a stick, and turn halfbond mode on, get a list of all bonds in the molecule

```
BONDS = mol.bonds
for b in BONDS:

    b.display = chimera.Bond.Smart
    b.drawMode = chimera.Bond.Stick
    b.halfbond = True
```

Residue Display Properties

Residues are not "displayed" in the same manner that atoms and bonds are. When residues are displayed, they are in the form of ribbons, and the attributes that control the visual details of the residues are named accordingly: `ribbonDisplay`, `ribbonColor`, `ribbonDrawMode`. The values for `ribbonDrawMode` can be `chimera.Residue.Ribbon_2D` (flat ribbon), `chimera.Residue.Ribbon_Edged` (sharp ribbon), or `chimera.Residue.Ribbon_Round` (round/smooth ribbon). If a residue's `ribbonDisplay` value is set to `False`, it doesn't matter what `ribbonDrawMode` is - the ribbon still won't be displayed! Residues have

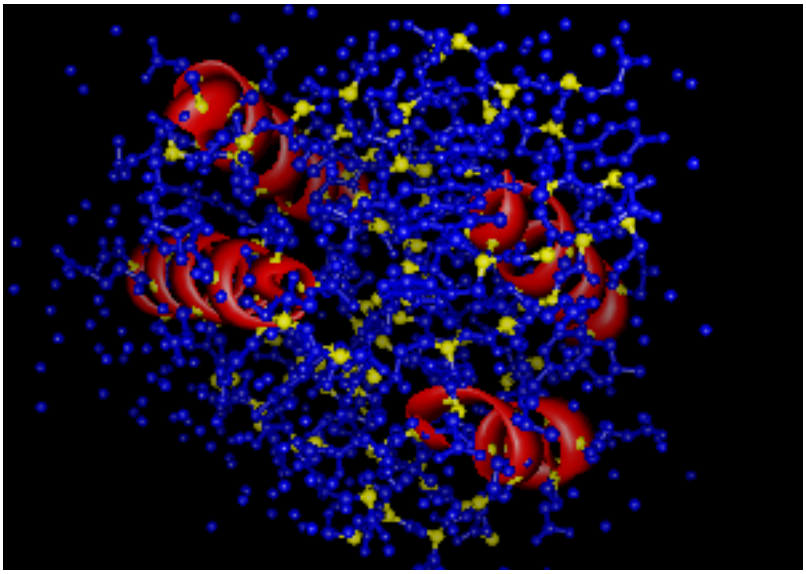
three attributes that control how the ribbon is drawn. `isTurn`, `isHelix`, and `isSheet` (same as `isStrand`) are set to either `True` or `False` based on secondary structure information contained in the source file (if available). For any residue, only one of these can be set to `True`. So, to display only the residues which are part of an alpha-helix, as a smooth ribbon, get a list of all the residues in the molecule

```
RESIDUES = mol.residues
for r in RESIDUES:

    only for residues that are part of an alpha-helix
    if r.isHelix:

        r.ribbonDisplay = True
        r.ribbonDrawMode = chimera.Residue.Ribbon_Round
```

This leaves us with a very colorful (if a little scientifically useless) model!!



Molecular Editing Using Python

Nearly all data in a Chimera session may be accessed using the Python interface. In particular, molecular data is arranged as instances of ***Atom***, ***Bond*** , ***Residue*** and ***Molecule*** classes in the ***chimera*** module. Instance attributes may be modified and the changes are automatically reflected in the main graphical window.

The code below illustrates how to show protein backbone while hiding all other atoms and bonds. The graphical window renders atoms (and associated bonds) whose `display` attribute is set to true. Thus, all that is needed to show or hide atoms (and bonds) is to set the `display` attribute to true or false, respectively.

Example *MolecularEditing.py*

Import system modules used in this example.

```
import re
```

Import Chimera modules used in this example.

```
import chimera
```

Define a regular expression for matching the names of protein backbone atoms (we do not include the carbonyl oxygens because they tend to clutter up the graphics display without adding much information).

```
MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
```

Do the actual work of setting the display status of atoms and bonds. The following `for` statement iterates over molecules. The function call `chimera.openModels.list(modelTypes=[chimera.Molecule])` returns a list of all open molecules; non-molecular models such as surfaces and graphics objects will not appear in the list. The loop variable `m` refers to each model successively.

```
for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
```

> The following `for` statement iterates over atoms. The attribute reference `m.atoms` returns a list of all atoms in model `m`, in no particular order. The loop variable `a` refers to each atom successively.
>
> ```
> for a in m.atoms:
> ```
>
> > Set the display status of atom `a`. First, we match the atom name, `a.name`, against the backbone atom name regular expression, `MAINCHAIN`. The function call `MAINCHAIN.match(a.name)` returns an `re.Match` object if the atom name matches the regular expression or `None` otherwise. The display status of the atom is set to true if there is a match (return value is not `None`) and false otherwise.
> >
> > ```
> > a.display = MAINCHAIN.match(a.name) != None
> > ```
>
> By default, bonds are displayed if and only if both endpoint atoms are displayed, so therefore we don't have to explicitly set bond display modes; they will automatically "work right".

Code Notes

The code indiscriminately hides atoms whose names do not match protein backbone atom names, so any non-protein molecules will be completely hidden.

Running the Example

You can execute the example code by downloading the linked Python script and opening it with the File→Open menu item or with the open command. Note that the ***.py*** extension is required for the open dialog/command to recognize that the file is a Python script.

You could also execute the example code by typing it in, line by line, into the main window of the Python **I**nteractive **D**eve**L**opment **E**nvironment extension (IDLE, for short). To display the IDLE window, activate the `Tools` menu and roll over the `General Controls` submenu to select `IDLE`. Alternatively, the example code may be saved in a disk file, ***e.g. ~/Desktop/backbone.py*** (***.py*** suffix still required) and executed from Chimera's Python command line by typing:

```
execfile("~/Desktop/backbone.py")
```

Note that the code in ***backbone.py*** could also have been executed via the `import` statement (***e.g.*** `import` backbone), but only if the directory containing ***backbone.py*** is on your Python path. Also, since modules are only imported once, the code could not have been executed again if desired. Using `execfile` allows multiple executions.

Toolbar Buttons

The toolbar is a user interface component that appears in the main Chimera window, typically on the left-hand side. The purpose of the toolbar is to hold buttons that invoke commonly used functionality (***e.g.*** displaying protein backbone) and sets of buttons that comprise the entire interface to an extension (***e.g.*** playing molecular dynamics movies).

There are four items associated with the button: an icon,, a Python function, a short description, and an URL to a full description. The icon is displayed in the button, and determines the size of the button. The Python function is called when the button is pressed. The description appears as balloon help text. The full description is displayed when context help is invoked.

The code below illustrates how to a button on the toolbar. The code ***must*** be executed from a file (***i.e.***, it cannot be typed in via the Python command line in the main Chimera window). The icon associated with the button is an image file named ***ToolbarButton.tiff*** in the same directory as the Python source code. The short description is ``Show Main Chain''. The Python function displays protein backbone and hides all other atoms and bonds, and the code in the body of the function is explained in greater detail in Molecular Editing Using Python.

Example *ToolbarButton.py*

Function `mainchain` sets the display status of atoms and requires no arguments. The body of the function is identical to the example described in Molecular Editing Using Python.

```
def mainchain():
```

Note that due to a fairly arcane Python behavior, we need to import modules used by a (script) function inside the function itself (the local scope) rather than outside the function (the global scope). This is because Chimera executes scripts in a temporary module so that names defined by the script don't conflict with those in Chimera's main namespace. When the temporary module is deleted, Python sets all names in the module's global namespace to `None`. Therefore, by the time this function is executed (by the toolbar button callback) any modules imported in the global namespace would have the value `None` instead of being a module object.

The regular expression module, `re`, is used for matching atom names.

```
import re
```

Import the object that tracks open models and the Molecule class from the `chimera` module.

```
from chimera import openModels, Molecule

mainChain = re.compile("^(N|CA|C)$", re.I)
for m in openModels.list(modelTypes=[Molecule]):

      for a in m.atoms:
            a.display = mainChain.match(a.name) != None
```

Need to import the `chimera` module to access the function to add the icon to the toolbar.

```
import chimera
```

Create a button in the toolbar. The first argument to `chimera.tkgui.app.toolbar.add` is the icon, which is either the path to an image file, or the name of a standard Chimera icon (which is the base name of an image file found in the "share/chimera/images" directory in the Chimera installation directory). In this case, since `ToolbarButton.tiff` is not an absolute path, the icon will be looked for under that name in both the current directory and in the Chimera images directory. The second argument is the Python function to be called when the button is pressed (a.k.a., the "callback function"). The third argument is a short description of what the button does (used typically as balloon help). The fourth argument is the URL to a full description. For this example the icon name is `ToolbarButton.tiff`; the Python function is `mainchain`; the short description is "Show Main Chain"; and there is no URL for context help.

```
chimera.tkgui.app.toolbar.add('ToolbarButton.tiff', mainchain, 'Show Main Chain', None)
```

Code Notes

The code in this example consists of two portions: defining the actual functionality in function mainchain and presenting an user interface to the functionality. While the example is presented as a single Python source file, there are good reasons for dividing the code into multiple source files and using a Python package instead. The advantages of the latter approach is illustrated in Packaging an Extension.

Running the Example

You can execute the example code by downloading the linked Python script and opening it with the File→Open menu item or with the open command. Note that the ***.py*** extension is required for the open dialog/command to recognize that the file is a Python script. The icon tiff file must be saved to a file named 'ToolbarButton.tiff' in the same directory as the script.

Packaging an Extension

Chimera extensions typically can be divided into two parts: data manipulation and user interface. For example, the code in Toolbar Buttons defines a function which changes the display status of some atoms (the data manipulation code) and then creates a toolbar button that invokes the function when pressed (the user interface code). The data manipulation code often may be reused when building a new extension, but the user interface code typically is not needed. Separating the parts into multiple source files simplifies reusing the data manipulation code, but complicates managing the extension code as a unit. Fortunately, Python supports the ***package*** concept for just such a situation.

A Python package consists of a set of modules (***.py***files) stored in the same directory in the file system. One of the modules must be named ***__init__.py*** , which is the initialization module that is automatically executed when the package is imported. By convention, Chimera extension packages implement the data manipulation code in ***__init__.py*** and the user interface code in a module named ***gui.py*** Implementors of new functionality can access the data manipulation code by:

```
import extension
```

and end users can display the user interface by:

```
import extension.gui
```

where *extension* is the name of the package. The code in Toolbar Buttons is divided in such a manner below:

Example *ToolbarButtonPackage/__init__.py*

The contents of ToolbarButtonPackage/__init__.py is identical to the first section of code in Toolbar Buttons.

```
def mainchain():

	import re
	from chimera import openModels, Molecule

	mainChain = re.compile("^(N|CA|C)$", re.I)
	for m in openModels.list(modelTypes=[Molecule]):
		for a in m.atoms:
			a.display = mainChain.match(a.name) != None
```

Example *ToolbarButtonPackage/gui.py*

The contents of ToolbarButtonPackage/gui.py is similar to the last section of code in Toolbar Buttons, with the exception that the `mainchain` function is now referenced as `ToolbarButtonPackage.mainchain`. The reason for the change is that `gui.py` is a submodule, while the `mainchain` function is in the main

package. Since a submodule cannot directly access items defined in the main package, `gui.py` must first import the package `import ToolbarButtonPackage` and refer to the function by prepending the package name (`ToolbarButtonPackage.mainchain` in the call to `chimera.tkgui.app.toolbar.add`).

```
import chimera
import ToolbarButtonPackage
chimera.tkgui.app.toolbar.add('ToolbarButton.tiff', ToolbarButtonPackage.mainchain,
'Show Main Chain', None)
```

Running the Example

The example code files must be saved in a directory named *`ToolbarButtonPackage`*. To run the example, start ***chimera*** , bring up the Tools preference category (via the `Preferences` entry in the `Favorites` menu; then choosing the "Tools" preference category) and use the `Add` button to add the directory above the *`ToolbarButtonPackage`* directory. Then type the following command into the IDLE command line:

```
import ToolbarButtonPackage.gui
```

The effect should be identical to running the Toolbar Buttons example.

Working with the Chimera Extension Manager

Chimera extensions typically can be divided into two parts: data manipulation and user interface. For example, the code in Toolbar Buttons defines a function which changes the display status of some atoms. This is the data manipulation part of that extension. The code also creates a toolbar button that invokes the function when pressed. This is the user interface part of the extension.

Data manipulation code may often be reused when building a new extension, but user interface code typically is not. Separating the parts into multiple source files simplifies reusing the data manipulation code, but complicates managing the extension as a unit. Fortunately, Python supports the ***package*** concept for just this purpose.

A Python package consists of a set of modules (***.py***files) stored in the same directory in the file system. One of the modules must be named ***__init__.py*** , which is the initialization module that is automatically executed when the package is imported. By convention, Chimera extension packages implement the data manipulation code in ***__init__.py*** . Implementors of new functionality can access the data manipulation code by:

```
import extension
```

where *extension* is the name of the package.

The package integrates its functionality into the Chimera extension manager by including a special module named ***ChimeraExtension.py*** in the package, and following a particular protocol within that module. Namely, for each separate function the package wants to offer through the extension manager, a class derived from `chimera.extension.EMO` (Extension Management Object) must be defined in the module and an instance registered with the extension manager.

The code in Toolbar Buttons is organized in such a manner below:

Example *ToolbarButtonExtension/__init__.py*

The contents of ToolbarButtonExtension/__init__.py is identical to the first section of code in Toolbar Buttons, with the exception that module os is not imported.

```
import re

import chimera

def mainchain():

	MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
	for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
		for a in m.atoms:
			a.display = MAINCHAIN.match(a.name) != None
```

Example *ToolbarButtonExtension/ChimeraExtension.py*

ChimeraExtension.py derives a class from `chimera.extension.EMO` to define how functionality defined in __init__.py may be invoked by the Chimera extension manager.

```
import chimera.extension
```

Class `MainChainEMO` is the derived class.

```
class MainChainEMO(chimera.extension.EMO):
```

Return the actual name of the extension.

```
    def name(self):

        return 'Backbone'
```

Return the short description that typically appears as balloon help or in the Tools preference category.

```
    def description(self):

        return 'display only protein backbone'
```

Return the categories in which this extension should appear. It is either a list or a dictionary. If it is a dictionary then the keys are the category names and the values are category-specific descriptions (and the description() method above is ignored).

```
    def categories(self):

        return ['Utilities']
```

Return the name of a file containing an icon that may be used on the tool bar to provide a shortcut for launching the extension.

```
    def icon(self):

        return self.path('mainchain.tiff')
```

Invoke the extension. Note that when this method is called, the content of "__init__.py" is not available. To simplify calling functions, the `EMO` provides a `module` method that locates modules in the extension package by name; if no name is supplied, the "__init__.py" module is returned.

```
    def activate(self):

        Call the mainchain function in the "__init__.py" module.

        self.module().mainchain()
```

Register an instance of `MainChainEMO` with the Chimera extension manager.

```
chimera.extension.manager.registerExtension(MainChainEMO(__file__))
```

Running the Example

The example code files and ToolbarButton.tiff must be saved in a directory named *ToolbarButtonExtension*. To run the example, start ***chimera*** , bring up the Tools preference category (via the Preferences entry in the Favorites menu; then choosing the "Tools" preference category), use the Add button to add the directory above the *ToolbarButtonExtension* directory. A MainChain entry should appear under the Utilities tools category.

Adding New Typed Commands

Just as menu commands are added in an extension's ***ChimeraExtension.py*** file, so are typed commands. This is for two reasons:

1. The command becomes available early, before command scripts might run.
2. The module itself doesn't have to be imported (which would slow startup).

The addCommand Function

Typed commands are added using ***Midas.midas_text*** 's addCommand function. addCommand has two mandatory arguments:

1. A string containing the typed command name. The user will be able to shorten the name to the shortest unique prefix.
2. The function to call when the command is invoked. Explained further in the **Callback Function** section below.

and it has three keyword arguments:

revFunc
: Specifies a function to call when ~***command*** is typed. If omitted, ~***command*** will raise an error.

help
: Specifies where help for the command is found. Commands whose help is provided with the Chimera documentation itself will set ***help*** to True. If ***help*** is a string, it is interpreted as an URL that will be brought up in a browser to display help. If ***help*** is a tuple, it should be a (***path*** , ***package***) 2-tuple, where ***path*** specifies a file relative to ***package*** 's ***helpdir*** subdirectory. The file will be displayed in a browser as help. Note that though ***package*** can be an actual imported package, importing the package would defeat the purpose of avoiding importing the module early, so ***package*** can just be a string specifying the module name instead. If the ***help*** keyword is omitted, no help will be provided.

changesDisplay
: A boolean that specifies if the command changes the display in the main graphics window (default: True). This is important for script processing so that Chimera knows if the display needs to update once a command has executing (and to avoid spurious extra frames during script execution -- important if recording animations).

Callback Function

The callback function you register with addCommand will be invoked with two arguments:

1. A string containing the name of the command as registered with addCommand.
2. The arguments to the command as typed by the user (a string).

The parsing of the typed arguments and calling of the function that actually performs the command work is typically handled through ***Midas.midas_text*** 's doExtensionFunc function. This is discussed in detail

below in **The doExtensionFunc Function**. Before getting into that, we know enough at this point that we can look at a brief example.

An Example

Here is the code from the ***ChimeraExtension.py*** file of the ***Define Attribute*** extension that implements adding the `defattr` command to the command line module:

```
def cmdAddAttr(cmdName, args):
        from AddAttr import addAttributes
        from Midas.midas_text import doExtensionFunc
        doExtensionFunc(addAttributes, args,
                                specInfo=[("spec", "models", "molecules")])

from Midas.midas_text import addCommand
addCommand("defattr", cmdAddAttr, help=True)
```

First, a callback function named `cmdAddAttr` is defined that will later be registered with `addCommand`. The callback imports a "workhorse" function (`addAttributes`) from the main module and `doExtensionFunc` from ***Midas.midas_text*** and then calls `doExtensionFunc` to process the typed arguments and call `addAttributes` appropriately. Note that `addAttributes` is imported ***inside*** the `cmdAddAttr` definition. If it were outside, then the whole module would be imported during Chimera startup, which we are trying to avoid.

After the `cmdAddAttr` function is defined, ***Midas.midas_text*** 's `addCommand` is called to add the `defattr` command to the command interpreter. Since the help for the `defattr` command is shipped with Chimera, the ***help*** keyword argument is set to `True`.

The doExtensionFunc Function

As seen in the **An Example** section above, `doExtensionFunc` has two mandatory arguments: the "workhorse" function that actually carries out the operation requested by the user, and a string containing the command arguments that the user typed. `doExtensionFunc` introspects the workhorse function to determine how many mandatory arguments it expects and what keyword arguments it accepts. The initial arguments in the typed string are assumed to correspond to the mandatory arguments, and the remainder of the typed string is assumed to specify valid keyword/value pairs (space separated rather than "=" separated). Keywords will be matched regardless of case, and the user need only type enough characters to distinguish keywords.

`doExtensionFunc` has two keyword arguments:

invalid
: A list of keyword arguments that cannot be used from the command line. `doExtensionFunc` will behave as if the workhorse function did not have these keywords.

specInfo
: If the workhorse function has argument(s) whose value should be a list of `Atoms`, `Residues`, ***etc.*** , for which the user needs to type an atom specifier, that information is given here. ***specInfo*** is a list of 3-tuples. The first component of the 3-tuple is the keyword the user should type or, if this is a positional argument, the name that the argument should be assumed to have for type-

guessing purposes (in either case it needs to end in "spec"). The next component is the real argument name that the function uses (it will automatically be added to ***invalid***). The final component is the method name to apply to the selection generated by the atom spec in order to extract the desired list (typically "atoms", "residues", "molecules", or "models"). If the method name is None, then the selection itself will be returned.

Typed arguments are processed using some heuristic rules to convert them to their most "natural" type. However, the argument name used by the workhorse function can influence how the typed argument is processed. In particular, if the argument name (ignoring case) ends in...

color
: The typed argument is treated as a color name and is converted to a `MaterialColor`.

spec
: The typed argument is assumed to be an atom specifier and is converted to a `Selection`.

file
: The typed argument is a file name. If the user types "browse" or "browser" then a file selection dialog is displayed for choosing the file. If the workhorse argument name ends in ***savefile*** , then a save-style browser will be used.

Furthermore, if the user provides a keyword argument multiple times, the value provided to the workhorse function will be a list of the individual values.

In some cases it may be desirable to provide a "shim" function between the `doExtensionFunc` "workhorse" function and the module's true workhorse function in order to provide more user-friendly argument names or default values than those of the normal module API.

MidasError

If you want to have errors from your command-line function handled the same was as other command-line errors (***i.e.*** shown as red text in the status line rather than raising an error dialog), then you need to have the function you register with `addCommand` raise ***MidasError*** in those cases instead of other error types. This may involve embedding your use of `doExtensionFunc` in a `try/except` block and re-raising caught errors as ***MidasError*** . ***MidasError*** is defined in the ***Midas*** module.

A Second Example

Example *ToolbarButtonCommand/ChimeraExtension.py*

The initial code is the same as for the ToolbarButtonExtension example

```
import chimera.extension

class MainChainEMO(chimera.extension.EMO):

      def name(self):
            return 'Backbone'

      def description(self):
            return 'display only protein backbone'
```

```
    def categories(self):
        return ['Utilities']

    def icon(self):
        return self.path('mainchain.tiff')

    def activate(self):
        self.module().mainchain()


chimera.extension.manager.registerExtension(MainChainEMO(__file__))
```

Here we define two functions, one to handle the "mainchain" command, and one to handle the "~mainchain" command.

```
def mainchainCmd(cmdName, args):
```

Import the module's workhorse function. It is imported inside the function definition so that it does not slow down Chimera startup with extra imports in the main scope.

```
    from ToolbarButtonCommand import mainchain
```

Import and use the Midas.midas_text doExtensionFunc procedure to process typed arguments and call the mainchain() function appropriately. For a simple function like mainchain(), which takes no arguments, using doExtensionFunc is probably overkill. One could instead use the approach applied in the revMainchainCmd function below and simply test for the presence of any arguments (raising MidasError if any are found) and directly calling the mainchain() function otherwise. As implemented here, using doExtensionFunc, if the user does provide arguments then doExtensionFunc will raise an error complaining that there were unknown keyword arguments supplied.

```
    from Midas.midas_text import doExtensionFunc
    doExtensionFunc(mainchain, args)
```

The function for "~mainchain"

```
def revMainchainCmd(cmdName, args):
```

We are going to implement ~mainchain as a synonym for "display", so we import runCommand which simplifies doing that.

```
    from chimera import runCommand
    from Midas import MidasError
    if args:
```

Raising MidasError will cause the error message to show up in the status line as red text

```
        raise MidasError("~mainchain takes no arguments")
```

runCommand takes any legal command-line command and executes it.

```
    runCommand("display")
```

Now actually register the "mainchain" command with the command interpreter by using addCommand(). The first argument is the command name and the second is the callback function for doing the work. The `revFunc` keyword specifies the function to implement "~mainchain". The `help` keyword has been omitted,

therefore no help will be provided.

```
from Midas.midas_text import addCommand
addCommand("mainchain", mainchainCmd, revFunc=revMainchainCmd)
```

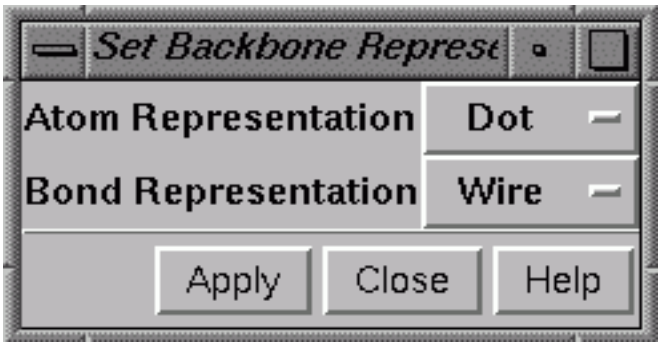
Running the Example

The example files (ChimeraExtension.py, __init__.py, and ToolbarButton.tiff) must be saved in a directory named *ToolbarButtonCommand*. To run the example, start ***chimera*** , bring up the Tools preference category (via the ***Preferences*** entry in the **Favorites** menu; then choosing the "Tools" preference category), use the ***Add*** button to add the directory above the *ToolbarButtonCommand* directory. You should then be able to type "mainchain" to the Chimera command line (start the command line from the Favorites menu if necessary).

Extension-Specific User Interface

Chimera implements its graphical user interface (GUI) using a Python interface (Tkinter module) to the Tcl/Tk toolkit. Since Chimera extensions are also written in Python, they can extend the user interface using the same mechanism. Any extension that requires user input will need to present a GUI. This example assumes that the reader is familiar with Tkinter and does not describe the Tkinter-specific code in detail.

The code below demonstrates how to change the display mode of protein backbone to a user-selected representation. The graphical window renders atoms and bonds according to their `drawMode` attribute. Thus, all that the example code in the main package, ***__init__.py*** , does is to change the attribute values of backbone atoms and bonds. The example code in the graphical user interface submodule, ***gui.py*** adds a button to the Chimera toolbar. When the user clicks the toolbar button, the window below is displayed.



The user can select the desired display representation for atoms and bonds using the option menus, and then set the backbone atom and bond representations by clicking the `Apply` button.

Example *ExtensionUI/__init__.py*

Import the standard modules used in this example.

```
import re
```

Import the Chimera modules used in this example.

```
import chimera
```

Define a regular expression for matching the names of protein backbone atoms (we do not include the carbonyl oxygens because they tend to clutter up the graphics display without adding much information).

```
MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
```

Define `mainchain` function for setting the display representation of protein backbone atoms and bonds. See Molecular Editing for a more detailed example on changing molecular attributes.

```
def mainchain(atomMode, bondMode):

      for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
            for a in m.atoms:
                  if MAINCHAIN.match(a.name):
```

```
                a.drawMode = atomMode
        for b in m.bonds:
            ends = b.atoms
            if MAINCHAIN.match(ends[0].name) \
            and MAINCHAIN.match(ends[1].name):
                b.drawMode = bondMode
```

Example *ExtensionUI/gui.py*

Import the standard modules used by this example.

```
import os
import Tkinter
```

Import the Chimera modules and classes used by this example.

```
import chimera
from chimera.baseDialog import ModelessDialog
```

Import the package for which the graphical user interface is designed. In this case, the package is named `ExtensionUI`.

```
import ExtensionUI
```

Define two module variables: `atomMode` and `bondMode` are Tk variables that keep track of the last selected display representations. These variables are initialized to be `None`, and are set to usable values when the GUI is created.

```
atomMode = None
bondMode = None
```

Define two dictionaries that map string names into Chimera enumerated constant values. The two variables `atomMode` and `bondMode` keep track of the representations as strings because they are displayed directly in the user interface. However, the `mainchain` function in the main package expects Chimera constants as its arguments. The dictionaries `atomModeMap` and `bondModeMap` provides the translation from string to enumerated constants.

```
atomModeMap = {

    'Dot': chimera.Atom.Dot,
    'Sphere': chimera.Atom.Sphere,
    'EndCap': chimera.Atom.EndCap,
    'Ball': chimera.Atom.Ball

}
bondModeMap = {

    'Wire': chimera.Bond.Wire,
    'Stick': chimera.Bond.Stick

}
```

Chimera offers two base classes to somewhat simplify the task of creating user interfaces: ModalDialog

and ModelessDialog. The former is designed for situations when information or response is required of the user immediately; the dialog stays in front of other Chimera windows until dismissed and prevents input from going to other Chimera windows. The latter dialog type is designed for "ongoing" interfaces; it allows input focus to go to other windows, and other windows can obscure it.

Here we declare a class that derives from ModelessDialog and customize it for the specific needs of this extension.

```
class MainchainDialog(ModelessDialog):
```

Chimera dialogs can either be ***named*** or ***nameless*** . Named dialogs are displayed using the `display(name)` function of the chimera.dialogs module. The ***name*** that should be used as an argument to the `display` function is given by the class variable `name`. Using a named dialog is appropriate when it might be desirable to invoke the dialog from other extensions or from Chimera itself.

A nameless dialog is intended for use only in the extension that defines the dialog. A nameless dialog is typically created and displayed by calling its constructor. Once created, a nameless dialog can be redisplayed (if it was withdrawn by clicking its `Cancel` button for example) by calling its `enter()` method.

For demonstration purposes, we use a named dialog here. A nameless dialog is used in the Color and Color Wells example.

```
name = "extension ui"
```

The buttons displayed at the bottom of the dialog are given in the class variable `buttons`. For modeless dialogs, a help button will automatically be added to the button list (the help button will be grayed out if no help information is provided). For buttons other than `Help`, clicking on them will invoke a class method of the same name.

Both dialog base classes provide appropriate methods for `Close`, so we won't provide a `Close` method in this subclass. The ModelessDialog base class also provides a stub method for `Apply`, but we will override it with our own `Apply` method later in the class definition.

```
buttons = ("Apply", "Close")
```

A help file or URL can be specified with the `help` class variable. A URL would be specified as a string (typically starting with "http://..."). A file would be specified as a 2-tuple of file name followed by a package. The file would then be looked for in the `helpdir` subdirectory of the package. A dialog of Chimera itself (rather than of an imported package) might only give a filename as the class help variable. That file would be looked for in /usr/local/chimera/share/chimera/helpdir.

```
help = ("ExtensionUI.html", ExtensionUI)
```

The title displayed in the dialog window's title bar is set via the class variable `title`.

```
title = "Set Backbone Representation"
```

Both ModelessDialog and ModalDialog, in their __init__ functions, set up the standard parts of the dialog interface (top-level window, bottom-row buttons, etc.) and then call a function named `fillInUI` so that the subclass can fill in the parts of the interface specific to the dialog. As an

argument to the function, a Tkinter Frame is provided that should be the parent to the subclass-provided interface elements.

```
def fillInUI(self, parent):
```

Declare that, in `fillInUI`, the names `atomMode` and `bondMode` refer to the global variables defined above.

```
global atomMode, bondMode
```

Create and initialize `atomMode` and `bondMode`, the two global Tk string variables that keep track of the currently selected display representation.

```
atomMode = Tkinter.StringVar(parent)
atomMode.set('Dot')
bondMode = Tkinter.StringVar(parent)
bondMode.set('Wire')
```

Create the label and option menu for selecting atom display representation. First create the label `Atom Representation` and place it on the left-hand side of the top row in the GUI window.

```
atomLabel = Tkinter.Label(parent, text='Atom Representation')
atomLabel.grid(column=0, row=0)
```

Create the menu button and the option menu that it brings up.

```
atomButton = Tkinter.Menubutton(parent, indicatoron=1,

      textvariable=atomMode, width=6,
      relief=Tkinter.RAISED, borderwidth=2)
atomButton.grid(column=1, row=0)
atomMenu = Tkinter.Menu(atomButton, tearoff=0)
```

Add radio buttons for all possible choices to the menu. The list of choices is obtained from the keys of the string-to-enumeration dictionary, and the radio button for each choice is programmed to update the `atomMode` variable when selected.

```
for mode in atomModeMap.keys():

      atomMenu.add_radiobutton(label=mode, variable=atomMode, value=mode)
```

Assigns the option menu to the menu button.

```
atomButton['menu'] = atomMenu
```

The lines below do the same thing for bond representation as the lines above do for atom representation.

```
bondLabel = Tkinter.Label(parent, text='Bond Representation')
bondLabel.grid(column=0, row=1)
bondButton = Tkinter.Menubutton(parent, indicatoron=1,

      textvariable=bondMode, width=6,
      relief=Tkinter.RAISED, borderwidth=2)
bondButton.grid(column=1, row=1)
```

```
        bondMenu = Tkinter.Menu(bondButton, tearoff=0)
        for mode in bondModeMap.keys():
              bondMenu.add_radiobutton(label=mode, variable=bondMode, value=mode)
        bondButton['menu'] = bondMenu
```

Define the method that is invoked when the `Apply` button is clicked. The function simply converts the currently selected representations from strings to enumerated constants, using the `atomModeMap` and `bondModeMap` dictionaries, and invokes the main package function `mainchain`.

```
def Apply(self):

        ExtensionUI.mainchain(atomModeMap[atomMode.get()],
              bondModeMap[bondMode.get()])
```

Now we register the above dialog with Chimera, so that it may be invoked via the `display(name)` method of the chimera.dialogs module. Here the class itself is registered, but since it is a named dialog deriving from ModalDialog/ModelessDialog, the instance will automatically reregister itself when first created. This allows the `dialogs.find()` function to return the instance instead of the class.

```
chimera.dialogs.register(MainchainDialog.name, MainchainDialog)
```

Create the Chimera toolbar button that displays the dialog when pressed. Note that since the package is not normally searched for icons, we have to prepend the path of this package to the icon's file name.

```
dir, file = os.path.split(__file__)
icon = os.path.join(dir, 'ExtensionUI.tiff')
chimera.tkgui.app.toolbar.add(icon, lambda d=chimera.dialogs.display,
n=MainchainDialog.name: d(n), 'Set Main Chain Representation', None)
```

Code Notes

The example above requires the user to first select the desired representation, then apply the selection to the protein backbone. An alternative interface style is to apply user selections immediately. The appropriate choice of style depends on the extension application. The reason for choosing the "Apply" style for this example is that the user is expected to change both atom and bond representations, and there is no need to edit and display intermediate representations.

If your extension brings up several instances of the same dialog, one per data set (***e.g.*** one Multalign Viewer dialog for each sequence alignment file), then you should register each dialog with the extension manager so that the user can raise a particular dialog instance should it get buried behind other windows. You do this by calling `chimera.extension.manager.registerInstance(self)` in either your `__init__` or `fillInUI` methods and deregister by calling `chimera.extension.manager.deregisterInstance(self)` in your `destroy` method (don't forget to call the parent class `destroy()` from your `destroy()`.

The ratio of 13 lines of functionality code to 34 lines of user interface code is fairly typical. Doing things is easy; figuring out what a user wants to do, that's hard.

For extensions based on the `ModalDialog` class, a different approach is typically used. The dialog is not registered (no call to `chimera.dialogs.register`). The function associated with the toolbar icon (the second argument to `chimera.tkgui.app.toolbar.add`) creates the modal dialog, calls the dialog's `run()` method, and uses that method's return value as appropriate (`None` is returned by a user-initiated Cancel of the dialog). When writing dialog methods, the return value is kept in the `self.returnValue`

attribute of the dialog. The dialog is destroyed when the toolbar function runs out of scope.

Running the Example

The example code files and toolbar icon must be saved in a directory named *ExtensionUI*. To run the example, start ***chimera*** , bring up the Tools preference category (via the Preferences entry in the Favorites menu; then choosing the "Tools" preference category) and use the Add button to add the directory above the *ExtensionUI* directory. Then type the following command into the IDLE command line:

```
import ExtensionUI.gui
```

This should display a button on the Chimera toolbar. Clicking the button should bring up a window similar to the one shown above.

Colors and Color Wells

The color editor and color wells used in Chimera are not Chimera-specific interface components. They handle colors as simple red/green/blue/alpha tuples, whereas Chimera color objects have additional attributes (such as shininess). Since translating between the two representations of colors would be tedious, a ColorOption class is provided that encapsulates the use of a color well so that the programmer need only deal with Chimera color objects.

The example below demonstrates how to display a color well, and how to update the color of protein backbone atoms when the color of the well changes. The code in the graphical user interface submodule, ***gui.py*** adds a button to the Chimera toolbar. When the user clicks the toolbar button, a window containing a color well is displayed. When the user changes the color in the well, the color of protein backbone atoms changes to match.

Example *ColorWellUI/__init__.py*

This code is analogous to the code found in the "__init__.py" modules in the Packaging an Extension and Extension-Specific User Interface examples. See Molecular Editing for a more detailed example on changing molecular attributes. Note that the `mainchain` function is expecting a color ***object*** as its argument (because the color is used to set an atomic attribute).

```
import chimera
import re

MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
def mainchain(color):

      for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
            for a in m.atoms:
                  if MAINCHAIN.match(a.name):
                        a.color = color
```

Example *ColorWellUI/gui.py*

Import the standard Python modules used by the example code.

```
import os
import Tkinter
```

Import the additional modules and classes needed. The ColorOption class facilitates interoperation between Chimera colors and color wells (which use rgba colors).

```
import chimera
from chimera.baseDialog import ModelessDialog
from chimera.tkoptions import ColorOption
import ColorWellUI

class ColorWellDialog(ModelessDialog):
```

ColorWellDialog is a "nameless" dialog. See the Extension-Specific User Interface example for a more detailed explanation of Chimera dialogs.

Set the title bar of the dialog to display `Set Backbone Color`.

```
title = 'Set Backbone Color'
```

```
def fillInUI(self, master):
```

Create a ColorOption instance. The ColorOption will contain a descriptive label and a color well. The arguments to the ColorOption constructor are:

- master widget
- row number to use when griding the ColorOption into the master widget. The default column is 0. The tkoptions module contains other options besides ColorOption (e.g. StringOption), which are generally intended to be put in vertical lists, and therefore a row number is specified in the constructor. In this example we are only using one option however.
- option label. This will be positioned to the left of the color well and a ":" will be appended.
- The default value for this option.
- A callback function to call when the option is set by the user (e.g. a color dragged to the well, or the well color edited in the color editor)
- An optional ballon-help message

```
coloropt = ColorOption(master, 0, 'Backbone Color', None, self.
_setBackboneColor, balloon='Protein backbone color')
```

Call `_updateBackboneColor` to make the color displayed in the color well reflect the current color of protein backbone atoms. While not strictly necessary, keeping the color in the well consistent with the color in the molecules enhances the extension usability.

```
self._updateBackboneColor(coloropt)
```

Define `_updateBackboneColor`, which is used to make the color of a well reflect the color of protein backbone atoms.

```
def _updateBackboneColor(self, coloroption):
```

Loop through all atoms in all molecules, looking for protein backbone atoms. When one is found, its color is compared against the last color seen, `theColor`. The first time this comparison is made, `theColor` does not exist yet and a NameError exception is raised; this exception is caught, and `theColor` is assigned the color of the atom. All subsequent times, the comparison between atom color and `theColor` should work as expected. If the two colors are different, the color well is set to show that multiple colors are present and execution returns to the caller. If the two colors are the same, the next atom is examined. If only one color is found among all protein backbone atoms (or zero if no molecules are present), then execution continues.

```
        for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):

            for a in m.atoms:
                if ColorWellUI.MAINCHAIN.match(a.name):
                    try:
                        if a.color != theColor:
                            coloroption.setMultiple()
                            return
                    except NameError:
                        theColor = a.color
```

Set the color of the well to match `theColor`. There are two possible cases:

1. `theColor` is not set (because there are no molecules),
2. `theColor` is `None` or a color object.

The `set` function will not result in a callback to `_setBackboneColor`.

```
        try:

            Handle case 2. Set the color well to the proper color

            coloroption.set(theColor)
        except NameError:

            Handle case 1. Set the color well to show that no color is present

            coloroption.set(None)
```

Define `_setBackboneColor`, which is invoked each time the color in the well changes. When called by the ColorOption, `_setBackboneColor` receives a single argument `coloropt`, which is the ColorOption instance.

```
    def _setBackboneColor(self, coloroption):

        Call the mainchain function in the main package, with the color object corresponding to
        the color well color as its argument (which will be None if No Color is the current selection
        in the well), to set the color of backbone atoms.

        ColorWellUI.mainchain(coloroption.get())
```

Define the module variable `dialog`, which keeps track of the dialog window containing the color well. It is initialized to `None`, and is assigned a usable value when the dialog is created.

```
dialog = None
```

Define `showColorWellUI`, which is invoked when the Chimera toolbar button is pressed.

```
def showColorWellUI():

    Declare that the name dialog refers to the global variable defined above.

    global dialog
```

Check whether the dialog has already been created. If so, the dialog window is displayed by calling it's `enter()` function, and then the rest of the function is skipped by returning.

```
if dialog is not None:

    dialog.enter()
    return
```

Otherwise, create the dialog.

```
dialog = ColorWellDialog()
```

Create the Chimera toolbar button that invokes the `showColorWellUI`

```
dir, file = os.path.split(__file__)
icon = os.path.join(dir, 'ColorWellUI.tiff')
chimera.tkgui.app.toolbar.add(icon, showColorWellUI, 'Set Main Chain Color', None)
```

Code Notes

This example registers a callback with the color well, so that any color change in the well results in the colors of protein backbone atoms being updated. An alternative style interface, similar to the one used in Extension-Specific User Interface, may be used by not registering the callback and adding an `Apply` button, which would invoke a function that fetches the color from the well and calls `_setBackboneColor`. For this example, since only one atomic attribute is being set, the immediate feedback seems more appropriate.

Note that there was no explicit mention of the color panel. Invocation of and interaction with the color panel is handled automatically by the color well.

The `_updateBackboneColor` function is used to synchronize the color displayed in the well with the color of the atoms. However, if the color of the atoms are altered through another agency (***e.g.***a different extension), then the well color and backbone color no longer match. The Trigger Notifications example shows how to keep the well color `up-to-date`.

Running the Example

The example code files must be saved in a directory named *`ColorWellUI`*. To run the example, start ***chimera*** , bring up the Tools preference category (via the `Preferences` entry in the `Favorites` menu; then choosing the "Tools" preference category) and use the `Add` button to add the directory above the *`ColorWellUI`* directory. Then type the following command into the IDLE command line:

```
import ColorWellUI.gui
```

This should display a button on the Chimera toolbar. Clicking the button should bring up a window with a color well inside. The color well may be used to manipulate the color of all protein backbone atoms.

Trigger Notifications

An extension often needs to respond to changes in data caused by other extensions. For instance, in the Colors and Color Wells example, a color well is used to control the color of protein backbone atoms; if another extension (e.g., Midas emulator) changes the color of some backbone atoms, the color in the well should change accordingly as well

Chimera provides the trigger mechanism for notifying interested parties of spontaneous changes in data. A ***trigger*** is an object which monitors changes for a set of data; an extension can register a ***handler*** to be invoked whenever the trigger detects data modification. A standard set of triggers is defined in `chimera.triggers`. In particular, there are triggers for objects of common classes; e.g., there is a trigger monitoring all `Atom` objects. Thus, tracking changes of standard objects is very straightforward. Besides standard object triggers, there are a few other triggers of general interest:

selection changed
: This trigger fires whenever the selection in the main graphics window is changed. Functions in `chimera.selection` are used to query and manipulate the selection.

chimera.APPQUIT
: This trigger fires when Chimera is quitting.

chimera.MOTION_START
: One or more models have started moving.

chimera.MOTION_STOP
: All models have stopped moving for at least a second. Useful for starting heavy-weight calculations/updates based on model positioning.

chimera.CLOSE_SESSION
: The user has selected the Close Session menu item.

Also, chimera.openModels maintains two triggers that fire when models are added to or removed from the list of open models. To register for these triggers use the `chimera.openModels.addAddHandler` and `chimera.openModels.addRemoveHandler` functions respectively. They each expect two arguments: a callback function and user data. They return a handler ID. The callback function will be invoked with two arguments: the added/removed models and the provided user data. You can deregister from these triggers with `chimera.openModels.deleteAddHandler` and `chimera.openModels.deleteRemoveHandler` respectively. These latter two functions expect the handler ID as an argument.

The example below derives from the code in Colors and Color Wells, and the code description assumes that the reader is familiar with that code. While the Colors and Color Wells code only synchronizes the color well with backbone atom colors when the graphical user interface is first created, the example below registers a handler with the `Atom` trigger and updates the color well whenever a backbone atom is changed. Note that changes in atom data may have nothing to do with colors; the `Atom` trigger invokes registered handlers whenever any change is made. However, it is computationally cheaper to recompute the well color on ***any*** change than to keep track of atom colors and only update the well color on ***color*** changes.

Example *AtomTrigger/__init__.py*

This file is identical to the ColorWellUI/__init__.py in the Colors and Color Wells example.

```
import chimera
```

```
import re

MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
def mainchain(color):

      for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
            for a in m.atoms:
                  if MAINCHAIN.match(a.name):
                        a.color = color
```

Example *AtomTrigger/gui.py*

The code here is very similar to the code in Colors and Color Wells and only differences from that code will be described.

```
import os
import Tkinter

import chimera
from chimera.baseDialog import ModelessDialog
from chimera.tkoptions import ColorOption
import ColorWellUI

class ColorWellDialog(ModelessDialog):


      title = 'Set Backbone Color'
```

Need to override `__init__` to initialize our extra state.

```
      def __init__(self, *args, **kw):
```

Whereas in the Colors and Color Wells example `coloropt` was a local variable, here the `coloropt` variable is stored in the instance because the trigger handler (which has access to the instance) needs to update the color well contained in the ColorOption. A new variable, `handlerId`, is created to keep track of whether a handler is currently registered. The handler is only created when needed. See `map` and `unmap` below. (Note that the instance variables must be set before calling the base __init__ method since the dialog may be mapped during initialization, depending on which window system is used.)

```
            self.colorOpt = None
            self.handlerId = None
```

Call the parent-class `__init__`.

```
            apply(ModelessDialog.__init__, (self,) + args, kw)

      def fillInUI(self, master):
```

Save ColorOption in instance.

```
            self.coloropt = ColorOption(master, 0, 'Backbone Color', None, self.
            _setBackboneColor, balloon='Protein backbone color')
```

```
            self._updateBackboneColor()

      def _updateBackboneColor(self):
            for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
                  for a in m.atoms:
                        if ColorWellUI.MAINCHAIN.match(a.name):
                              try:
                                    if a.color != theColor:
                                          self.coloropt.setMultiple()
                                          return
                              except NameError:
                                    theColor = a.color

            try:
                  self.coloropt.set(theColor)
            except NameError:
                  self.coloropt.set(None)

      def _setBackboneColor(self, coloroption):
            ColorWellUI.mainchain(coloroption.get())
```

Register a trigger handler to monitor changes in the backbone atom list when we're make visible. We ignore the event argument.

```
      def map(self, *ignore):
```

Synchronize with well color.

```
            self._updateBackboneColor()
```

If no handler is currently registered, register one.

```
            if self.handlerId is None:
```

Registration occurs when the `chimera.triggers` object is requested to add a handler. ***Registration requires three arguments*** :

- the name of the trigger,
- the handler function to be invoked when the trigger fires, and
- an additional argument to be passed to the handler function when it is invoked.

In this case, the trigger name is the same as the name of the class of objects being monitored, "Atom". The handler function is `_handler`, defined below. And the additional argument is empty (None) `--` it could have been the ColorOption instance (`coloropt`) but that is accessible via the instance. The return value from the registration is a unique handler identifier for the handler/argument combination. This identifier is required for deregistering the handler.

The handler function is always invoked by the trigger with three arguments :

- the name of the trigger,
- the additional argument passed in at registration time, and
- an instance with three attributes
  - created: set of created objects
  - deleted: set of deleted objects
  - modified: set of modified objects

Note that with a newly opened model, objects will just appear in both the `created` set and not in the `modified` set, even though the newly created objects will normally have various of their default attributes modified by later code sections.

```
self.handlerId = chimera.triggers.addHandler('Atom', self._handler,
None)
```

The `_handler` function is the trigger handler invoked when attributes of `Atom` instances change.

```
def _handler(self, trigger, additional, atomChanges):
```

Check through modified atoms for backbone atoms.

```
for a in atomChanges.modified:
```

If any of the changed atoms is a backbone atom, call `_updateBackboneColor` to synchronize the well color with backbone atom colors.

```
if ColorWellUI.MAINCHAIN.match(a.name):

    self._updateBackboneColor()
    return
```

unmap is called when the dialog disappears. We ignore the event argument.

```
def unmap(self, *ignore):
```

Check whether a handler is currently registered (***i.e.***, the handler identifier, `handlerId`, is not `None`) and deregister it if necessary.

```
if self.handlerId is not None:
```

Deregistration requires two arguments: the name of the trigger and the unique handler identifier returned by the registration call.

```
chimera.triggers.deleteHandler('Atom', self.handlerId)
```

Set the unique handler identifier to `None` to indicate that no handler is currently registered.

```
self.handlerId = None
```

Define the module variable `dialog`, which tracks the dialog instance. It is initialized to `None`, and is assigned a usable value when the dialog is created.

```
dialog = None
```

Define `showColorWellUI`, which is invoked when the Chimera toolbar button is pressed.

```
def showColorWellUI():

      global dialog
      if dialog is not None:
            dialog.enter()
            return

      dialog = ColorWellDialog()


dir, file = os.path.split(__file__)
icon = os.path.join(dir, 'AtomTrigger.tiff')
chimera.tkgui.app.toolbar.add(icon, showColorWellUI, 'Set Main Chain Color', None)
```

Code Notes

Monitoring changes in atoms can result in many handler invocations. In an attempt to reduce computation, the example above deregisters its handler when the user interface is not being displayed.

Running the Example

The example code files must be saved in a directory named *`AtomTrigger`*. To run the example, start ***chimera*** , bring up the Tools preference category (via the `Preferences` entry in the `Favorites` menu; then choosing the "Tools" preference category) and use the `Add` button to add the directory above the *`AtomTrigger`* directory. Then type the following command into the IDLE command line:

```
import AtomTrigger.gui
```

This should display a button on the Chimera toolbar. Clicking the button should bring up a window with a color well inside. The color well may be used to manipulate the color of all protein backbone atoms. Changing atom colors through another mechanism, ***e.g.***the Midas emulator, should result in appropriate color changes in the color well.

Selections

The Selection class (and its subclasses) is used to manage collections of items derived from class Selectable. Molecules, Residues, Atoms, and Bonds are all derived from Selectable. The examples given will use these common Selectables for simplicity, but one should keep in mind that classes such as Model and PseudoBond are also Selectables and can be managed with Selections.

An important thing to understand is that a Selection is not necessarily a fixed set of Selectables. It may encapsulate an algorithm for choosing particular Selectables. For example, a Selection may be used to hold "all bonds/atoms in aromatic rings in all models". This Selection, when its contents are queried, would return differing results as models are opened/closed.

Every Selectable has an associated ***selection level*** :

- SelGraph (***e.g.*** Molecule)
- SelSubgraph (***e.g.*** Residue)
- SelVertex (***e.g.*** Atom)
- SelEdge (***e.g.*** Bond)

Any Selectable's selection level is returned by its oslLevel() member function.

Selections typically only explicitly hold vertices and edges. Higher-level Selectables' (graphs/subgraphs) membership in a Selection is computed from member vertices/edges. Selection methods that enumerate graphs can do so based on that graph's vertices/edges either being completely or partially present in the selection, as desired. Inclusion of graphs that have no vertex or edge sub-components (such as VRML models) is explicity tracked in a selection.

Since a Selection may encapsulate an algorithm for choosing Selectables, or instead may hold a specific set of Selectables, there is no method in the class Selection for specifying the items held in the Selection. Therefore, Selection is only used as a base class and all actual selections use subclasses. The basic subclasses Chimera defines in `chimera.selection` are the following:

ItemizedSelection: holds a fixed set of Selectables. However, Selectables ***will*** be deleted from the selection when the model(s) containing those Selectables are closed. Therefore, sometimes it is convenient to use an ItemizedSelection to track atoms, ***etc.*** , just to avoid having to write trigger-handling code yourself.

OrderedSelection: subclass of ItemizedSelection. Used in the infrequent case where the ordering of the Selectables is important (***e.g.*** structure matching). Ordering is only maintained relative to Selectables at the same selection level.

OSLSelection: holds Selectables based on an OSL string. "OSL" stands for Object Selection Language, and a rundown of OSL syntax can be found here. Whenever Selectables are requested from an OSLSelection, the OSL string will be re-evaluated to obtain the matching Selectables. Therefore, if the string contains an attribute test, it may return different Selectables at different times.

CodeSelection: uses a Python code string to determine what is selected. The code is expected to apply

functions (provided as local variables) to selected objects.

In addition to these classes, `chimera.selection` has many functions for manipulating the current selection shown in the Chimera graphics display. `help(chimera.selection)` in the IDLE shell will display information about them.

The code below demonstrates how to use a selection to hold the atoms/bonds of the protein mainchain, and then highlight them in the main Chimera graphics window.

Example *BackboneSel/__init__.py*

Import the standard modules used in this example.

```
import re
```

Import the Chimera modules used in this example.

```
import chimera
from chimera import selection
```

Define a function that will select protein backbone atoms in the main Chimera graphics window

```
def selBackbone(op=None):
```

Define a regular expression for matching the names of protein backbone atoms (we do not include the carbonyl oxygens because they tend to clutter up the graphics display without adding much information).

```
MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
```

The `list` method of chimera.openModels will return a list of currently open models, and takes several optional keyword arguments to restrict this list to models matching certain criteria. When called with no arguments, this method will return a list of all visible models, essentially models that were created by the user. Internally managed (`hidden`) models, such as the distance monitor pseudobondgroup, do not show up in this list. A list of hidden models can be obtained by setting the optional keyword argument `hidden` to True. The `all` argument (True/False) can be used to return a list of all open models (including both hidden and visible). Other optional arguments include:

`id` and `subid`, which restrict the returned list to models with the given id and subid, respectively, while `modelTypes` (a list of model types, i.e. `[chimera.Molecule]`) will restrict the returned list to models of a particular type.

```
bbAtoms = []
for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):

      for a in m.atoms:
            if MAINCHAIN.match(a.name):
                  bbAtoms.append(a)
```

Create a selection instance that we can use to hold the protein backbone atoms. We could have added the atoms one by one to the selection while we were in the above loop, but it is more efficient to add items in bulk to selections if possible.

```
backboneSel = selection.ItemizedSelection()
backboneSel.add(bbAtoms)
```

Add the connecting bonds to the selection. The `addImplied` method of `Selection` adds bonds if both bond endpoint atoms are in the selection, and adds atoms if any of the atom's bonds are in the selection. We use that method here to add the connecting bonds.

```
backboneSel.addImplied()
```

Change the selection in the main Chimera window in the manner indicated by this function's op keyword argument. If op is `None`, then use whatever method is indicated by the `Selection Mode` item in Chimera's Select menu. Otherwise, op should be one of: `selection.REPLACE`, `selection.INTERSECT`, `selection.EXTEND` or `selection.REMOVE`.

- `REPLACE` causes the Chimera selection to be replaced with `backboneSel`.
- `INTERSECT` causes the Chimera selecion to be intersected with `backboneSel`.
- `EXTEND` causes `backboneSel` to be appended to the Chimera selection.
- `REMOVE` causes `backboneSel` to be unselected in the Chimera window.

```
if op is None:
      chimera.tkgui.selectionOperation(backboneSel)
else:
      selection.mergeCurrent(op, backboneSel)
```

Code Notes

See the Registering Selectors example for how to make the "selBackbone" function available from the Chimera Select menu.

Running the Example

The example code files must be saved in a directory named *`BackboneSel`*. To run the example, start ***chimera*** , bring up the Tools preference category (via the `Preferences` entry in the `Favorites` menu; then choosing the "Tools" preference category) and use the `Add` button to add the directory above the *`BackboneSel`* directory. Then type the following command into the IDLE command line:

```
import BackboneSel

BackboneSel.selBackbone()
```

This will cause the selection state of all protein backbone atoms to change, depending on the Selection Mode chosen in the Chimera Select menu. If the mode is the default ("replace"), then the protein backbone will become selected and all other atoms/bonds will become deselected.

Saving Extension State in a Session File

This example shows how an extension can save its state in a Chimera session file. A session file is Python code that is executed by Chimera to restore the state information. The SimpleSession module saves the core Chimera state such as opened molecules and the camera settings. It then invokes a SAVE_SESSION trigger that extensions can receive to save their state.

Example *SessionSaveExample.py*

The code below saves state variables "some_path" and "some_number" in a session file when the Save Session menu entry is used.

```
some_path = '/home/smith/tralala.data'
some_number = 3

def restoreState(path, number):
  global some_path, some_number
  some_path = path
  some_number = number

def saveSession(trigger, x, file):
  restoring_code = \
"""
def restoreSessionSaveExample():
  import SessionSaveExample
  SessionSaveExample.restoreState(%s,%s)
try:
  restoreSessionSaveExample()
except:
  reportRestoreError('Error restoring SessionSaveExample')
"""
  state = (repr(some_path), repr(some_number))
  file.write(restoring_code % state)

import chimera
import SimpleSession
chimera.triggers.addHandler(SimpleSession.SAVE_SESSION, saveSession, None)
```

The last line registers the saveSession routine to be called when the SAVE_SESSION trigger is invoked. This happens when the user saves the session from the Chimera file menu. The saveSession routine writes Python code to the session file that will restore the state of the some_path and some_number global variables.

Large Data Structures

If you need to save data structures whose `repr()` would be hundreds or thousands of characters long, you should use SimpleSession's `sesRepr()` function instead which will insert newlines periodically. The resulting session has a better chance of being user-editable and passing through various mailer programs without corruption.

Session File Format

Here is the code written to the session file by the above example.

```
def restoreSessionSaveExample():
  import SessionSaveExample
  SessionSaveExample.restoreState('/home/smith/tralala.data','3')
try:
  restoreSessionSaveExample()
except:
  reportRestoreError('Error restoring SessionSaveExample')
```

Code written by other extensions will appear before and after this code. The restoreSessionSaveExample function is defined to keep extra names out of the global namespace of the session file. This is to avoid name conflicts. The restoreSessionSaveExample routine is called within a try statement so that if an error occurs it won't prevent code later in the session file from being called. The reportRestoreError routine is defined at the top of the session file by SimpleSession.

Saving Complex Data

The SessionUtil module helps writing and reading the state data for extensions that have alot of state. It can convert class instances to dictionaries so they can be written to a file. This is similar to the standard Python pickle module but provides better human readable formatted output. It can handle a tree of data structures. Look at the ScaleBar extension code to see how to use SessionUtil.

Restoring References to Molecular Data

You can save references to molecules, residues, atom, bonds, pseudobonds, VRML models, and MSMS surfaces by calling SimpleSession's `sessionID()` function with the item to save as the only argument. The `repr()` of the return value can then be written into the session file. During the restore, the written session ID value should be given to SimpleSession's `idLookup()` function, which will return the corresponding molecular data item.

Custom Molecular Attributes

If you add non-standard attributes to Molecules/Residues/Bonds/Atoms that you want preserved in restored sessions, use SimpleSession's `registerAttribute()` function, ***e.g,***

```
  import chimera
  from SimpleSession import registerAttribute
  registerAttribute(chimera.Molecule, "qsarVal")
```

Note that the Define Attribute tool automatically does this registration, so it's only attributes that you add directly from your own Python code that need to be registered as above. Also, only attributes whose values are recoverable from their `repr()` can be saved by this mechanism, so values that are C++ types (Atoms, MaterialColors, ***etc.***) could not be preserved this way.

Restoring and Referring to Special Models

Non-molecular models cannot be located with the `sessionID()`/ `idLookup()` mechanism. Instead, SimpleSession has a `modelMap` dictionary that can be used to map between a id/subid tuple for a saved model

and the actual restored model. So, during a session save, you would get the tuple to save from a model with code like:

```
refToSave = (model.id, model.subid)
```

The values in `modelMap` are actually lists of models with those particular id/subid values, since multiple models may have the same id/subid (***e.g.*** a molecule and its surface). So if you are restoring a special model and want to update `modelMap`, you would use code like this:

```
import SimpleSession
SimpleSession.modelMap.setdefault(refToSave, []).append(restoredModel)
```

Keep in mind that the id/subid of the saved model may not be the same as the restored model, particularly if sessions are being merged. The key used by `modelMap` is always the id/subid of the saved model.

If you are trying to refer to a non-molecular model using `modelMap`, and that non-molecular model is of class ***X***, you would use code like this:

```
import SimpleSession
restoredModel = filter(lambda m: isinstance(m, X), SimpleSession.modelMap[refToSave])[0]
```

Session Merging

If you are restoring your own models, you should try to restore them into their original ids/subids if possible so that scripts and so forth will continue to work across session saves. In the case of sessions being merged, this of course could be problematic. SimpleSession has a variable `modelOffset` (***i.e.*** `SimpleSession.modelOffset`) which should be added to your model's id to avoid conflicts during session merges. `modelOffset` is zero during a non-merging session restore.

Special Molecular/VRML Models

Some extensions may create their own Molecule or VRML instances that need to be restored by the extension itself rather than by the automatic save/restore for Molecules/VRML provided by SimpleSession. In order to prevent SimpleSession from attempting to save the instance, use the `noAutoRestore()` function once the instance has been created, like this:

```
import SimpleSession
SimpleSession.noAutoRestore(instance)
```

Your extension is responsible for restoring all aspects of the instance, including selection state.

Post-Model Restore

If restoring code should be called only after all models have been restored then the SimpleSession.registerAfterModelsCB routine should be used.

```
def afterModelsRestoredCB(arg):
  # do some state restoration now that models have been created
```

```
  import SimpleSession
  SimpleSession.registerAfterModelsCB(afterModelsRestoredCB, arg)
```

The 'arg' can be omitted in both the registration and callback functions.

Saving Colors

Similar to `sessionID` for molecular data, there is `colorID` function that returns a value whose `repr()` can be saved into a session file. During a restore, that value can be given to SimpleSession's `getColor` function to get a Color object back.

Closing a Session

The Close Session entry in the Chimera file menu is meant to reset the state of Chimera, unloading all currently loaded data. Extensions can reset their state when the session is closed by handling the CLOSE_SESSION trigger as illustrated below.

```
def closeSession(trigger, a1, a2):
  default_path = '/default/path'
  default_number = 1
  restoreState(default_path, default_number)

import chimera
chimera.triggers.addHandler(chimera.CLOSE_SESSION, closeSession, None)
```

Changing Behavior During a Session Restore

If an extension needs to behave differently during a session restore than at other times (***e.g.***react differently to newly opened models), then it can register for the BEGIN_RESTORE_SESSION and END_RESTORE_SESSION triggers, in an analogous manner to the CLOSE_SESSION trigger in the preceding section.

Running the Example

To try the example, save the above sections of code shown in red as file ***SessionSaveExample.py*** . Use the Chimera Favorites/Preferences/Tools/Locations interface to add the directory where you have placed the file to the extension search path. Show the Python shell window using the Tools/Programming/IDLE menu entry and type the following command.

```
>>> import SessionSaveExample
```

Now save the session with the File/Save Session As... menu entry and take a look at the session file in an editor. You should be able to find the restoreSessionSaveExample code in the file. The current value of the some_path variable can be inspected as follows.

```
>>> SessionSaveExample.some_path
'/home/smith/tralala.data'
```

Now close the session with the File/Close Session menu entry and see that the some_path variable has been reset to the default value.

```
>>> SessionSaveExample.some_path
'/default/path'
```

Now reload the session with the File/Open Session menu entry and see that the some_path variable has been restored.

```
>>> SessionSaveExample.some_path
'/home/smith/tralala.data'
```

Measure Atomic-Level Quantities

This example shows how to measure atomic-level (rather than volumetric) quantities such and angles, RMSDs, surface areas, and so forth.

Atomic Coordinates in Chimera

The first concept to understand is that when models are moved in Chimera, the atomic coordinates are not changed. Instead, a "transformation" matrix is updated that transforms the model's original coordinates into "world" coordinates (***i.e.*** into the overall coordinate system). Consequently, there are two methods for obtaining a coordinate from a chimera.Atom object: `coord()`, which returns the Atom's original coordinate, and `xformCoord()`, which returns the transformed coordinate. Note that some structure-editing activities in Chimera ***will*** change the original coordinates (***e.g.*** changing a torsion angle).

Therefore, if you are measuring quantities that might involve multiple models, you should use the `xformCoord()` method. If your measurements are completely intra-model you can instead use the very slightly faster `coord()` method.

Getting Atoms, Bonds, Residues

The "Chimera's Object Model" example discusses how to access various Chimera objects in detail, but here's an executive summary for the tl;dr crowd:

Getting a list of open chimera.Molecule models

```
from chimera import OpenModels, Molecule
mols = OpenModels.list(modelTypes=[Molecule])
```

Getting lists of chimera.Atoms/Bonds/Residues from a chimera.Molecule object

The Atoms/Bonds/Residues in a chimera.Molecule object are contained in that object's `atoms`/`bonds`/`residues` attributes, respectively.

Getting Atoms/Bonds/Residues from the current selection

To get the Atoms/Bonds/Residues in the current selection (perhaps set by the user or earlier in the code via the runCommand() function), use the `currentAtoms`/`currentBonds`/`currentResidues` functions in the chimera.selection module, ***e.g.***

```
from chimera.selection import currentAtoms
```

```
sel_atoms = currentAtoms()
```

Getting Atoms/Bonds/Residues/Molecules from related Atoms/Bonds/Residues

Here are some import methods/attributes for accessing Atoms/Bonds/Residues related to other Atoms/Bonds/Residues:

Atom `.neighbors`

Atom `.primaryNeighbors()`
: Returns a list of the Atoms bonded to the given Atom. Some high-resolution structures can have multiple positions for a single atom, and in those cases `primaryNeighbors()` will only return one Atom among those positions whereas `neighbors` will return all of them.

Atom `.bonds`

Atom `.primaryBonds()`
: Returns a list of the Bonds the Atom is involved in. `primaryBonds()` is analogous to ***Atom*** `.primaryNeighbors()`.

Atom `.bondsMap`
: Returns a dictionary whose keys are Atoms the given Atom is bonded to, and the values are the corresponding Bonds.

Atom `.residue`
: Returns the Residue the Atom is in.

Atom `.molecule`
: Returns the Molecule model the Atom is in.

Bond `.atoms`
: Returns a list of the two Atoms forming the Bond.

Residue `.atoms`
: Returns a list of the Atoms in the Residue.

Residue `.atomsMap`
: Returns a dictionary whose keys are atom names. The values are ***lists*** of Atoms with the corresponding name. The values are lists because in some structure formats (***e.g.***Mol2, XYZ) small molecules atoms are not given unique names (for example, all carbons are named "C"). Also, PDB files where an atom has alternate locations will produce multiple Atoms with the same name in a Residue.

Residue `.molecule`
: Returns the Molecule model the Residue is in.

Point objects

Both the ***Atom*** `.coord()` and ***Atom*** `.xformCoord()` methods return chimera.Point objects. Point objects have the following built-in measurement methods:

Point `.distance(`***Point*** `)`
: Returns the distance in angstroms between the two Points.

Point `.sqdistance(`***Point*** `)`
: Returns the square of the distance in angstroms between the two Points. Taking square roots is slow, so this method is faster than the `distance()` method. Therefore in code where speed is important, when possible you should work with squares of distances rather than the distances

themselves (***e.g.*** when comparing a distance to a cutoff value, compare the squares instead [and make sure to only compute the square of the cutoff once!]).

Basic Measurement Functions

The chimera module offers several basic measurement functions:

`chimera.distance(`***Point*** `,` ***Point*** `)`
`chimera.sqdistance(`***Point*** `,` ***Point*** `)`
Returns the distance (or distance squared) in angstroms between the two Points. Functionally identical to ***Point*** .distance(***Point***) and ***Point*** .sqdistance(***Point***) methods respectively.
`chimera.angle(`***Point*** `,` ***Point*** `,` ***Point*** `)`
Returns the angle in degrees formed by the points. The angle value ranges from 0 to 180.
`chimera.dihedral(`***Point*** `,` ***Point*** `,` ***Point*** `,` ***Point*** `)`
Returns the dihedral angle in degrees formed by the points. The angle value ranges from -180 to 180. Note that Residues have ***phi*** , ***psi*** , and ***chi1*** through ***chi4*** attributes that can be queried for the corresponding values (value will be ***None*** if the Residue lacks that kind of angle). In fact, those attributes can be set and the structure will be adjusted appropriately!

Here's a simple code snippet for finding the angle between three atoms (***a1, a2, a3***) that may not all be in the same model (and therefore need to have the `xformCoord()` method used to fetch their coordinates):

```
import chimera
angle = chimera.angle(a1.xformCoord(), a2.xformCoord(), a3.xformCoord())
```

Alternatively, if the three atoms are in a list (***atoms***), you can use slightly fancier Python:

```
import chimera
angle = chimera.angle(*[a.xformCoord() for a in atoms])
```

Axes, Planes, Centroids

Preliminaries

The `centroid`, `axis`, and `plane` functions described below utilize chimera.Point, chimera.Vector, and chimera.Plane objects for some of their return values. A chimera.Point object, described previously, abstracts a point in Cartesian 3-space. A chimera.Vector objects abstracts a direction vector in 3-space with finite length (***i.e.*** it is not infinite and has an associated length). A chimera.Plane object abstracts an infinite plane in 3-space. Each of these objects has useful member functions that you can learn about by using the ***help*** Python function in the IDLE tool (***e.g.*** `help(chimera.Plane)`). For instance, if ***p*** is a Plane and ***pt*** is a Point, then ***p***.`distance(`***pt*** `)` is the distance from the Point ***pt*** to the Plane ***p***.

The axis and plane functions take an ***n***-by-3 numpy array as one of their input arguments. The easiest way to generate such an array from Atom coordinates is to use the numpyArrayFromAtoms function from the ***chimera*** module (***i.e.*** chimera.numpyArrayFromAtoms(***Atoms***) or, if transformed coordinates are required, chimera.numpyArrayFromAtoms(***Atoms***, xformed=True)).

The Functions

The ***StructMeasure*** module has three convenient functions for finding the best-fitting axis, centroid, or plane through a set of points. They are:

StructMeasure.centroid(***points***, weights=None)
: Returns a chimera.Point object. ***points*** is a sequence of chimera.Point objects. weights is an optional sequence of corresponding numeric weights to give those Points when computing the centroid. weights is most frequently used when mass weighting is desired. To that end, it is useful to know that the mass of atom ***a*** is given by ***a***.element.mass.

StructMeasure.axis(***xyzs***, findBounds=False, findRadius=False, iterate=True, weights=None)
: Returns a chimera.Point and chimera.Vector. The Point is the center of the axis, and the Vector indicates the direction of the axis (and is of unit length). As discussed in **Preliminaries**, ***xyzs*** is an ***n***-by-3 numpy array. If findBounds is True, two floating point numbers are appended to the return values, indicating the scaling values needed for the Vector to reach the approximate end of the axis given the input coordinates. One of the scaling values will be negative. If findRadius is True, a floating point number, indicating the approximate radius of the axis given the input coordinates, will be appended to the return values. If iterate is True, the best-fitting axis as determined by principal-component analysis will be iteratively tweaked to try to get the axis as equidistant as possible from the points determining the axis. For helical sets of atoms, the principal-component axis will tend to tilt towards the final atoms of the helix. The tilt is more pronounced the shorter the helix, and iterate attempts to correct the tilt. weights is the same as in the centroid function.

StructMeasure.plane(***xyzs***, findBounds=False)
: Returns a chimera.Plane whose origin is the centroid of ***xyzs*** As discussed in **Preliminaries**, ***xyzs*** is an ***n***-by-3 numpy array. If findBounds is True, a Point, which represents the furthest ***xyz*** from the origin when projected onto the Plane, is appended to the return value.

Surface Areas

Once a surface has been computed for a model, all Atoms and Residues of that model will have an areaSAS attribute (solvent accessible surface area) and an areaSES attribute (solvent excluded surface area). One possible way to get a surface computed for a model is to call the ***surface*** command via the runCommand() function.

Running a Background Process

Chimera may be used in conjunction with command line programs, ***e.g.***Modeller, by creating the required input files, executing the program, and importing the generated results, ***e.g.***as attributes of molecules.

Example *RunSubprocess.py*

Class `CountAtoms` assigns two attributes, "numAtoms" and "numHetatms", to a molecule by exporting the molecule as a PDB file and running the "grep" program twice. The "grep" invocations are run in the background so that Chimera stays interactive while they execute.

```
class CountAtoms:
```

The constructor sets up a temporary file for the PDB output, and a Chimera task instance for showing progress to the user.

```
def __init__(self, m, grepPath):
```

Generate a temporary file name for PDB file. We use Chimera's `osTemporaryFile` function because it automatically deletes the file when Chimera exits.

```
import OpenSave
self.pdbFile = OpenSave.osTemporaryFile(suffix=".pdb", prefix="rg")
self.outFile = OpenSave.osTemporaryFile(suffix=".out", prefix="rg")
```

Write molecule in to temporary file in PDB format.

```
self.molecule = m
import Midas
Midas.write([m], None, self.pdbFile)
```

Set up a task instance for showing user our status.

```
from chimera import tasks
self.task = tasks.Task("atom count for %s" % m.name, self.cancelCB)
```

Start by counting the ATOM records first.

```
self.countAtoms()
```

`cancelCB` is called when user cancels via the task panel

```
def cancelCB(self):

    self.molecule = None
```

`countAtoms` uses "grep" to count the number of ATOM records.

```
def countAtoms(self):

    from chimera import SubprocessMonitor as SM
    self.outF = open(self.outFile, "w")
```

```
        self.subproc = SM.Popen([ grepPath, "-c", "^ATOM", self.pdbFile ],
        stdout=self.outF)
        SM.monitor("count ATOMs", self.subproc, task=self.task, afterCB=self.
        _countAtomsCB)
```

_countAtomsCB is the callback invoked when the subprocess started by countAtoms completes.

```
def _countAtomsCB(self, aborted):
```

Always close the open file created earlier

```
        self.outF.close()
```

If user canceled the task, do not continue processing.

```
        if aborted or self.molecule is None:

            self.finished()
            return
```

Make sure the process exited normally.

```
        if self.subproc.returncode != 0 and self.subproc.returncode != 1:

            self.task.updateStatus("ATOM count failed")
            self.finished()
            return
```

Process exited normally, so the count is in the output file. The error checking code (in case the output is not a number) is omitted to keep this example simple.

```
        f = open(self.outFile)
        data = f.read()
        f.close()
        self.molecule.numAtoms = int(data)
```

Start counting the HETATM records

```
        self.countHetatms()
```

countHetatms uses "grep" to count the number of HETATM records.

```
def countHetatms(self):

        from chimera import SubprocessMonitor as SM
        self.outF = open(self.outFile, "w")
        self.subproc = SM.Popen([ grepPath, "-c", "^HETATM", self.pdbFile ],
        stdout=self.outF)
        SM.monitor("count HETATMs", self.subproc, task=self.task, afterCB=self.
        _countHetatmsCB)
```

_countHetatmsCB is the callback invoked when the subprocess started by countHetatms completes.

```
def _countHetatmsCB(self, aborted):
```

Always close the open file created earlier

```
self.outF.close()
```

If user canceled the task, do not continue processing.

```
if aborted or self.molecule is None:

    self.finished()
    return
```

Make sure the process exited normally.

```
if self.subproc.returncode != 0 and self.subproc.returncode != 1:

    self.task.updateStatus("HETATM count failed")
    self.finished()
    return
```

Process exited normally, so the count is in the output file. The error checking code (in case the output is not a number) is omitted to keep this example simple.

```
f = open(self.outFile)
data = f.read()
f.close()
self.molecule.numHetatms = int(data)
```

No more processing needs to be done.

```
self.finished()
```

`finished` is called to clean house.

```
def finished(self):
```

Temporary files will be removed when Chimera exits, but may be removed here to minimize their lifetime on disk. The task instance must be notified so that it is labeled completed in the task panel.

```
self.task.finished()
```

Set instance variables to None to release references.

```
self.task = None
self.molecule = None
self.subproc = None
```

Below is the main program. First, we find the path to the "grep" program. Then, we run CountAtoms for each molecule.

```
from CGLutil import findExecutable
grepPath = findExecutable.findExecutable("grep")
if grepPath is None:

    from chimera import NonChimeraError
```

```
        raise NonChimeraError("Cannot find path to grep")
```

Add "numAtoms" and "numHetatms" attributes to all open molecules.

```
import chimera
from chimera import Molecule
for m in chimera.openModels.list(modelTypes=[Molecule]):

        CountAtoms(m, grepPath)
```

Running the Example

You can execute the example code by downloading the linked Python script and opening it with the File→Open menu item or with the open command. Note that the ***.py*** extension is required for the open dialog/command to recognize that the file is a Python script.

Writing a C/C++ Chimera Extension

Caveat

The header files declaring Molecules, Residues, ***etc.*** , are available for download. This allows C/C++ extensions to work with molecular data classes, but not add methods and/or data to those classes. The support for inserting methods and data into molecular classes will be made available some time later in the Chimera "developer" release.

What you can do now

Basically, you write an extension conforming to Python's normal C/C++ API. Once you have a compiled shared library, put it in Chimera's lib subdirectory and you will then be able to import it and use its functions from your Python code.

Creating Molecules

Molecules may be created using only Python code. The following example shows how to create a single water molecule.

Example *CreateMolecule.py*

Function `createWater` creates a water molecule.

```
def createWater():
```

Import the object that tracks open models and several classes from the `chimera` module.

```
from chimera import openModels, Molecule, Element, Coord
```

Create an instance of a Molecule

```
m = Molecule()
```

Molecule contains residues. For our example, we will create a single residue of HOH. The four arguments are: the residue type, chain identifier, sequence number and insertion code. Note that a residue is created as part of a particular molecule.

```
r = m.newResidue("HOH", " ", 1, " ")
```

Now we create the atoms. The newAtom function arguments are the atom name and its element type, which must be an instance of Element. You can create an Element instance from either its name or atomic number.

```
atomO = m.newAtom("O", Element("O"))
atomH1 = m.newAtom("H1", Element(1))
atomH2 = m.newAtom("H2", Element("H"))
```

Set the coordinates for the atoms so that they can be displayed.

```
from math import radians, sin, cos
bondLength = 0.95718
angle = radians(104.474)
atomO.setCoord(Coord(0, 0, 0))
atomH1.setCoord(Coord(bondLength, 0, 0))
atomH2.setCoord(Coord(bondLength * cos(angle), bondLength * sin(angle), 0))
```

Next, we add the atoms into the residue.

```
r.addAtom(atomO)
r.addAtom(atomH1)
r.addAtom(atomH2)
```

Next, we create the bonds between the atoms.

```
m.newBond(atomO, atomH1)
m.newBond(atomO, atomH2)
```

Finally, we add the new molecule into the list of open models.

```
openModels.add([m])
```

Call the function to create a water molecule.

```
createWater()
```

Code Notes

If multiple water molecules were needed, they should probably be created as multiple residues (with different sequence numbers) in the same molecule.

Running the Example

You can execute the example code by downloading the linked Python script and opening it with the File→Open menu item or with the open command. Note that the ***.py*** extension is required for the open dialog/command to recognize that the file is a Python script.

Chimera Menu/Widget Text Guidelines

I. Goals

- to promote consistent text usage in Chimera's user interfaces
- to provide guidelines for developers and programmers
- to promote awareness and discussion

II. Font

The default font type and size should be used.

III. Menus

IIIa. General Scheme

```
Primary (one word, noun or verb, capitalized)
  Secondary (words capitalized as in a title)
    tertiary or lower (numeral or lowercase,
                       except proper nouns)
```

Proper nouns include atom types, elements, and extension (tool) names.

Examples:

```
Select
  Chemistry
    element
      other
        Fe-Hg
          Fe
          (etc.)
  Residue
    amino acid category
      aliphatic
      (etc.)
  Selection Mode (replace)
    append
    (etc.)
Actions
  Atoms/Bonds
    wire width
      1
      (etc.)
  Ribbon
    show
    (etc.)
```

```
Tools
  Utilities
    Browser Configuration
    (etc.)
```

IIIb. Usage of Ellipses

The ellipsis string "..." should indicate a menu item that opens an additional interface which requires user input to accomplish the function described by its name (one-shot panels, as opposed to those intended to be left up for a series of user interactions). For now, **Tools** are exempted from this guideline.

We decided that "..." should not indicate a menu item which simply opens an additional interface, since practically all items would then require it.

There also needs to be consistency in whether "..." is preceded by a space; we recommend no space.

Finally, should "..." appear on buttons as well as menu items? If so, the same criteria should apply.

IV. Widgets (GUIs)

This is the broadest grouping, and thus less amenable to standardization. It includes panels and dialog boxes generated by built-in functions as well as extensions to Chimera. General recommendations:

- **Title of Widget**
  - one or more words to appear on the top bar, capitalized as a title, no colon or period at the end; should be the same text as the invoking menu item or button (except sans any "...")
- **Brief Header** for a section
  - capitalized as a title, optional colon at the end (but no colon when sections are treated as "index cards")
- **Longer description of a section**
  - first word capitalized, subsequent words not capitalized unless proper nouns or acronyms; optional colon at the end, no period
- **Instructive statement**
  - first word capitalized, subsequent words not capitalized unless proper nouns or acronyms; no period
  Example:
  Ctrl-click on histogram to add or delete thresholds in the **Volume Viewer Display** panel
- **[box] Description next to a checkbox**
  - first word capitalized, subsequent words not capitalized unless proper nouns or acronyms; no period or question mark
    - exception: when the checkbox indicates a section to be expanded/compacted, the text may be capitalized as a title (instead of only the first word being capitalized).
- **Item name: [blank, color well, slider, pulldown menu or checkbox list]**
  or (especially if there are many of these in the widget)
  item name: [blank, color well, slider, pulldown menu or checkbox list]
  - first word of item name optionally capitalized, subsequent words not capitalized unless proper nouns or acronyms; colon separating the item name from the value(s); options in a pulldown menu or checkbox list not capitalized unless proper nouns or acronyms
    - exception: when the item name and pulldown option together describe a section, both should be capitalized and the colon is optional

Examples:
Inspect [Atom/etc.] in the **Selection Inspector**
Category: [New Molecules/etc.] in the **Preferences Tool**

- **Phrase with [blank, color well, pulldown menu, or checkbox list] embedded**
  - first word capitalized, no colon, period or question mark; the blank (etc.) should not start the phrase
- **Phrase with [button] embedded**
  - 1-2 words actually on the button, others trailing and/or preceding; the first word should be capitalized whether or not on the button; no colon, period or question mark; the button may start the phrase
- buttons marked **OK**, **Apply**, **Cancel**, **Help**
  - common but optional
- widget-specific buttons
  - 1-2 words, each capitalized if the button brings up another panel, at least the first word capitalized otherwise; if another panel is evoked, consider using "..."

```
#    Import Chimera modules used in this example.
import chimera

#    First, we'll open up a model to work with. This molecule (4fun) is very small,
#    comprised of just a couple residues, but it is perfect for illustrating
#    some important components of Chimera's object model.
#    For more information on how to open/close models in Chimera, see the
#    "Basic Model Manipulation" Example in the Programmer's Guide (coming soon). For now,
#    just understand that this code opens up any molecules stored in the file
#    "4fun.pdb" and returns a list of references to opened models.
#    (Put 4fun.pdb on your desktop or change the path in the command below.)
#
#    .. "4fun.pdb" 4fun.pdb
opened = chimera.openModels.open('~/Desktop/4fun.pdb')

#    Because only one molecule was opened, 'opened' is a list with just one element.
#    Get a reference to that element (which is a 'Molecule'
#    instance) and store it in 'mol'
mol = opened[0]


#    Now that we have a molecule to work with, an excellent way of examining its data structures is to flatten it out and write
#    it to a file. We'll write this file in the 'mol2' format, a free-format ascii file that describes molecular structure.
#    It is not necessary to have any prior knowledge of the 'mol2' format to understand this example, just a basic
#    comprehension of file formats that use coordinate data. Check out the "finished product".
#    It should serve as a good reference while you're going through the example.
#    Get a reference to a file to write to:
#
#    .. "finished product" 4fun.mol2
f = open("4fun.mol2", 'w')


#    mol2 uses a series of Record Type Indicators (RTI), that indicate the type of structure that will be described in
#    the following lines.
#    An RTI is simply an ASCII string which starts with an asterisk ('@'), followed by a string of characters,
#    and is terminated by a new line.
#    Here, we define some RTI's that we will use througout the file to describe the various parts of our model:

MOLECULE_HEADER = "@<TRIPOS>MOLECULE"
ATOM_HEADER     = "@<TRIPOS>ATOM"
BOND_HEADER     = "@<TRIPOS>BOND"
SUBSTR_HEADER   = "@<TRIPOS>SUBSTRUCTURE"


#    The 'chimera2sybyl' dictionary is used to map Chimera atom types
#    to Sybyl atom types.  See section below on writing out per-atom
#    information.
chimera2sybyl = {
       'C3'  : 'C.3',     'C2'  : 'C.2',     'Car' : 'C.ar',    'Cac' : 'C.2',
       'C1'  : 'C.1',     'N3+' : 'N.4',     'N3'  : 'N.3',     'N2'  : 'N.2',
       'Npl' : 'N.pl3',   'Ng+' : 'N.pl3',   'Ntr' : 'N.2',     'Nox' : 'N.4',
       'N1+' : 'N.1',     'N1'  : 'N.1',     'O3'  : 'O.3',     'O2'  : 'O.2',
```

```
      'Oar' : 'O.2',     'O3-' : 'O.co2',   'O2-' : 'O.co2',   'S3+' : 'S.3',
      'S3'  : 'S.3',     'S2'  : 'S.2',     'Sac' : 'S.O2',    'Son' : 'S.O2',
      'Sxd' : 'S.O',     'Pac' : 'P.3',     'Pox' : 'P.3',     'P3+' : 'P.3',
      'HC'  : 'H',       'H'   : 'H',       'DC'  : 'H',       'D'   : 'H',
      'P'   : 'P.3',     'S'   : 'S.3',     'Sar' : 'S.2',     'N2+' : 'N.2'
}

#    Writing Out per-Molecule Information
#    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#    The "<TRIPOS>MOLECULE" RTI indicates that the next couple of lines will contain information relevant
#    to the molecule as a whole. First, write out the Record Type Indicator (RTI):
f.write("%s\n" % MOLECULE_HEADER)

#    The next line contains the name of the molecule. This can be accessed through the 'mol.name' attribute.
#    (Remember, 'mol' is a reference to the molecule we opened). If the model you open came from a pdb file, 'name' will most
#    often be the name of the file (without the '.pdb' extension).  For this example, 'mol.name' is "4fun".
f.write("%s\n" % mol.name)

#    Next, we need to write out the number of atoms, number of bonds, and number of substructures in the model (substructures
#    can be several different things; for the sake of simplicity, the only substructures we'll worry about here are residues).
#    This data is accessible through attributes of a molecule object: 'mol.atoms', 'mol.bonds', and 'mol.residues' all contain
#    lists of their respective components. We can determine how many atoms, bonds, or residues this
#    molecule has by taking the 'len' of the appropriate list.
#    save the list of references to all the atoms in 'mol':
ATOM_LIST = mol.atoms
#    save the list of references to all the bonds in 'mol':
BOND_LIST = mol.bonds
#    save the list of references to all the residues in 'mol':
RES_LIST  = mol.residues

f.write("%d %d %d\n" % ( len(ATOM_LIST), len(BOND_LIST), len(RES_LIST)) )

#    type of molecule
f.write("PROTEIN\n")

#    indicate that no charge-related information is available
f.write("NO_CHARGES\n")

f.write("\n\n")


#    Writing Out per-Atom Information
#    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#    Next, write out atom-related information. In order to indicate this, we must first write out the
#    atom RTI:
f.write("%s\n" % ATOM_HEADER)

#   Each line under the 'ATOM' RTI consists of information pertaining to a single atom. The following information about each
#   atom is required: an arbitrary atom id number, atom name, x coordinate, y coordinate, z coordinate, atom type, id of the
#   substructure to which the atom belongs , name of the substructure to which the atom belongs.
```

```
#    You can look at each atom in the molecule by looping through its 'atoms' attribute.
#    Remember, 'ATOM_LIST' is the list of atoms stored in 'mol.atoms.' It's more efficient
#    to get the list once, and assign it to a variable, then to repeatedly ask for 'mol.atoms'.
for atom in ATOM_LIST:
    #    Now that we have a reference to an atom, we can write out all the necessary information to the file.
    #    The first field is an arbitrary id number. We'll just use that atom's index within the 'mol.atoms' list.
    f.write("%d " % ATOM_LIST.index(atom) )

    #    Next, we need the name of the atom, which is accessible via the 'name' attribute.
    f.write("%s " % atom.name)

    #    Now for the x, y, and z coordinate data.
    #    Get the atom's 'xformCoord' object. This is essentially a wrapper that holds information about the
    #    coordinate position (x,y,z) of that atom. 'xformCoord.x', 'xformCoord.y', and 'xformCoord.z' store the x, y,
    #    and z coordinates,
    #    respectively, as floating point integers. This information comes from the coordinates given for each atom
    #    specification in the input file
    coord = atom.xformCoord()
    f.write("%g %g %g " % (coord.x, coord.y, coord.z) )

    #    The next field in this atom entry is the atom type. This is a string which stores information about the
    #    chemical properties of the atom. It is accessible through the 'idatmType' attribute of an atom object.
    #    Because Chimera uses slightly different atom types than SYBYL (the modeling program for which .mol2 is the primary
    #    input format), use a dictionary called chimera2sybyl (defined above) that converts Chimera's atom types to
    #    the corresponding SYBYL version of the atom's type.
    f.write("%s " % chimera2sybyl[atom.idatmType])

    #    The last two fields in an atom entry pertain to any substructures to which the atom may belong.
    #    As previously noted, we are only interested in residues for this example.
    #    Every atom object has a 'residue' attribute, which is a reference to the residue to which that atom belongs.
    res   = atom.residue

    #    Here, we'll use 'res.id' for the substructure id field. 'res.id' is a string which represents a unique id
    #    for that residue (a string representation of a number, i.e. "1" , which are sequential, for all the
    #    residues in a molecule).
    f.write("%s " % res.id)

    #    The last field to write is substructure name. Here, we'll use the 'type' attribute of 'res'. the 'type' attribute contains
    #    a string representation of the residue type (e.g. "HIS", "PHE", "SER"...).  Concatenate onto this the residue's 'id'
    #    to make a unique name for this substructure (because it is possible, and probable, to have more than one
    #    "HIS" residue in a molecule. This way, the substructure name will be "HIS6" or "HIS28")
    f.write("%s%s\n" % (res.type, res.id) )

f.write("\n\n")


#    Writing Out per-Bond Information
#    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#    Now for the bonds. The bond RTI says that the following lines will contain information about bonds.
f.write("%s\n" % BOND_HEADER)
```

```
#    Each line after the bond RTI contains information about one bond in the molecule.
#    As noted earlier, you can access all the bonds in a molecule through the 'bonds' attribute,
#    which contains a list of bonds.
for bond in BOND_LIST:

    #    each bond object has an 'atoms' attribute, which is list of length 2, where each item in the list is
    #    a reference to one of the atoms to which the bond connects.
    a1, a2 = bond.atoms

    #    The first field in a mol2 bond entry is an arbitrary bond id. Once again, we'll just use that
    #    bond's  index in the 'mol.bonds' list
    f.write("%d " % BOND_LIST.index(bond) )

    #    The next two fields are the ids of the atoms which the bond connects. Since we have a reference to both these
    #    atoms (stored in 'a1' and 'a2'), we can just get the index of those objects in the 'mol.atoms' list:
    f.write("%s %s " % (ATOM_LIST.index(a1), ATOM_LIST.index(a2)) )

    #    The last field in this bond entry is the bond order. Chimera doesn't currently calcuate bond orders,
    #    but for our educational purposes here, this won't be a problem.
    #    The mol2 format expects bond order as a string: "1" (first-order), "2" (second-order), etc.,  so
    #    just write out "1" here (even though this may not be correct).
    f.write("1\n")

f.write("\n\n")


#    Writing Out per-Residue Information
#    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#    Almost done!!! The last section contains information about the substructures (i.e. residues for this example)
#    You know the drill:
f.write("%s\n" % SUBSTR_HEADER)

#    We've already covered some of these items (see above):
for res in RES_LIST:
    #    residue id field
    f.write("%s " % res.id )

    #    residue name field
    f.write("%s%s " % (res.type, res.id) )

    #    the next field specifies the id of the root atom of the substructure. For the case of residues,
    #    we'll use the alpha-carbon as the root.
    #    Each residue has an 'atomsMap' attribute which is a dictionary. The keys in this dictionary are
    #    atom names (e.g. 'C', 'N', 'CA'), and the values are lists of references to atoms in the residue that have that
    #    name. So, to get the alpha-carbon of this residue:
    alpha_carbon = res.atomsMap['CA'][0]

    #    and get the id of 'alpha_carbon' from the 'mol.atoms' list
    f.write("%d " % ATOM_LIST.index(alpha_carbon) )


    #    The final field of this substructure entry is a string which specifies what type of substructure it is:
```

```
    f.write("RESIDUE\n")

f.write("\n\n")
f.close()

#    And that's it! Don't worry if you didn't quite understand all the ins and outs of the mol2 file format.
#    The purpose of this exercise was to familiarize yourself with Chimera's object model; writing out a mol2 file
#    was just a convenient way to do that. The important thing was to gain an understanding of how Chimera's atoms,
#    bonds, residues, and molecules all fit together.
```

```
@<TRIPOS>MOLECULE
4fun.pdb
47 48 6
PROTEIN
NO_CHARGES


@<TRIPOS>ATOM
0 N 49.668 24.248 10.436 N.4 1 HIS1
1 CA 50.197 25.578 10.784 C.3 1 HIS1
2 C 49.169 26.701 10.917 C.2 1 HIS1
3 O 48.241 26.524 11.749 O.2 1 HIS1
4 CB 51.312 26.048 9.843 C.3 1 HIS1
5 CG 50.958 26.068 8.34 C.ar 1 HIS1
6 ND1 49.636 26.144 7.86 N.pl3 1 HIS1
7 CD2 51.797 26.043 7.286 C.ar 1 HIS1
8 CE1 49.691 26.152 6.454 C.ar 1 HIS1
9 NE2 51.046 26.09 6.098 N.pl3 1 HIS1
10 N 49.788 27.85 10.784 N.pl3 2 SER2
11 CA 49.138 29.147 10.62 C.3 2 SER2
12 C 47.713 29.006 10.11 C.2 2 SER2
13 O 46.74 29.251 10.864 O.2 2 SER2
14 CB 49.875 29.93 9.569 C.3 2 SER2
15 OG 49.145 31.057 9.176 O.3 2 SER2
16 N 47.62 28.367 8.973 N.pl3 3 GLN3
17 CA 46.287 28.193 8.308 C.3 3 GLN3
18 C 45.406 27.172 8.963 C.2 3 GLN3
19 O 44.198 27.508 9.014 O.2 3 GLN3
20 CB 46.489 27.963 6.806 C.3 3 GLN3
21 CG 45.138 27.8 6.111 C.3 3 GLN3
22 CD 45.304 27.952 4.603 C.2 3 GLN3
23 OE1 46.432 28.202 4.112 O.2 3 GLN3
24 NE2 44.233 27.647 3.897 N.pl3 3 GLN3
25 N 46.014 26.394 9.871 N.pl3 4 GLY4
26 CA 45.422 25.287 10.68 C.3 4 GLY4
27 C 43.892 25.215 10.719 C.2 4 GLY4
28 O 43.287 26.155 11.288 O.2 4 GLY4
29 N 43.406 23.993 10.767 N.pl3 5 THR5
30 CA 42.004 23.642 10.443 C.3 5 THR5
31 C 40.788 24.146 11.252 C.2 5 THR5
32 O 39.804 23.384 11.41 O.2 5 THR5
33 CB 41.934 22.202 9.889 C.3 5 THR5
```

34 OG1 41.08 21.317 10.609 O.3 5 THR5
35 CG2 43.317 21.556 9.849 C.3 5 THR5
36 N 40.628 25.463 11.441 N.pl3 6 PHE6
37 CA 39.381 25.95 12.104 C.3 6 PHE6
38 C 38.156 25.684 11.232 C.2 6 PHE6
39 O 37.231 25.002 11.719 O.2 6 PHE6
40 CB 39.407 27.425 12.584 C.3 6 PHE6
41 CG 38.187 27.923 13.43 C.ar 6 PHE6
42 CD1 36.889 27.518 13.163 C.ar 6 PHE6
43 CD2 38.386 28.862 14.419 C.ar 6 PHE6
44 CE1 35.813 27.967 13.909 C.ar 6 PHE6
45 CE2 37.306 29.328 15.177 C.ar 6 PHE6
46 CZ 36.019 28.871 14.928 C.ar 6 PHE6

@<TRIPOS>BOND
0 2 3 SINGLE
1 2 1 SINGLE
2 1 0 SINGLE
3 1 4 SINGLE
4 4 5 SINGLE
5 7 9 SINGLE
6 7 5 SINGLE
7 8 9 SINGLE
8 8 6 SINGLE
9 5 6 SINGLE
10 12 11 SINGLE
11 12 13 SINGLE
12 11 14 SINGLE
13 11 10 SINGLE
14 14 15 SINGLE
15 2 10 SINGLE
16 18 17 SINGLE
17 18 19 SINGLE
18 17 20 SINGLE
19 17 16 SINGLE
20 20 21 SINGLE
21 22 21 SINGLE
22 22 23 SINGLE
23 22 24 SINGLE
24 12 16 SINGLE
25 27 26 SINGLE
26 27 28 SINGLE

```
27 26 25 SINGLE
28 18 25 SINGLE
29 31 30 SINGLE
30 31 32 SINGLE
31 30 33 SINGLE
32 30 29 SINGLE
33 33 35 SINGLE
34 33 34 SINGLE
35 27 29 SINGLE
36 38 39 SINGLE
37 38 37 SINGLE
38 37 36 SINGLE
39 37 40 SINGLE
40 40 41 SINGLE
41 42 44 SINGLE
42 42 41 SINGLE
43 43 45 SINGLE
44 43 41 SINGLE
45 44 46 SINGLE
46 45 46 SINGLE
47 31 36 SINGLE


@<TRIPOS>SUBSTRUCTURE
1 HIS1 1 RESIDUE
2 SER2 11 RESIDUE
3 GLN3 17 RESIDUE
4 GLY4 26 RESIDUE
5 THR5 30 RESIDUE
6 PHE6 37 RESIDUE
```

```
import chimera

#    open up a molecule to work with:
opened = chimera.openModels.open('3fx2', type="PDB")
mol    = opened[0]

#    Molecule Display Properties
#    ~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#    the 'color' attribute represents the model-level color.
#    This color can be controlled by the midas command 'modelcolor'.
#    The 'color' assigned to a newly opened model is determined by a configurable preference (see discussion above).
#    Programmatically, the model
#    color can be changed by simply assigning a 'MaterialColor' to 'molecule.color'. Molecules also have a
#    'display' attribute, where a value of 'True' corresponds to being displayed, and a value of 'False'
#     means the molecule is not displayed.
#     So to make sure the molecule is shown (it is by default when first opened):
mol.display = True

#     To color the molecule red,
#     get a reference to Chimera's notion of the color red (returns a 'MaterialColor' object)
from chimera.colorTable import getColorByName
red = getColorByName('red')

#     and assign it to 'mol.color'.
mol.color = red
#     Note that the model will appear red at this point because all the atoms/bonds/residues
#     'color' attributes are set to 'None'

#    Atom Display Properties
#    ~~~~~~~~~~~~~~~~~~~~~~~
#
#    Each atom in a molecule has its own individual color,
#    accessible by the 'color' attribute. Upon opening a molecule, each atom's 'color' is set to 'None';
#    it can be changed by assigning a new 'MaterialColor' to 'atom.color'.
#    So, if we wanted to color all the alpha-carbon atoms blue, and all the rest yellow,
#    get references to the colors:
blue    = getColorByName('blue')
yellow  = getColorByName('yellow')

#    get a list of all the atoms in the molecule
ATOMS = mol.atoms
for at in ATOMS:
   #    check to see if this atom is an alpha-carbon
   if at.name == 'CA':
      at.color = yellow
   else:
      at.color = blue

#    Now, even though 'mol.color' is set to red, the molecule will appear to be blue and yellow. This is because each individual
#    atom's 'color' is visible over 'mol.color'.

#     Like molecules, atoms also have a 'display' attribute that controls whether or not the atom is shown.
#     While 'atom.display' controls whether the atom can be seen at all, 'atom.drawMode' controls its visual representation.
#     The value of 'drawMode' can be one of four constants, defined in the 'Atom' class.
#     Acceptable values for 'drawMode'
#     are 'chimera.Atom.Dot' (dot representation), 'chimera.Atom.Sphere' (sphere representation),
```

```
#    'chimera.Atom.EndCap' (endcap representation), or 'chimera.Atom.Ball' (ball representation).
#    So, to represent all the atoms in the molecule as "balls":
for at in ATOMS:
   at.drawMode = chimera.Atom.Ball



#   Bond Display Properties
#   ~~~~~~~~~~~~~~~~~~~~~~~
#
#   Bonds also contain 'color', and 'drawMode' attributes. They serve the same purposes here as they do
#   in atoms ('color' is the color specific to that bond, and 'drawMode' dictates
#   how the bond is represented). 'drawMode' for bonds can be either 'chimera.Bond.Wire' (wire representation)
#   or 'chimera.Bond.Stick' (stick representation).
#   The 'bond.display' attribute accepts slightly different values than that of other objects.
#   While other objects' 'display' can be set to either 'False' (not displayed)
#   or 'True' (displayed), 'bond.display' can be assigned a value of 'chimera.Bond.Never' (same as 'False' - bond is not
#   displayed), 'chimera.Bond.Always' (same as 'True' - bond is displayed), or 'chimera.Bond.Smart' which means that the
#   bond will only be
#   displayed if both the atoms it connects to are displayed. If not, the bond will not be displayed.
#   The heuristic that determines bond color is also a little more complicated than for atoms.
#   Bonds have an attribute called 'halfbond'
#   that determines the source of the bond's color. If 'halfbond' is set to 'True', then the
#   bond derives its color from the atoms which
#   it connects, and ignores whatever 'bond.color' is. If both those atoms are the same color (blue, for instance),
#   then the bond will appear blue. If the bonds atoms are different colors, then each half of the bond will correspond to the color
#   of the atom on that side. However, if 'bond.halfbond' is set to 'False', then that bond's color
#   will be be derived from its 'color' attribute, regardless of the 'color's of the atoms which it connects (except in the case
#   'bond.color' is 'None', the bond will derive its color from one of the atoms to which it connects).
#   To set each bond's display mode to "smart", represent it as a stick, and turn halfbond mode on,
#   get a list of all bonds in the molecule
BONDS = mol.bonds
for b in BONDS:
   b.display  = chimera.Bond.Smart
   b.drawMode = chimera.Bond.Stick
   b.halfbond = True



#   Residue Display Properties
#   ~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#   Residues are not "displayed" in the same manner that atoms and bonds are. When residues are displayed, they are
#   in the form of ribbons, and the attributes that control the visual details of the residues are named accordingly:
#   'ribbonDisplay', 'ribbonColor', 'ribbonDrawMode'. The values for 'ribbonDrawMode' can be 'chimera.Residue.Ribbon_2D' (flat ribbon),
#   'chimera.Residue.Ribbon_Edged' (sharp ribbon), or 'chimera.Residue.Ribbon_Round' (round/smooth ribbon).
#   If a residue's 'ribbonDisplay' value is set to 'False', it doesn't matter what 'ribbonDrawMode'
#   is - the ribbon still won't be displayed!
#   Residues have three attributes that control how the ribbon is drawn. 'isTurn', 'isHelix', and  'isSheet' (same as 'isStrand') are
#   set to either 'True' or 'False' based on secondary structure information contained in the source file (if available).
#   For any residue, only one of these can be set to 'True'.
#   So, to display only the residues which are part of an alpha-helix, as a smooth ribbon,
#   get a list of all the residues in the molecule
RESIDUES = mol.residues
for r in RESIDUES:
   #    only for residues that are part of an alpha-helix
   if r.isHelix:
      r.ribbonDisplay  = True
      r.ribbonDrawMode = chimera.Residue.Ribbon_Round
```

```
#    Import system modules used in this example.
import re

# Import Chimera modules used in this example.
import chimera

#    Define a regular expression for matching the names of protein
#    backbone atoms (we do not include the carbonyl oxygens because
#    they tend to clutter up the graphics display without adding
#    much information).
MAINCHAIN = re.compile("^(N|CA|C)$", re.I)

#    Do the actual work of setting the display status of atoms and
#    bonds.  The following 'for' statement iterates over molecules.
#    The function call
#    'chimera.openModels.list(modelTypes=[chimera.Molecule])'
#    returns a list of all open molecules; non-molecular models such
#    as surfaces and graphics objects will not appear in the list.
#    The loop variable 'm' refers to each model successively.
for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):

   #    The following 'for' statement iterates over atoms. The
   #    attribute reference 'm.atoms' returns a list of all atoms
   #    in model 'm', in no particular order.  The loop variable
   #    'a' refers to each atom successively.
   for a in m.atoms:
      #    Set the display status of atom 'a'.  First, we match
      #    the atom name, 'a.name', against the backbone atom
      #    name regular expression, 'MAINCHAIN'. The function
      #    call 'MAINCHAIN.match(a.name)' returns an 're.Match'
      #    object if the atom name matches the regular expression
      #    or 'None' otherwise.  The display status of the atom
      #    is set to true if there is a match (return value is not
      #    'None') and false otherwise.
      a.display = MAINCHAIN.match(a.name) != None

   #    By default, bonds are displayed if and only if both endpoint
   #    atoms are displayed, so therefore we don't have to explicitly
   #    set bond display modes; they will automatically "work right".
```

```
#   Function 'mainchain' sets the display status of atoms
#   and requires no arguments.  The body of the function is
#   identical to the example described in
#   "Molecular Editing Using Python".
#
#   .. "Molecular Editing Using Python" MolecularEditing.html
def mainchain():
   #   Note that due to a fairly arcane Python behavior, we need to
   #   import modules used by a (script) function inside the function itself
   #   (the local scope) rather than outside the function (the
   #   global scope).  This is because Chimera executes scripts in a
   #   temporary module so that names defined by the script don't
   #   conflict with those in Chimera's main namespace.  When the
   #   temporary module is deleted, Python sets all names in the
   #   module's global namespace to 'None'.  Therefore, by the time
   #   this function is executed (by the toolbar button callback)
   #   any modules imported in the global namespace would have the
   #   value 'None' instead of being a module object.

   #   The regular expression module, 're', is used for matching atom names.
   import re

   #   Import the object that tracks open models and the Molecule
   #   class from the 'chimera' module.
   from chimera import openModels, Molecule

   mainChain = re.compile("^(N|CA|C)$", re.I)
   for m in openModels.list(modelTypes=[Molecule]):
      for a in m.atoms:
         a.display = mainChain.match(a.name) != None

#   Need to import the 'chimera' module to access the function to
#   add the icon to the toolbar.
import chimera

#   Create a button in the toolbar.
#   The first argument to 'chimera.tkgui.app.toolbar.add' is the icon,
#   which is either the path to an image file, or the name of a standard
#   Chimera icon (which is the base name of an image file found in the
#   "share/chimera/images" directory in the Chimera installation directory).
#   In this case, since 'ToolbarButton.tiff' is not an absolute path, the
#   icon will be looked for under that name in both the current directory
```

```
#	and in the Chimera images directory.
#	The second argument is the Python function to be called when the button
#	is pressed (a.k.a., the "callback function").
#	The third argument is a short description of what the button does
#	(used typically as balloon help).
#	The fourth argument is the URL to a full description.
#	For this example the icon name is 'ToolbarButton.tiff';
#	the Python function is 'mainchain';
#	the short description is "Show Main Chain";
#	and there is no URL for context help.
chimera.tkgui.app.toolbar.add('ToolbarButton.tiff', mainchain, 'Show Main Chain', None)
```

Example *MolecularEditing.py*

Import system modules used in this example.

```
import re
```

Import Chimera modules used in this example.

```
import chimera
```

Define a regular expression for matching the names of protein backbone atoms (we do not include the carbonyl oxygens because they tend to clutter up the graphics display without adding much information).

```
MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
```

Do the actual work of setting the display status of atoms and bonds. The following `for` statement iterates over molecules. The function call `chimera.openModels.list(modelTypes=[chimera.Molecule])` returns a list of all open molecules; non-molecular models such as surfaces and graphics objects will not appear in the list. The loop variable `m` refers to each model successively.

```
for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
```

> The following `for` statement iterates over atoms. The attribute reference `m.atoms` returns a list of all atoms in model `m`, in no particular order. The loop variable `a` refers to each atom successively.
>
> ```
> for a in m.atoms:
> ```
>
> > Set the display status of atom `a`. First, we match the atom name, `a.name`, against the backbone atom name regular expression, `MAINCHAIN`. The function call `MAINCHAIN.match(a.name)` returns an `re.Match` object if the atom name matches the regular expression or `None` otherwise. The display status of the atom is set to true if there is a match (return value is not `None`) and false otherwise.
> >
> > ```
> > a.display = MAINCHAIN.match(a.name) != None
> > ```
>
> By default, bonds are displayed if and only if both endpoint atoms are displayed, so therefore we don't have to explicitly set bond display modes; they will automatically "work right".

```
#    The contents of "ToolbarButtonPackage/__init__.py" is
#    identical to the first section of code in "Toolbar Buttons".
#
#    .. "ToolbarButtonPackage/__init__.py" ToolbarButtonPackage/__init__.py
#    .. "Toolbar Buttons" ToolbarButton.html

def mainchain():
    import re
    from chimera import openModels, Molecule

    mainChain = re.compile("^(N|CA|C)$", re.I)
    for m in openModels.list(modelTypes=[Molecule]):
        for a in m.atoms:
            a.display = mainChain.match(a.name) != None
```

Example *ToolbarButton.py*

Function `mainchain` sets the display status of atoms and requires no arguments. The body of the function is identical to the example described in Molecular Editing Using Python.

```
def mainchain():
```

Note that due to a fairly arcane Python behavior, we need to import modules used by a (script) function inside the function itself (the local scope) rather than outside the function (the global scope). This is because Chimera executes scripts in a temporary module so that names defined by the script don't conflict with those in Chimera's main namespace. When the temporary module is deleted, Python sets all names in the module's global namespace to `None`. Therefore, by the time this function is executed (by the toolbar button callback) any modules imported in the global namespace would have the value `None` instead of being a module object.

The regular expression module, `re`, is used for matching atom names.

```
    import re
```

Import the object that tracks open models and the Molecule class from the `chimera` module.

```
    from chimera import openModels, Molecule

    mainChain = re.compile("^(N|CA|C)$", re.I)
    for m in openModels.list(modelTypes=[Molecule]):

        for a in m.atoms:
            a.display = mainChain.match(a.name) != None
```

Need to import the `chimera` module to access the function to add the icon to the toolbar.

```
import chimera
```

Create a button in the toolbar. The first argument to `chimera.tkgui.app.toolbar.add` is the icon, which is either the path to an image file, or the name of a standard Chimera icon (which is the base name of an image file found in the "share/chimera/images" directory in the Chimera installation directory). In this case, since `ToolbarButton.tiff` is not an absolute path, the icon will be looked for under that name in both the current directory and in the Chimera images directory. The second argument is the Python function to be called when the button is pressed (a.k.a., the "callback function"). The third argument is a short description of what the button does (used typically as balloon help). The fourth argument is the URL to a full description. For this example the icon name is `ToolbarButton.tiff`; the Python function is `mainchain`; the short description is "Show Main Chain"; and there is no URL for context help.

```
chimera.tkgui.app.toolbar.add('ToolbarButton.tiff', mainchain, 'Show Main Chain', None)
```

```
#   The contents of "ToolbarButtonPackage/gui.py" is similar to
#   the last section of code in "Toolbar Buttons", with the
#   exception that the 'mainchain' function is now referenced as
#   'ToolbarButtonPackage.mainchain'. The reason for the change is
#   that 'gui.py' is a submodule, while the 'mainchain' function is in
#   the main package.  Since a submodule cannot directly access items
#   defined in the main package, 'gui.py' must first import the package
#   'import ToolbarButtonPackage' and refer to the function by prepending
#   the package name ('ToolbarButtonPackage.mainchain' in the call to
#   'chimera.tkgui.app.toolbar.add').
#
#   .. "ToolbarButtonPackage/gui.py" ToolbarButtonPackage/gui.py
#   .. "Toolbar Buttons" ToolbarButton.html

import chimera
import ToolbarButtonPackage
chimera.tkgui.app.toolbar.add('ToolbarButton.tiff', ToolbarButtonPackage.mainchain, 'Show Main Chain', None)
```

```
#    The contents of "ToolbarButtonExtension/__init__.py" is
#    identical to the first section of code in "Toolbar Buttons",
#    with the exception that module 'os' is not imported.
#
#    .. "ToolbarButtonExtension/__init__.py" ToolbarButtonExtension/__init__.py
#    .. "Toolbar Buttons" ToolbarButton.html

import re

import chimera

def mainchain():
   MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
   for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
      for a in m.atoms:
         a.display = MAINCHAIN.match(a.name) != None
```

```
#   "ChimeraExtension.py" derives a class from 'chimera.extension.EMO'
#   to define how functionality defined in "__init__.py" may be invoked
#   by the Chimera extension manager.
#
#   .. "ChimeraExtension.py" ToolbarButtonExtension/ChimeraExtension.py
#   .. "__init__.py" ToolbarButtonExtension/__init__.py

import chimera.extension

#   Class 'MainChainEMO' is the derived class.
class MainChainEMO(chimera.extension.EMO):
   # Return the actual name of the extension.
   def name(self):
      return 'Backbone'

   # Return the short description that typically appears as
   # balloon help or in the Tools preference category.
   def description(self):
      return 'display only protein backbone'

   # Return the categories in which this extension should appear.
   # It is either a list or a dictionary.  If it is a dictionary
   # then the keys are the category names and the values are
   # category-specific descriptions (and the description() method
   # above is ignored).
   def categories(self):
      return ['Utilities']

   # Return the name of a file containing an icon that may be used
   # on the tool bar to provide a shortcut for launching the extension.
   def icon(self):
      return self.path('mainchain.tiff')

   # Invoke the extension.  Note that when this method is called,
   # the content of "__init__.py" is not available.  To simplify
   # calling functions, the 'EMO' provides a 'module' method that
   # locates modules in the extension package by name; if no name
   # is supplied, the "__init__.py" module is returned.
   def activate(self):
      # Call the 'mainchain' function in the "__init__.py" module.
      self.module().mainchain()
```

```
# Register an instance of 'MainChainEMO' with the Chimera
# extension manager.
chimera.extension.manager.registerExtension(MainChainEMO(__file__))
```

```
# The initial code is the same as for the ToolbarButtonExtension example
import chimera.extension

class MainChainEMO(chimera.extension.EMO):
    def name(self):
        return 'Backbone'

    def description(self):
        return 'display only protein backbone'

    def categories(self):
        return ['Utilities']

    def icon(self):
        return self.path('mainchain.tiff')

    def activate(self):
        self.module().mainchain()

chimera.extension.manager.registerExtension(MainChainEMO(__file__))

# Here we define two functions, one to handle the "mainchain" command,
# and one to handle the "~mainchain" command.
def mainchainCmd(cmdName, args):
        # Import the module's workhorse function.
        # It is imported inside the function definition so that
        # it does not slow down Chimera startup with extra imports
        # in the main scope.
        from ToolbarButtonCommand import mainchain

        # Import and use the Midas.midas_text doExtensionFunc procedure
        # to process typed arguments and call the mainchain() function
        # appropriately.  For a simple function like mainchain(), which
        # takes no arguments, using doExtensionFunc is probably overkill.
        # One could instead use the approach applied in the revMainchainCmd
        # function below and simply test for the presence of any
        # arguments (raising MidasError if any are found) and directly calling
        # the mainchain() function otherwise.  As implemented here, using
        # doExtensionFunc, if the user does provide arguments then
        # doExtensionFunc will raise an error complaining that there
        # were unknown keyword arguments supplied.
        from Midas.midas_text import doExtensionFunc
```

```
    doExtensionFunc(mainchain, args)

# The function for "~mainchain"
def revMainchainCmd(cmdName, args):
    # We are going to implement ~mainchain as a synonym for "display",
    # so we import runCommand which simplifies doing that.
    from chimera import runCommand
    from Midas import MidasError
    if args:
        # Raising MidasError will cause the error message
        # to show up in the status line as red text
        raise MidasError("~mainchain takes no arguments")
    # runCommand takes any legal command-line command and executes it.
    runCommand("display")

# Now actually register the "mainchain" command with the command interpreter
# by using addCommand().  The first argument is the command name and the
# second is the callback function for doing the work.  The 'revFunc' keyword
# specifies the function to implement "~mainchain".  The 'help' keyword has
# been omitted, therefore no help will be provided.
from Midas.midas_text import addCommand
addCommand("mainchain", mainchainCmd, revFunc=revMainchainCmd)
```

```
#    Import the standard modules used in this example.
import re

#    Import the Chimera modules used in this example.
import chimera

#    Define a regular expression for matching the names of protein backbone
#    atoms (we do not include the carbonyl oxygens because they tend to
#    clutter up the graphics display without adding much information).
MAINCHAIN = re.compile("^(N|CA|C)$", re.I)

#    Define 'mainchain' function for setting the display representation
#    of protein backbone atoms and bonds.  See "Molecular Editing" for a
#    more detailed example on changing molecular attributes.
#
#    .. "Molecular Editing" MolecularEditing.html
def mainchain(atomMode, bondMode):
   for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
      for a in m.atoms:
         if MAINCHAIN.match(a.name):
            a.drawMode = atomMode
      for b in m.bonds:
         ends = b.atoms
         if MAINCHAIN.match(ends[0].name) \
         and MAINCHAIN.match(ends[1].name):
            b.drawMode = bondMode
```

```
#    Import the standard modules used by this example.
import os
import Tkinter

#    Import the Chimera modules and classes used by this example.
import chimera
from chimera.baseDialog import ModelessDialog

#    Import the package for which the graphical user interface
#    is designed.  In this case, the package is named 'ExtensionUI'.
import ExtensionUI

#     Define two module variables:
#    'atomMode' and 'bondMode' are Tk variables that keep track of
#    the last selected display representations. These variables are
#    initialized to be 'None', and are set to usable values when
#    the GUI is created.
atomMode = None
bondMode = None

#    Define two dictionaries that map string names into Chimera
#    enumerated constant values. The two variables 'atomMode' and
#    'bondMode' keep track of the representations as strings because
#    they are displayed directly in the user interface. However,
#    the 'mainchain' function in the main package expects Chimera
#    constants as its arguments. The dictionaries 'atomModeMap' and
#    'bondModeMap' provides the translation from string to enumerated
#    constants.
atomModeMap = {
   'Dot': chimera.Atom.Dot,
   'Sphere': chimera.Atom.Sphere,
   'EndCap': chimera.Atom.EndCap,
   'Ball': chimera.Atom.Ball
}
bondModeMap = {
   'Wire': chimera.Bond.Wire,
   'Stick': chimera.Bond.Stick
}

#    Chimera offers two base classes to somewhat simplify the task of
#    creating user interfaces: ModalDialog and ModelessDialog.  The
#    former is designed for situations when information or response
#    is required of the user immediately; the dialog stays in front
#    of other Chimera windows until dismissed and prevents input from
#    going to other Chimera windows.  The latter dialog type is designed
#    for "ongoing" interfaces; it allows input focus to go to other
#    windows, and other windows can obscure it.
#
#    Here we declare a class that derives from ModelessDialog and
#    customize it for the specific needs of this extension.
class MainchainDialog(ModelessDialog):

   #    Chimera dialogs can either be *named* or *nameless*.  Named
   #    dialogs are displayed using the 'display(name)' function
   #    of the chimera.dialogs module.  The *name* that should be used
   #    as an argument to the 'display' function is given by the class
   #    variable 'name'.  Using a named dialog is appropriate when
   #    it might be desirable to invoke the dialog from other extensions
   #    or from Chimera itself.
```

```
#
#    A nameless dialog is intended for use only in the extension that
#    defines the dialog.  A nameless dialog is typically created and
#    displayed by calling its constructor.  Once created, a nameless
#    dialog can be redisplayed (if it was withdrawn by clicking its
#    'Cancel' button for example) by calling its 'enter()' method.
#
#    For demonstration purposes, we use a named dialog here.  A
#    nameless dialog is used in the "Color and Color Wells" example.
#
#    .. "Color and Color Wells" Main_ColorWellUI.html
name = "extension ui"

#    The buttons displayed at the bottom of the dialog are given
#    in the class variable 'buttons'.  For modeless dialogs, a
#    help button will automatically be added to the button list
#    (the help button will be grayed out if no help information
#    is provided).  For buttons other than 'Help', clicking on
#    them will invoke a class method of the same name.
#
#    Both dialog base classes provide appropriate methods for
#    'Close', so we won't provide a 'Close' method in this
#    subclass.  The ModelessDialog base class also provides a
#    stub method for 'Apply', but we will override it with our
#    own 'Apply' method later in the class definition.
buttons = ("Apply", "Close")

#    A help file or URL can be specified with the 'help' class
#    variable.  A URL would be specified as a string (typically
#    starting with "http://...").  A file would be specified as
#    a 2-tuple of file name followed by a package.  The file
#    would then be looked for in the 'helpdir' subdirectory of
#    the package.  A dialog of Chimera itself (rather than of an
#    imported package) might only give a filename as the class
#    help variable.  That file would be looked for in
#    /usr/local/chimera/share/chimera/helpdir.
help = ("ExtensionUI.html", ExtensionUI)

#    The title displayed in the dialog window's title bar is set
#    via the class variable 'title'.
title = "Set Backbone Representation"

#    Both ModelessDialog and ModalDialog, in their __init__
#    functions, set up the standard parts of the dialog interface
#    (top-level window, bottom-row buttons, etc.) and then call
#    a function named 'fillInUI' so that the subclass can fill
#    in the parts of the interface specific to the dialog.  As
#    an argument to the function, a Tkinter Frame is provided
#    that should be the parent to the subclass-provided interface
#    elements.
def fillInUI(self, parent):

    #    Declare that, in 'fillInUI', the names 'atomMode' and
    #    'bondMode' refer to the global variables defined above.
    global atomMode, bondMode

    #    Create and initialize 'atomMode' and 'bondMode', the
    #    two  global Tk string variables that keep track of the
    #    currently selected display representation.
    atomMode = Tkinter.StringVar(parent)
```

```
        atomMode.set('Dot')
        bondMode = Tkinter.StringVar(parent)
        bondMode.set('Wire')

        #    Create the label and option menu for selecting atom
        #    display representation. First create the label 'Atom
        #    Representation' and place it on the left-hand side of
        #    the top row in the GUI window.
        atomLabel = Tkinter.Label(parent, text='Atom Representation')
        atomLabel.grid(column=0, row=0)
        #    Create the menu button and the option menu that it brings up.
        atomButton = Tkinter.Menubutton(parent, indicatoron=1,
                                textvariable=atomMode, width=6,
                                relief=Tkinter.RAISED, borderwidth=2)
        atomButton.grid(column=1, row=0)
        atomMenu = Tkinter.Menu(atomButton, tearoff=0)
        #    Add radio buttons for all possible choices to the menu.
        #    The list of choices is obtained from the keys of the
        #    string-to-enumeration dictionary, and the radio button
        #    for each choice is programmed to update the 'atomMode'
        #    variable when selected.
        for mode in atomModeMap.keys():
           atomMenu.add_radiobutton(label=mode, variable=atomMode, value=mode)
        #    Assigns the option menu to the menu button.
        atomButton['menu'] = atomMenu

        #    The lines below do the same thing for bond representation
        #    as the lines above do for atom representation.
        bondLabel = Tkinter.Label(parent, text='Bond Representation')
        bondLabel.grid(column=0, row=1)
        bondButton = Tkinter.Menubutton(parent, indicatoron=1,
                                textvariable=bondMode, width=6,
                                relief=Tkinter.RAISED, borderwidth=2)
        bondButton.grid(column=1, row=1)
        bondMenu = Tkinter.Menu(bondButton, tearoff=0)
        for mode in bondModeMap.keys():
           bondMenu.add_radiobutton(label=mode, variable=bondMode, value=mode)
        bondButton['menu'] = bondMenu

    #    Define the method that is invoked when the 'Apply' button
    #    is clicked. The function simply converts the currently
    #    selected representations from strings to enumerated constants,
    #    using the 'atomModeMap' and 'bondModeMap' dictionaries, and
    #    invokes the main package function 'mainchain'.
    def Apply(self):
        ExtensionUI.mainchain(atomModeMap[atomMode.get()],
                              bondModeMap[bondMode.get()])

#   Now we register the above dialog with Chimera, so that it may be
#   invoked via the 'display(name)' method of the chimera.dialogs module.
#   Here the class itself is registered, but since it is a named dialog
#   deriving from ModalDialog/ModelessDialog, the instance will automatically
#   reregister itself when first created.  This allows the 'dialogs.find()'
#   function to return the instance instead of the class.
chimera.dialogs.register(MainchainDialog.name, MainchainDialog)

#    Create the Chimera toolbar button that displays the dialog when
#    pressed.  Note that since the package is not normally searched for
#    icons, we have to prepend the path of this package to the icon's
#    file name.
```

```
dir, file = os.path.split(__file__)
icon = os.path.join(dir, 'ExtensionUI.tiff')
chimera.tkgui.app.toolbar.add(icon, lambda d=chimera.dialogs.display, n=MainchainDialog.name: d(n), 'Set Main Chain Representation', None)
```

RGBA tuples

RGBA tuples are 4-tuples where the respective tuple components represent red, green, blue, and alpha (opacity) values for a color. Each value is a floating point number between 0.0 and 1.0. For example, the tuple (1, 0, 0, 1) represents an opaque red, while (0, 1, 0, 0.5) represents a half transparent green.

Back to example.

```
#    This code is analogous to the code found in the "__init__.py"
#    modules in the "Packaging an Extension" and "Extension-Specific
#    User Interface" examples.  See "Molecular Editing" for a more
#    detailed example on changing molecular attributes. Note that
#    the 'mainchain' function is expecting a color *object* as its
#    argument (because the color is used to set an atomic attribute).
#
#    .. "Packaging an Extension" Main_ExtensionPackage.html
#    .. "Extension-Specific User Interface" Main_ExtensionUI.html
#    .. "Molecular Editing" MolecularEditing.html
import chimera
import re

MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
def mainchain(color):
   for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
      for a in m.atoms:
         if MAINCHAIN.match(a.name):
            a.color = color
```

```
#    Import the standard Python modules used by the example code.
import os
import Tkinter

#    Import the additional modules and classes needed.
#    The ColorOption class facilitates interoperation between Chimera
#    colors and color wells (which use rgba colors).
import chimera
from chimera.baseDialog import ModelessDialog
from chimera.tkoptions import ColorOption
import ColorWellUI

class ColorWellDialog(ModelessDialog):
    #    ColorWellDialog is a "nameless" dialog.  See the
    #    "Extension-Specific User Interface" example for a more detailed
    #    explanation of Chimera dialogs.
    #
    #    ..  "Extension-Specific User Interface"  Main_ExtensionUI.html

    #    Set the title bar of the dialog to display 'Set Backbone Color'.
    title = 'Set Backbone Color'

    def fillInUI(self, master):
        #    Create a ColorOption instance.  The ColorOption will contain
        #    a descriptive label and a color well.  The arguments to the
        #    ColorOption constructor are:
        #     - master widget
        #     - row number to use when 'grid'ing the ColorOption into the
        #       master widget.  The default column is 0.  The tkoptions
        #       module contains other options besides ColorOption (e.g.
        #       StringOption), which are generally intended to be put in
        #       vertical lists, and therefore a row number is specified in
        #       the constructor.  In this example we are only using one
        #       option however.
        #     - option label.  This will be positioned to the left of the
        #       color well and a ":" will be appended.
        #     - The default value for this option.
        #     - A callback function to call when the option is set by the
        #       user (e.g. a color dragged to the well, or the well color
        #       edited in the color editor)
        #     - An optional ballon-help message
        #
        coloropt = ColorOption(master, 0, 'Backbone Color', None, self._setBackboneColor, balloon='Protein backbone color')

        #    Call '_updateBackboneColor' to make the color displayed
        #    in the color well reflect the current color of protein
        #    backbone atoms. While not strictly necessary, keeping the
        #    color in the well consistent with the color in the molecules
        #    enhances the extension usability.
        self._updateBackboneColor(coloropt)
```

```
  #   Define '_updateBackboneColor', which is used to make the color
  #   of a well reflect the color of protein backbone atoms.
  def _updateBackboneColor(self, coloroption):
     #   Loop through all atoms in all molecules, looking for protein
     #   backbone atoms. When one is found, its color is compared
     #   against the last color seen, 'theColor'. The first time this
     #   comparison is made, 'theColor' does not exist yet and a
     #   NameError exception is raised; this exception is caught,
     #   and 'theColor' is assigned the color of the atom.  All
     #   subsequent times, the comparison between atom color and
     #   'theColor' should work as expected. If the two colors are
     #   different, the color well is set to show that multiple colors
     #   are present and execution returns to the caller.  If the two
     #   colors are the same, the next atom is examined. If only one
     #   color is found among all protein backbone atoms (or zero if
     #   no molecules are present), then execution continues.
     for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
        for a in m.atoms:
           if ColorWellUI.MAINCHAIN.match(a.name):
              try:
                 if a.color != theColor:
                    coloroption.setMultiple()
                    return
              except NameError:
                 theColor = a.color

     #   Set the color of the well to match 'theColor'.  There are
     #   two possible cases:
     #     1 'theColor' is not set (because there are no molecules),
     #     2 'theColor' is 'None' or a color object.
     #   The 'set' function will not result in a callback to
     #   '_setBackboneColor'.
     try:
        #   Handle case 2.  Set the color well to the proper color
        coloroption.set(theColor)
     except NameError:
        #   Handle case 1.  Set the color well to show that no color
        #   is present
        coloroption.set(None)

  #   Define '_setBackboneColor', which is invoked each time the
  #   color in the well changes.  When called by the ColorOption,
  #   '_setBackboneColor' receives a single argument 'coloropt',
  #   which is the ColorOption instance.
  def _setBackboneColor(self, coloroption):
     #   Call the 'mainchain' function in the main package, with
     #   the color object corresponding to the color well color
     #   as its argument (which will be None if 'No Color' is
     #   the current selection in the well), to set the color of
     #   backbone atoms.
     ColorWellUI.mainchain(coloroption.get())
```

```
#    Define the module variable 'dialog', which keeps track of the
#    dialog window containing the color well. It is initialized to
#    'None', and is assigned a usable value when the dialog is created.
dialog = None

#    Define 'showColorWellUI', which is invoked when the Chimera
#    toolbar button is pressed.
def showColorWellUI():
   #    Declare that the name 'dialog' refers to the global variable
   #    defined above.
   global dialog
   #    Check whether the dialog has already been created. If so,
   #    the dialog window is displayed by calling it's 'enter()'
   #    function, and then the rest of the function is skipped by returning.
   if dialog is not None:
      dialog.enter()
      return

   #   Otherwise, create the dialog.
   dialog = ColorWellDialog()

#    Create the Chimera toolbar button that invokes the 'showColorWellUI'
dir, file = os.path.split(__file__)
icon = os.path.join(dir, 'ColorWellUI.tiff')
chimera.tkgui.app.toolbar.add(icon, showColorWellUI, 'Set Main Chain Color', None)
```

```
#    This file is identical to the "ColorWellUI/__init__.py"
#    in the "Colors and Color Wells" example.
#
#    .. "ColorWellUI/__init__.py" ColorWellUI/__init__.py
#    .. "Colors and Color Wells" Main_ColorWellUI.html
import chimera
import re

MAINCHAIN = re.compile("^(N|CA|C)$", re.I)
def mainchain(color):
   for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
      for a in m.atoms:
         if MAINCHAIN.match(a.name):
            a.color = color
```

```
#   The code here is very similar to the code in "Colors
#   and Color Wells" and only differences from that code
#   will be described.
#
#   .. "Colors and Color Wells" Main_ColorWellUI.html

import os
import Tkinter

import chimera
from chimera.baseDialog import ModelessDialog
from chimera.tkoptions import ColorOption
import ColorWellUI

class ColorWellDialog(ModelessDialog):

   title = 'Set Backbone Color'

   #   Need to override '__init__' to initialize our extra state.
   def __init__(self, *args, **kw):

      #   Whereas in the "Colors and Color Wells" example 'coloropt'
      #   was a local variable, here the 'coloropt' variable is stored
      #   in the instance because the trigger handler (which has access
      #   to the instance) needs to update the color well contained in
      #   the ColorOption.  A new variable, 'handlerId', is created to
      #   keep track of whether a handler is currently registered.  The
      #   handler is only created when needed.  See 'map' and 'unmap'
      #   below.  (Note that the instance variables must be set before
      #   calling the base __init__ method since the dialog may be mapped
      #   during initialization, depending on which window system is used.)
      #
      #   .. "Colors and Color Wells" Main_ColorWellUI.html
      self.colorOpt = None
      self.handlerId = None

      #   Call the parent-class '__init__'.
      apply(ModelessDialog.__init__, (self,) + args, kw)

   def fillInUI(self, master):
      #   Save ColorOption in instance.
      self.coloropt = ColorOption(master, 0, 'Backbone Color', None, self._setBackboneColor, balloon='Protein backbone color')

      self._updateBackboneColor()

   def _updateBackboneColor(self):
      for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
         for a in m.atoms:
            if ColorWellUI.MAINCHAIN.match(a.name):
               try:
                  if a.color != theColor:
                     self.coloropt.setMultiple()
```

```
                return
            except NameError:
                theColor = a.color

        try:
            self.coloropt.set(theColor)
        except NameError:
            self.coloropt.set(None)

    def _setBackboneColor(self, coloroption):
        ColorWellUI.mainchain(coloroption.get())

    #   Register a trigger handler to monitor changes in the
    #   backbone atom list when we're make visible.  We ignore
    #   the event argument.
    def map(self, *ignore):
        #   Synchronize with well color.
        self._updateBackboneColor()

        #   If no handler is currently registered, register one.
        if self.handlerId is None:
            #   Registration occurs when the 'chimera.triggers' object
            #   is requested to add a handler. *Registration requires
            #   three arguments*:
            #     - the name of the trigger,
            #     - the handler function to be invoked when the
            #       trigger fires, and
            #     - an additional argument to be passed to the handler
            #       function when it is invoked.
            #   In this case, the trigger name is the same as the name
            #   of the class of objects being monitored, "Atom".
            #   The handler function is '_handler', defined below.
            #   And the additional argument is empty (None) -- it could
            #   have been the ColorOption instance ('coloropt') but that
            #   is accessible via the instance.  The return value from
            #   the registration is a unique handler identifier for
            #   the handler/argument combination.  This identifier is
            #   required for deregistering the handler.
            #
            #   *The handler function is always invoked by the trigger
            #   with three arguments*:
            #     - the name of the trigger,
            #     - the additional argument passed in at registration
            #       time, and
            #     - an instance with three attributes
            #      - created: set of created objects
            #      - deleted: set of deleted objects
            #      - modified: set of modified objects
            #   Note that with a newly opened model, objects will just
            #   appear in both the 'created' set and not in the 'modified'
            #   set, even though the newly created objects will normally have
            #   various of their default attributes modified by later
            #   code sections.
            self.handlerId = chimera.triggers.addHandler('Atom', self._handler, None)
```

```
    #    The '_handler' function is the trigger handler invoked when
    #    attributes of 'Atom' instances change.
    def _handler(self, trigger, additional, atomChanges):
        #    Check through modified atoms for backbone atoms.
        for a in atomChanges.modified:
            #    If any of the changed atoms is a backbone atom, call
            #    '_updateBackboneColor' to synchronize the well color
            #     with backbone atom colors.
            if ColorWellUI.MAINCHAIN.match(a.name):
                self._updateBackboneColor()
                return

    #    'unmap' is called when the dialog disappears.  We ignore the
    #    event argument.
    def unmap(self, *ignore):
        #    Check whether a handler is currently registered (*i.e.*, the
        #    handler identifier, 'handlerId', is not 'None') and
        #    deregister it if necessary.
        if self.handlerId is not None:

            #    Deregistration requires two arguments: the name of the
            #    trigger and the unique handler identifier returned by
            #    the registration call.
            chimera.triggers.deleteHandler('Atom', self.handlerId)

            #    Set the unique handler identifier to 'None' to indicate
            #    that no handler is currently registered.
            self.handlerId = None

#    Define the module variable 'dialog', which tracks the dialog instance.
#    It is initialized to 'None', and is assigned a usable value when the
#    dialog is created.
dialog = None

#    Define 'showColorWellUI', which is invoked when the Chimera
#    toolbar button is pressed.
def showColorWellUI():
    global dialog
    if dialog is not None:
        dialog.enter()
        return

    dialog = ColorWellDialog()

dir, file = os.path.split(__file__)
icon = os.path.join(dir, 'AtomTrigger.tiff')
chimera.tkgui.app.toolbar.add(icon, showColorWellUI, 'Set Main Chain Color', None)
```

OSL

OSL stands for Object Selection Language. The OSL is a syntax for choosing Selectables. For example, ":51@ca" is an OSL string that refers to residue 51, atom CA of all molecules. "@/color=red" refers to all red vertices (usually atoms) of all models. Further information about the OSL can be found here.

Object Selection Language syntax

The OSL is a syntax for choosing Selectables. There are four generic types of Selectables: graphs, subgraphs, vertices, and edges. In a molecular model, these correspond to the entire molecule, the residues, atoms, and bonds. For a non-molecular model, the only relevant Selectable type, graph, corresponds to the entire model.

The OSL is almost exactly the same as the Midas atom specification syntax as implemented in the Chimera Midas emulator. A detailed description of that syntax can be found here. There are three features found in Midas atom specifiers that are not found in the OSL. They are:

1. Current selection synonyms ("selected", "sel", ***etc*** .).
2. Zones (***e.g.***"z<5")
3. Intersections (the & symbol)

```
#    Import the standard modules used in this example.
import re

#    Import the Chimera modules used in this example.
import chimera
from chimera import selection

#    Define a function that will select protein backbone atoms in the
#    main Chimera graphics window
def selBackbone(op=None):
   #    Define a regular expression for matching the names of protein backbone
   #    atoms (we do not include the carbonyl oxygens because they tend to
   #    clutter up the graphics display without adding much information).
   MAINCHAIN = re.compile("^(N|CA|C)$", re.I)

   #    The 'list' method of chimera.openModels will return a list of
   #    currently open models, and takes several optional keyword arguments
   #    to restrict this list to models matching certain criteria.
   #    When called with no arguments, this method will
   #    return a list of all visible models, essentially models that
   #    were created by the user. Internally managed ('hidden') models,
   #    such as the distance monitor pseudobondgroup, do not show up in this
   #    list. A list of hidden models can be obtained by setting the
   #    optional keyword argument 'hidden' to True.
   #    The 'all' argument (True/False) can be used to return a list of all open models
   #    (including both hidden and visible). Other optional arguments include:
   #   'id' and 'subid', which restrict the returned list to models with the given
   #    id and subid, respectively, while 'modelTypes' (a list of model types,
   #    i.e. '[chimera.Molecule]') will restrict the returned list to models
   #    of a particular type.
   bbAtoms = []
   for m in chimera.openModels.list(modelTypes=[chimera.Molecule]):
      for a in m.atoms:
         if MAINCHAIN.match(a.name):
            bbAtoms.append(a)

   #    Create a selection instance that we can use to hold the protein
   #    backbone atoms.  We could have added the atoms one by one to the
   #    selection while we were in the above loop, but it is more efficient
   #    to add items in bulk to selections if possible.
   backboneSel = selection.ItemizedSelection()
   backboneSel.add(bbAtoms)
```

```
    #    Add the connecting bonds to the selection.  The 'addImplied' method
    #    of Selection adds bonds if both bond endpoint atoms are in the
    #    selection, and adds atoms if any of the atom's bonds are in the
    #    selection.  We use that method here to add the connecting bonds.
    backboneSel.addImplied()

    #    Change the selection in the main Chimera window in the manner
    #    indicated by this function's 'op' keyword argument. If op is
    #    'None', then use whatever method is indicated by the 'Selection Mode'
    #    item in Chimera's Select menu.  Otherwise, op should
    #    be one of: 'selection.REPLACE', 'selection.INTERSECT',
    #    'selection.EXTEND' or 'selection.REMOVE'.
    #      - 'REPLACE' causes the Chimera selection to be replaced with
    #        'backboneSel'.
    #      - 'INTERSECT' causes the Chimera selecion to be intersected
    #        with 'backboneSel'.
    #      - 'EXTEND' causes 'backboneSel' to be appended to the Chimera
    #        selection.
    #      - 'REMOVE' causes 'backboneSel' to be unselected in the
    #        Chimera window.
    if op is None:
       chimera.tkgui.selectionOperation(backboneSel)
    else:
       selection.mergeCurrent(op, backboneSel)
```

The Selection Manager

Some extensions may identify sets of Selectables that it would be useful for other extensions to be able to identify, or for the user to select from the Selection menu. For example, the ChemGroup extension identifies chemical entities such as aromatic rings. It is useful to allow other extensions (and the user) to also pick out these entities without replicating the ChemGroup code. This is accomplished by registering selections with the Chimera selection manager.

The Chimera selection manager is defined in `chimera.selection.manager`. Selections are registered with the selection manager as strings (either Python code or OSL strings) and are made available in the Chimera Selection menu. Selections can (only) be retrieved from the selection manager en masse as a dictionary organized in the same fashion as the Selection menu. The keys in the dictionary are the same as the menu labels, and the values are either a 2-tuple of the registered selection string and a help description, or a dictionary introducing a submenu which is organized the same as the main dictionary. The `selectionFromText` method of the selection manager converts a registered selection string into the appropriate Selection subclass.

`chimera.selection.manager` also defines the **CodeItemizedSelection** class, which is similar to the **CodeSelection** class described above, except that the code is given an empty ItemizedSelection to fill in instead of functions to apply.

The Python strings registered with the selection manager are expected to be useable in conjunction with **CodeItemizedSelection**s.

This example is not yet finished. Sorry!

```
#    Class 'CountAtoms' assigns two attributes, "numAtoms" and "numHetatms",
#    to a molecule by exporting the molecule as a PDB file and running
#    the "grep" program twice.  The "grep" invocations are run in the
#    background so that Chimera stays interactive while they execute.
class CountAtoms:

   #    The constructor sets up a temporary file for the PDB output,
   #    and a Chimera task instance for showing progress to the user.
   def __init__(self, m, grepPath):

      #    Generate a temporary file name for PDB file.
      #    We use Chimera's 'osTemporaryFile' function
      #    because it automatically deletes the file when
      #    Chimera exits.
      import OpenSave
      self.pdbFile = OpenSave.osTemporaryFile(suffix=".pdb", prefix="rg")
      self.outFile = OpenSave.osTemporaryFile(suffix=".out", prefix="rg")

      #    Write molecule in to temporary file in PDB format.
      self.molecule = m
      import Midas
      Midas.write([m], None, self.pdbFile)

      #    Set up a task instance for showing user our status.
      from chimera import tasks
      self.task = tasks.Task("atom count for %s" % m.name, self.cancelCB)

      #    Start by counting the ATOM records first.
      self.countAtoms()

   #    'cancelCB' is called when user cancels via the task panel
   def cancelCB(self):
      self.molecule = None

   #    'countAtoms' uses "grep" to count the number of ATOM records.
   def countAtoms(self):
      from chimera import SubprocessMonitor as SM
      self.outF = open(self.outFile, "w")
      self.subproc = SM.Popen([ grepPath, "-c", "^ATOM", self.pdbFile ], stdout=self.outF)
      SM.monitor("count ATOMs", self.subproc, task=self.task, afterCB=self._countAtomsCB)

   #    '_countAtomsCB' is the callback invoked when the subprocess
```

```
    #    started by 'countAtoms' completes.
    def _countAtomsCB(self, aborted):

        #    Always close the open file created earlier
        self.outF.close()

        #    If user canceled the task, do not continue processing.
        if aborted or self.molecule is None:
            self.finished()
            return

        #    Make sure the process exited normally.
        if self.subproc.returncode != 0 and self.subproc.returncode != 1:
            self.task.updateStatus("ATOM count failed")
            self.finished()
            return

        #    Process exited normally, so the count is in the output file.
        #    The error checking code (in case the output is not a number)
        #    is omitted to keep this example simple.
        f = open(self.outFile)
        data = f.read()
        f.close()
        self.molecule.numAtoms = int(data)

        #    Start counting the HETATM records
        self.countHetatms()

    #    'countHetatms' uses "grep" to count the number of HETATM records.
    def countHetatms(self):
        from chimera import SubprocessMonitor as SM
        self.outF = open(self.outFile, "w")
        self.subproc = SM.Popen([ grepPath, "-c", "^HETATM", self.pdbFile ], stdout=self.outF)
        SM.monitor("count HETATMs", self.subproc, task=self.task, afterCB=self._countHetatmsCB)

    #    '_countHetatmsCB' is the callback invoked when the subprocess
    #    started by 'countHetatms' completes.
    def _countHetatmsCB(self, aborted):

        #    Always close the open file created earlier
        self.outF.close()

        #    If user canceled the task, do not continue processing.
```

```
        if aborted or self.molecule is None:
            self.finished()
            return

        #    Make sure the process exited normally.
        if self.subproc.returncode != 0 and self.subproc.returncode != 1:
            self.task.updateStatus("HETATM count failed")
            self.finished()
            return

        #    Process exited normally, so the count is in the output file.
        #    The error checking code (in case the output is not a number)
        #    is omitted to keep this example simple.
        f = open(self.outFile)
        data = f.read()
        f.close()
        self.molecule.numHetatms = int(data)

        #    No more processing needs to be done.
        self.finished()

    #    'finished' is called to clean house.
    def finished(self):

        #    Temporary files will be removed when Chimera exits, but
        #    may be removed here to minimize their lifetime on disk.
        #    The task instance must be notified so that it is labeled
        #    completed in the task panel.
        self.task.finished()

        #    Set instance variables to None to release references.
        self.task = None
        self.molecule = None
        self.subproc = None

#    Below is the main program.  First, we find the path to
#    the "grep" program.  Then, we run CountAtoms for each molecule.
from CGLutil import findExecutable
grepPath = findExecutable.findExecutable("grep")
if grepPath is None:
    from chimera import NonChimeraError
    raise NonChimeraError("Cannot find path to grep")
```

```
#    Add "numAtoms" and "numHetatms" attributes to all open molecules.
import chimera
from chimera import Molecule
for m in chimera.openModels.list(modelTypes=[Molecule]):
    CountAtoms(m, grepPath)
```

```
#    Function 'createWater' creates a water molecule.
def createWater():

    #    Import the object that tracks open models and several
    #    classes from the 'chimera' module.
    from chimera import openModels, Molecule, Element, Coord

    #    Create an instance of a Molecule
    m = Molecule()

    #    Molecule contains residues.  For our example, we will
    #    create a single residue of HOH.  The four arguments are:
    #    the residue type, chain identifier, sequence number and
    #    insertion code.  Note that a residue is created as part
    #    of a particular molecule.
    r = m.newResidue("HOH", " ", 1, " ")

    #    Now we create the atoms.  The newAtom function arguments
    #    are the atom name and its element type, which must be
    #    an instance of Element.  You can create an Element
    #    instance from either its name or atomic number.
    atomO = m.newAtom("O", Element("O"))
    atomH1 = m.newAtom("H1", Element(1))
    atomH2 = m.newAtom("H2", Element("H"))

    #    Set the coordinates for the atoms so that they can be displayed.
    from math import radians, sin, cos
    bondLength = 0.95718
    angle = radians(104.474)
    atomO.setCoord(Coord(0, 0, 0))
    atomH1.setCoord(Coord(bondLength, 0, 0))
    atomH2.setCoord(Coord(bondLength * cos(angle), bondLength * sin(angle), 0))

    #    Next, we add the atoms into the residue.
    r.addAtom(atomO)
    r.addAtom(atomH1)
    r.addAtom(atomH2)

    #    Next, we create the bonds between the atoms.
    m.newBond(atomO, atomH1)
    m.newBond(atomO, atomH2)
```

```
    #    Finally, we add the new molecule into the list of
    #    open models.
    openModels.add([m])

#    Call the function to create a water molecule.
createWater()
```