Python Scripting

Matthieu Bal and Karl Bzdusek
Advanced Development, Radiation Oncology Systems
April, 2 2014

PHILIPS

Scripting Applications in Clinics

Large sets of patient data:
- Data mining to characterize patient outcomes
- Retrospective studies

Process and workflow management:
- Tracking and guiding processes
- Tumor board discussions
- Clinical decision support

Task automation:
- Data transfer
- Plan verification and documentation
- Class solutions
- Automate steps in tasks

Offering to Research Collaborators

- Script Manager ScriptTree
  - Allow to manage both Pinnacle^3 and Python scripts
- Pinnacle^3 scripting
  - For retrospective studies using a non-research release of Pinnacle^3
- Python Scripting
  - Replace plugin support to collaborators
  - Easy to learn, very powerful rapid prototyping environment
  - Full access to almost all Pinnacle^3 objects and messages
  - Specific functions added to facilitate typical tasks:
    - Full access to volume data
    - User defined objectives for IMRT and IMPT
    - User defined optimization routine
    - Flexible mechanism to add additional functionality

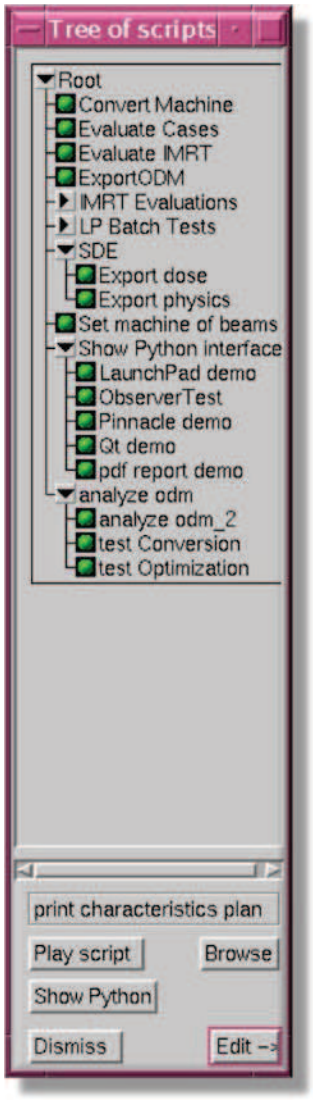
PHILIPS

Scripts Manager

Typical users have more scripts than the ~10 currently supported with HotScripts

Features:

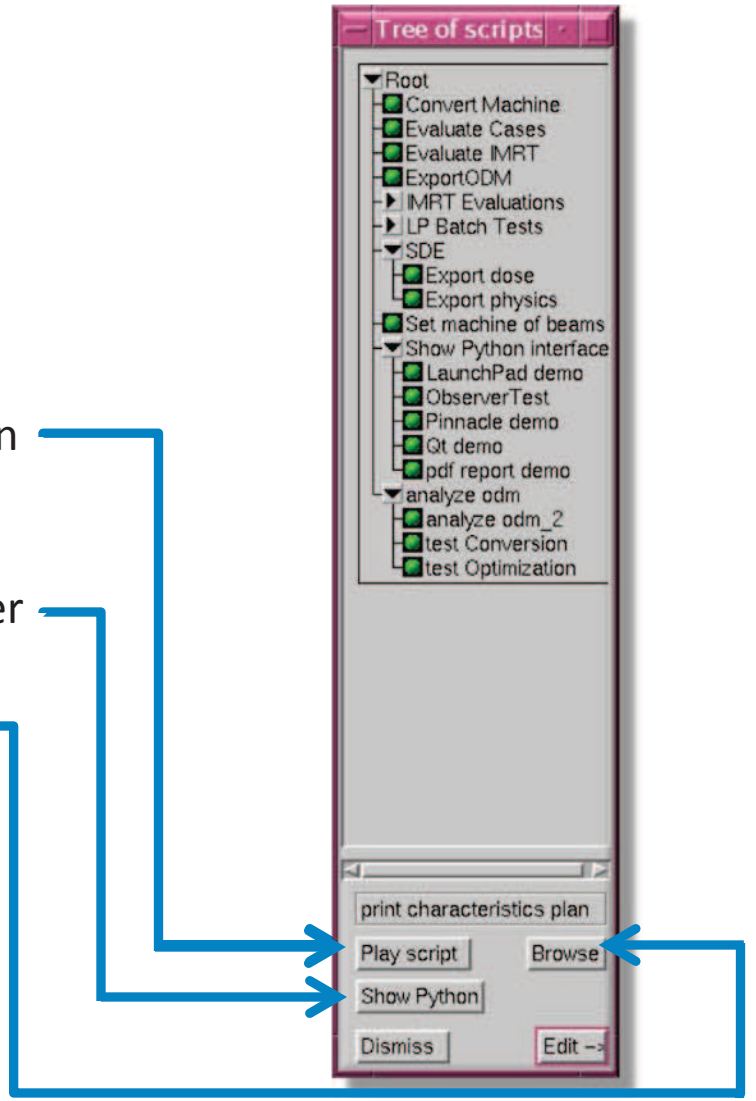
- Organizes scripts in tree structure
- Handles both classic and Python scripts
- Directly open script in editor



PHILIPS

Scripts Manager

- Run the currently selected Pinnacle or python script
- Show the window with the python interpreter
- Open a file browser to select a script file



PHILIPS

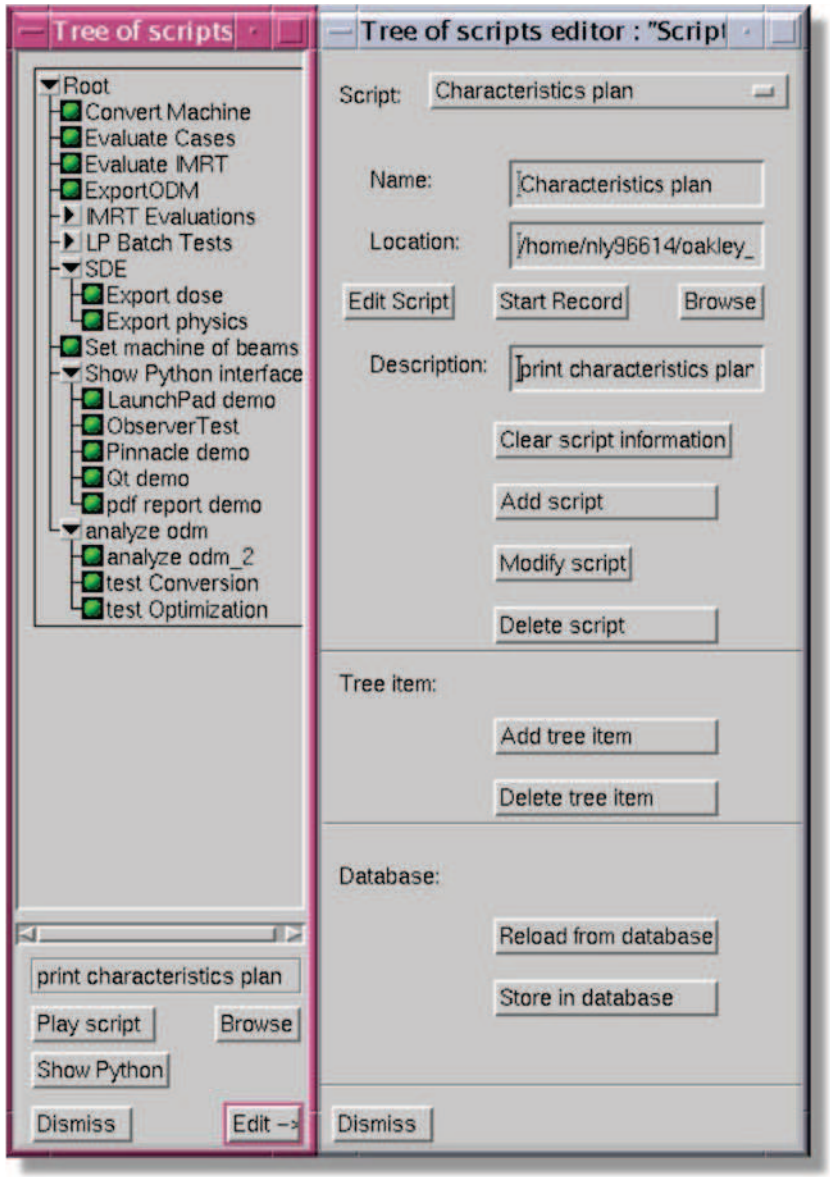
Script Tree Editor

Editing, two lists:

1. List of scripts, per script:
   - Name
   - Location
   - Description

2. List of tree items, per tree item:
   - Parent
   - Script

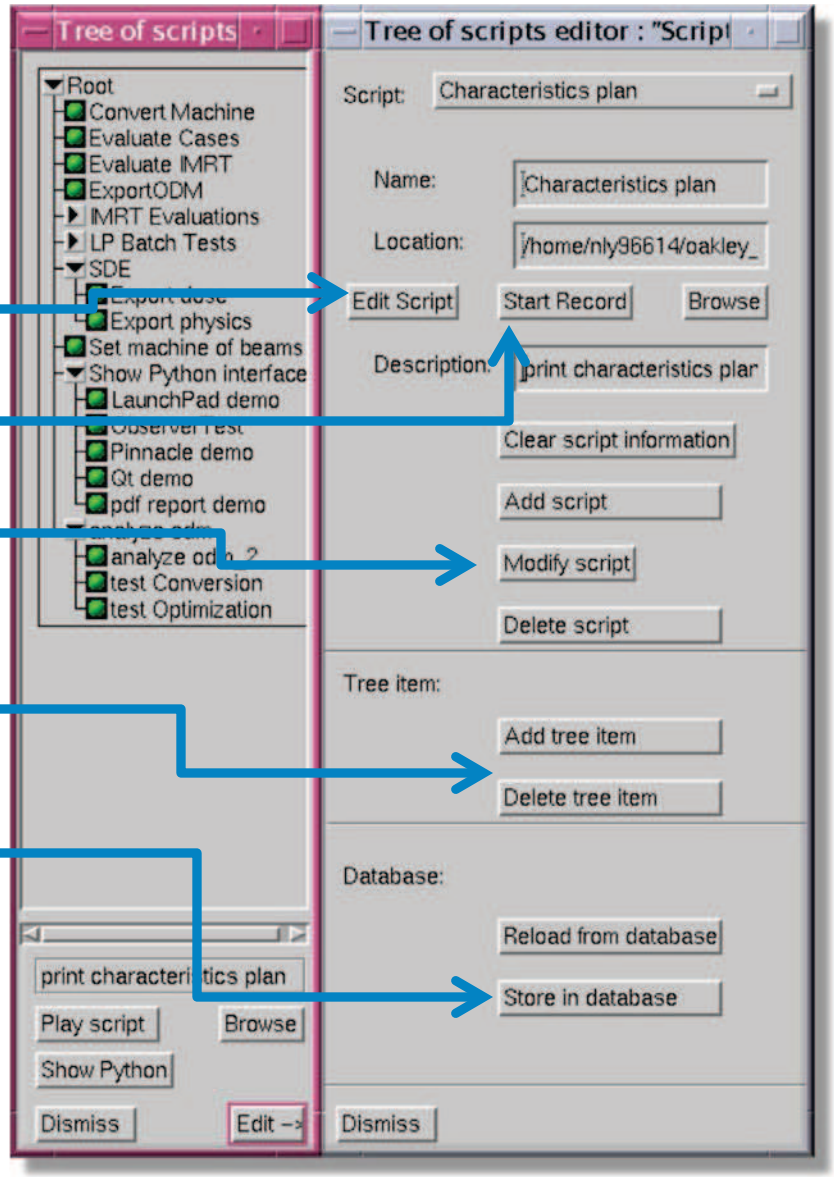
Modified items need to be committed to database. This allows to undo changes and editing simultaneously by several users.



PHILIPS

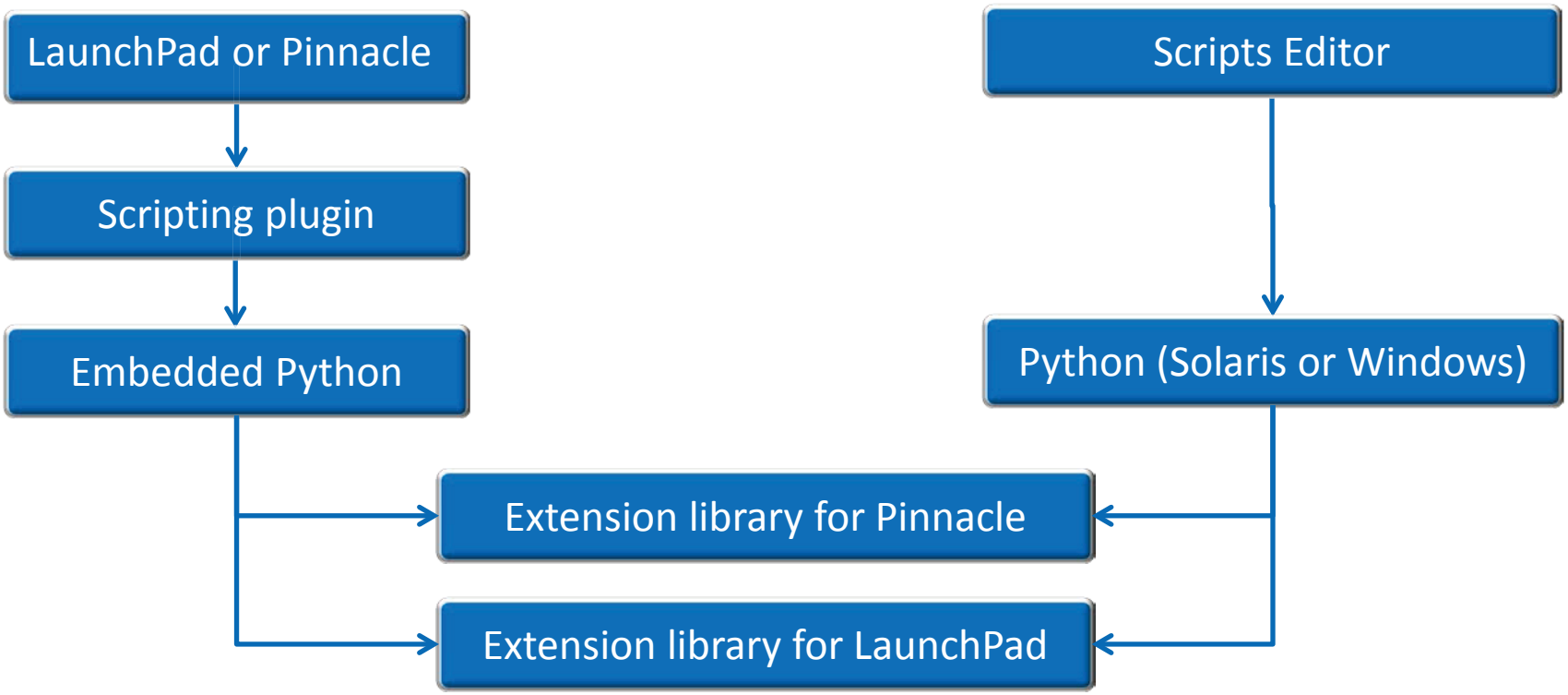
Script Tree Editor

- Open a text editor and edit script
- Start recording a Pinnacle^3 script
- Add, modify or delete a script item from the list
- Add or delete a script from the tree
- Commit local changes to database or reload data from database



PHILIPS

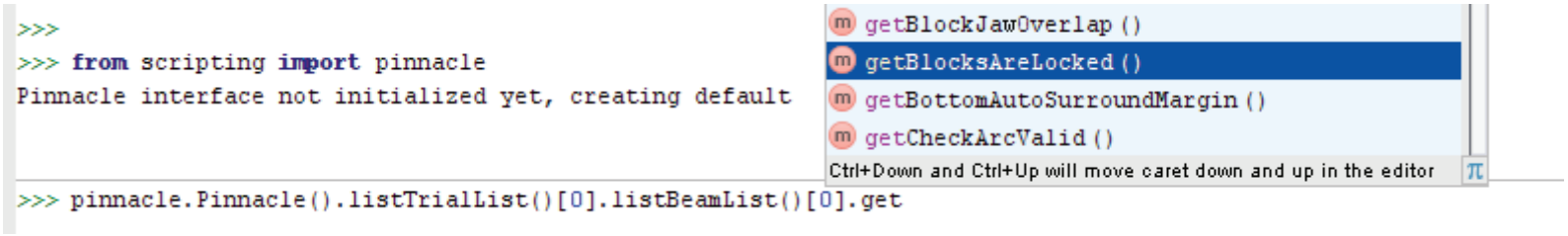
System Design



PHILIPS

Modern Script Editors

- Use any standard Python editor (http://wiki.python.org/moin/PythonEditors):
  - Code completion, syntax validation
  - Test script offline (arguments, user permissions, …)
  - UNIX or Windows

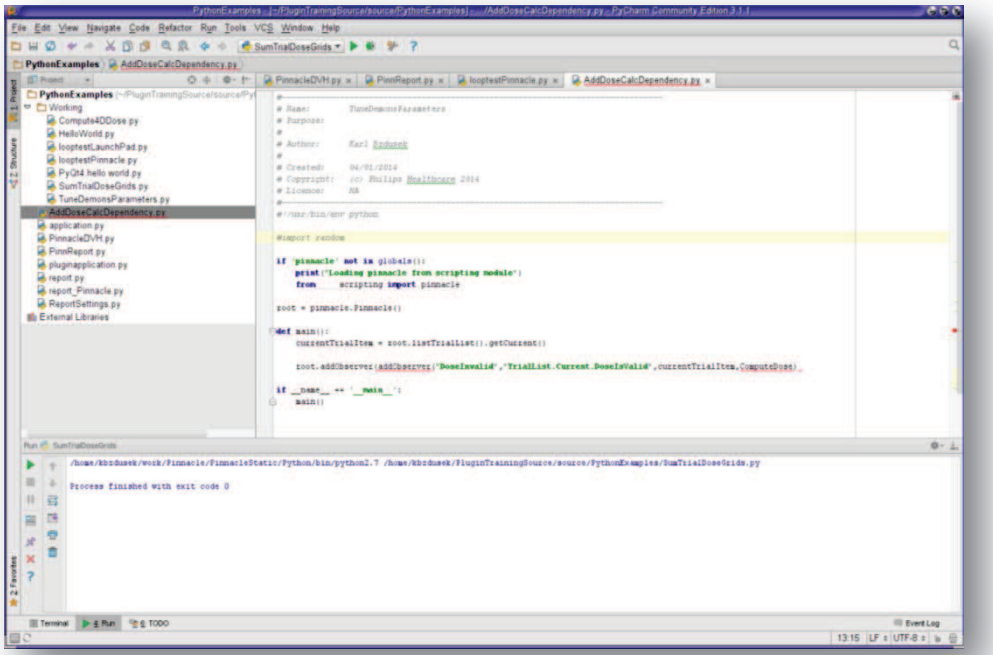


PHILIPS

Editing Python on UNIX

PyCharm

- Graphical editor that comes as part of the training package
- Modern editor features such as auto completion, syntax validation, debugging, etc



PHILIPS

Editing Python scripts on Windows

Pyscripter or pycharm

- Write Pinnacle python scripts in 64 bit windows
- Scripts written in windows can be transferred to pinnacle to be executed
- You can test your scripts in windows prior to transferring them to run on Pinnacle

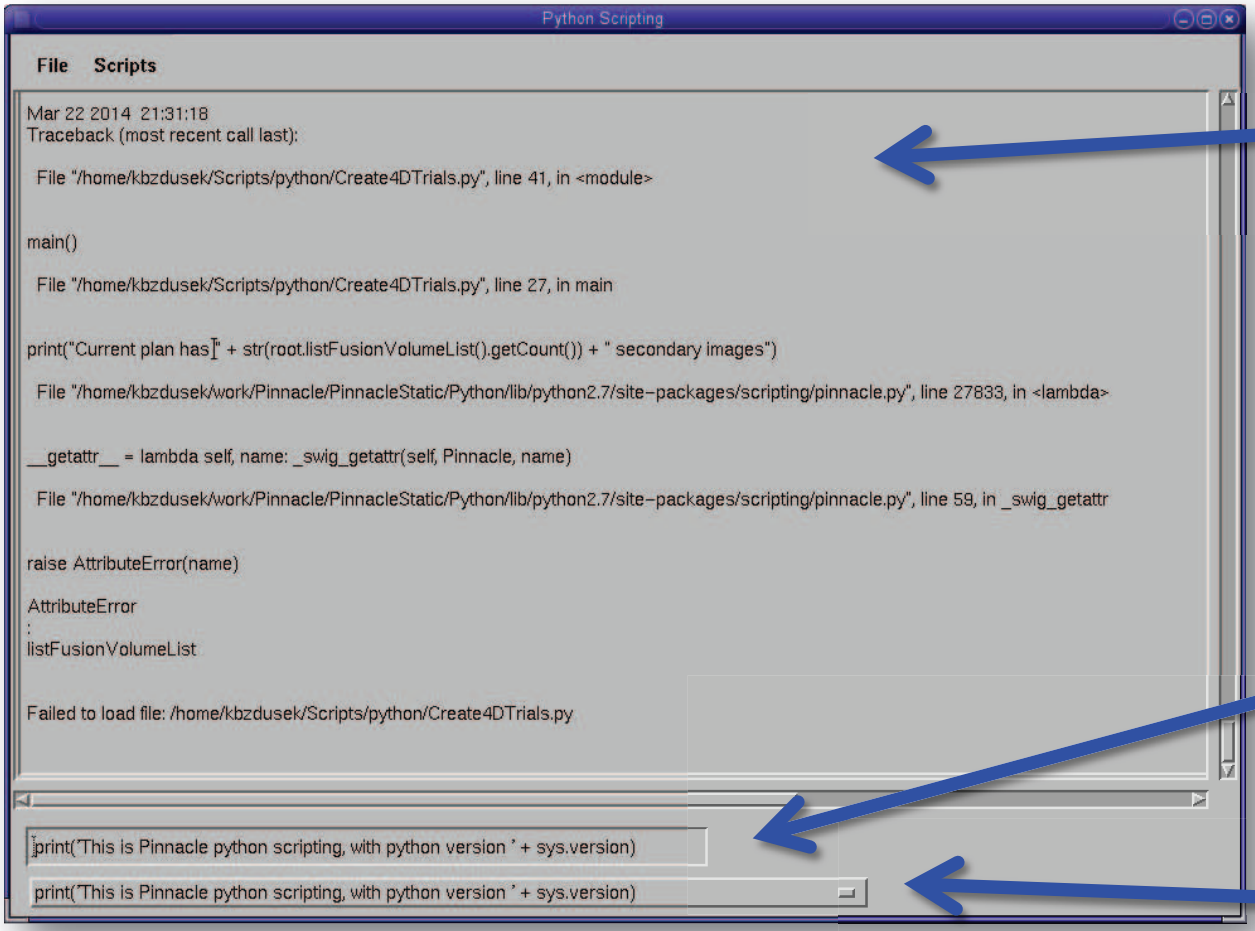
1. Download python-2.7.2.amd64.msi and install

2. Download and install:
   64 bit version of PyScripter from http://code.google.com/p/pyscripter/
   Or PyCharm from http://www.jetbrains.com/pycharm

3. Must place all the files from scripting.zip to *c:/Python27/Lib/site-packages/scripting*

4. Download and install numpy from
   http://www.lfd.uci.edu/~gohlke/pythonlibs/numpy-MKL-1.7.1.win-amd64-py2.7.?exe

Python Interpreter Interface



Python output

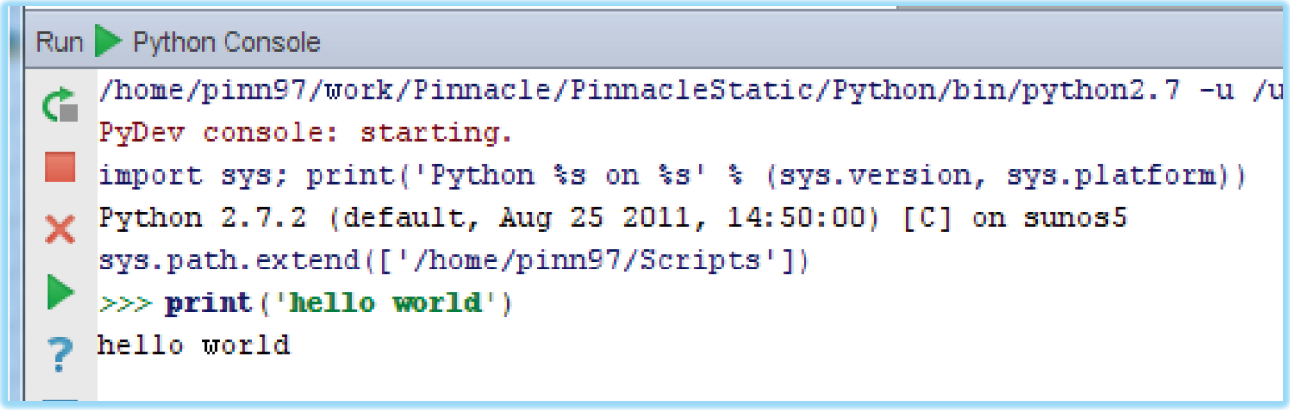
Python input

history

PHILIPS

Programming in Python: Basics

Python is an interactive, interpreted language:



- We typed *print('hello world')* and got back as output *hello world*

```
>>> import math
>>> math.exp(-2)
0.1353352832366127
```

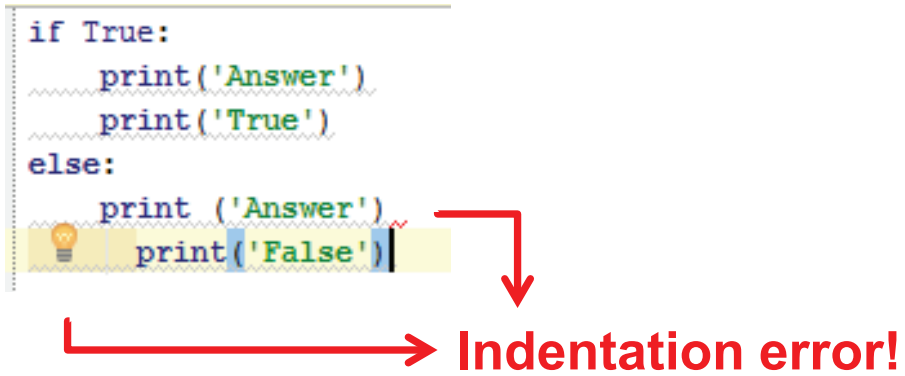
- We loaded the math module and used the exponent function

http://www.tutorialspoint.com/python/python_quick_guide.htm

Programming in Python: Basics

- A Python identifier is a name used to identify a variable, function, class, module, or other object
  - Case sensitive
  - Only numeric, alpha, or underscore (_) characters are allowed

- Lines and indentations
  - No braces or brackets!
  - Instead spaces are used to define blocks of code



PHILIPS

Programming in Python: Basics

- String literals can be single ('), double (") and triple (''' or """) quotes as long as the same type of quote starts and ends the string
- # = Comments

```
# First comment
print "Hello, Python!“;          # second comment
```

- Use semi-colons “;” only to use multiple statements in the same line

```
import sys; x = ‘test'; sys.stdout.write(x + '\n')
```

- Blank lines are ignored

Programming in Python: Control Flow

Control flow statements:

- begin with the keyword *if, for, while, …*
- terminate with a colon (:)
- are followed by one or more lines which make up the *suite*

```
for fusionVolumeItem in root.listFusionVolumeList():
    createNewTrial()                          # Create a new trial
    newTrialItem = root.listTrialList()[-1]   # Get the last trial which is also the new trial

    print(newTrialItem.getName())             # print the name of the new trial
```

- Functions are defined using the "**def**" keyword

```
def testRoi():                                # can also provide arguments
  "test manipulation of rois"                 # optional docstring
  names = ['prostate', 'bladder', 'rectum']   # array of names

  for name in names:
    roi = root.listRoiList().addNewObject()
    roi.setName(name)
    print('\t' + roi.getName())
```

Programming in Python: Control Flow

- *if, else, elif:* Can be nested

```
if expression1:
   statement(s)
      if expression2:
         statement(s)
      elif expression3:
         statement(s)
      else statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

- *while* and *for* loops: Support *break* and *continue*

Programming in Python: Code Groups

- Module
  - Allows a user to put definitions in a file and use them in other scripts or in an interactive instance of the interpreter
  - Definitions from a module can be ***imported*** into other modules

```
import CreateRegionOfInterests          # Import module CreateRegionOfInterests
```

- ***main*** module
  - Top level module that is a collection of variables that you have access to in a script or in calculator mode

Python and Pinnacle^3

PHILIPS

LaunchPad and Pinnacle^3 Messages

All messages with well defined input and output are available as python functions:

- get… Get a value of an object in LaunchPad or Pinnacle^3
- set… Set a value of an object in LaunchPad or Pinnacle^3
- do… Perform an action
- sub… Access a subobject of an object
- list… Access a list of objects

Work with messages:

- Query: queryInt, queryFloat, queryDouble, queryString, queryVectorString
- Set: setBool, setInt, setFloat, setDouble, setString, setVectorString
- Actions: doAction, doActionThreaded (without blocking python interpreter)
- Observer: addObserver, removeObserver, getObservers
- getMessages(): returns all messages of object

Pinnacle Lists with Python

- Pinnacle contains a number of object lists such as a trial list, beam list, DVH list, ROI list, etc

- The following are interface commands to access objects in the list:
  - **getCurrent**() gets current object in list
  - **list()[**index**]** returns the object at the index
  - **getFirst**() gets the first object in the list
  - **getLast**() or list()**[**-1**]** returns the last element in the list
  - **getCount**() gets the number of objects in the list

  - **addNewObject**() add new item to list
  - **removeObjectByNumber**(index) remove object at index
  - **removeObjectByName**(name) remove object with name

PHILIPS

Example 1: Add Items to a List

```
If 'launchpad' not in globals():
   print("Loading launchpad from scripting module"
   from scripting import launchpad

root = launchpad.PatientDB()                                  # Get a pointer to the Patient DB

def main()
   buildTag = root.getAppVersionAndRevision()                 # Get some revision info and print
   print(buildTag)
   addObjects()                                               # Call addObjects function

def addObjects():                                             # Function definition
   root = launchpad.PatientDB()                               # Get a handle to the main object
   institution = root.listInstitutionList().addNewObject()    # Add institution
   patient = institution.listPatientLiteList().addNewObject() # Add patient
   plan = patient.subPatient().listPlanList().addNewObject()  # Add plan
   return "Added plan %s to patient %s in institution %s" % (str(plan), str(patient), str(institution))

# Another option
if __name__ == '__main__':                                    # Only execute if file directly called
   addObjects()
```

Pinnacle^3 Prompts: Automatic Handling

To allow to run scripts autonomously, a new mechanism is provided by the python plugin to capture prompts in LaunchPad and Pinnacle^3.

- The automatic capturing can be switched on by the following Pinnacle command:
  PluginManager.ScriptingPlugin.AutoHandleMessages = 1;
- The default module used to handle the prompts is:
  PinnacleStatic/Python autoHandleMessages.py
- The module name used can be changed with the Pinnacle command:
  PluginManager.ScriptingPlugin.AutoHandlePythonModule = ...;

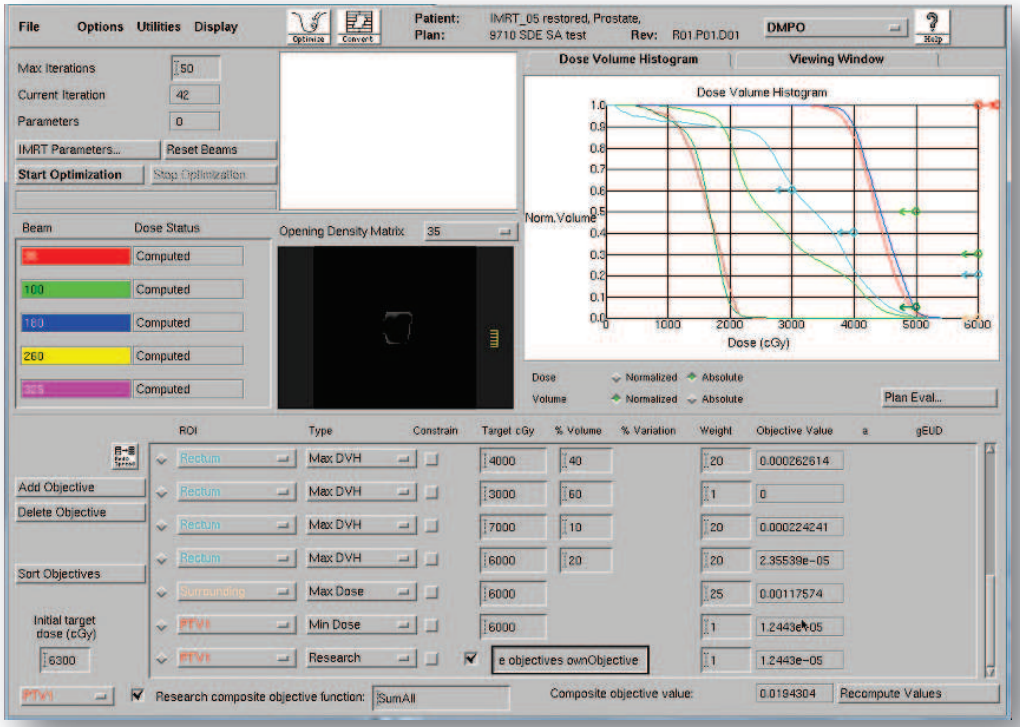
Documentation Objects and Messages

- How autocomplete works depends on the used editor
- A complete list of available python interface messages is documented in *PythonScriptingUserGuide.pdf*
  - You can look up the object by name (alphabetical order)
  - Message names are listed along with a short description

  For example:
  POI
  → DicomCoordDisplayOn query of type action
  → DicomXCoord query and set of type action

Creating New objective functions

- An interface to allow Python definitions of objective functions is provided
- Use the existing IMRT GUI to combine user defined an existing functions
- Traditional summing composite function is provided



PHILIPS

Example 2: Research Objectives

```
from numpy import *                                          # numpy is a scientific computing library
if 'pinnacle' not in globals():
   from scripting import pinnacle

doseLevel = 6000.0                                           # constant variable defined

def ownObjectiveFunction(par, dose, roi, volumes, weight):   # objective value computation
   funcValue = 0.0
   for roiItem, volumeItem in zip(roi, volumes):             # loop through roi voxel index and volume
      voxelDose = dose[roiItem]                              # get dose of roi voxel
      if voxelDose < doseLevel:                              # minimum dose objective
         relDeltaDose = (voxelDose - doseLevel) / doseLevel  # scale objective with minimum dose
         funcValue += volumeItem * relDeltaDose * relDeltaDose
   return funcValue * weight                                 # return objective value

def ownObjectiveGradient(par, dose, roi, volumes, weight, gradient):   # gradient computation
    for roiItem, volumeItem in zip(roi, volumes):            # loop through roi voxel index and volume
      voxelDose = dose[roiItem]                              # get dose of roi voxel
      if voxelDose < doseLevel:                              # minimum dose objective
         deltaDose = voxelDose - doseLevel
         gradient[roiItem] += 2.0 * weight * volumeItem * deltaDose / (doseLevel * doseLevel)
```


PHILIPS

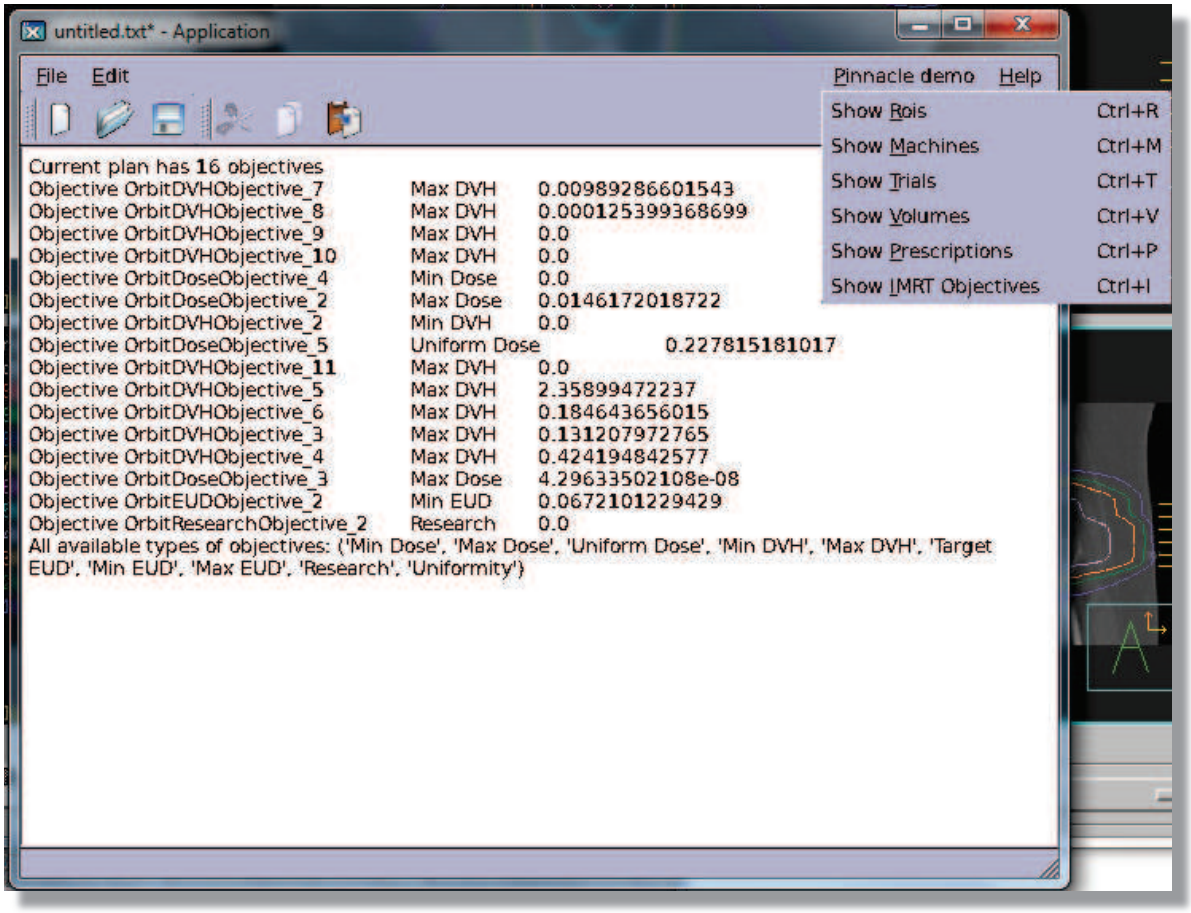
Example 3: PyQt4 User Interface

```
from PyQt4 import Qt                                          # import PyQt4 module partly
from pluginapplication import PluginApplication               # PluginApplication is a base class for an
                                                              application that works together with Launchpad or Pinnacle
class HelloApplication(PluginApplication):                    # Derive class from PluginApplication
   def addWidgets(self):                                      # add widgets to the window
      self.hellobutton = Qt.QPushButton("Say 'Hello world!'", None)
      self.hellobutton.connect(self.hellobutton, Qt.SIGNAL("clicked()"), self.slotSayHello)
      self.hellobutton.show()                                 # connect function to an event slot

   def slotSayHello(self):                                    # function that is called when button is clicked
      print "Hello, World!“

if __name__ == "__main__":                                    # execute when main code
   if 'launchpad' in globals():                               # create application
     app = HelloApplication('launchpad')                      # work together with Launchpad
   elif 'pinnacle' in globals():
     app = HelloApplication('pinnacle')                       # work together with Pinnacle
   else:
     app = HelloApplication('')                               # work standalone
   app.addWidgets()
   app.exec_()                                                # start application
```

PyQt4 Interface: Create Your Own Windows



PHILIPS

Example 4: Report Generation – part 1

```
import PinnReport                                   # module to create reports, based on ReportLab
import ReportSettings                               # helps to convert colors

if 'pinnacle' not in globals():
   from scripting import pinnacle

class DVHGroup:
   def __init__(self):                              # constructor
      self.data = []                                # empty array for dvh data
      self.color = []                               # colors of dvh lines
      self.width = []                               # width of lines
      self.stroke = []                              # stroke of lines

.....                                               # actual export of DVH data into placeholders above

if __name__ == '__main__':
   myPDF = PinnReport.PinnacleReport('/home/testuser/', 'dvhTest', "DVH Report", [])
   dvh = DVHGroup()                                 # create object of class DVHGroup
   dvh.getDVHData()                                 # import dvh data from Pinnacle[3]
   dvh.createPlot(myPDF)                            # add DVH plot to pdf
   myPDF.save()                                     # store pdf plot
```


PHILIPS

```
def addDVH(self, color = 'red', stroke = 'Solid', width = 1, data=[]):
    self.data.append(data)
    self.color.append(ReportSettings.colorSet[ReportSettings.colorsGraph.index(color)])    # convert colo
    if stroke == 'Double Dash':                          # get stroke of line
        self.stroke.append([5, 2])
    elif stroke == 'Solid':
        self.stroke.append(None)
    if width == 1:                                       # get width of line
        self.width.append(0.1)
    else:
        self.width.append(width)

def getDVHData(self):                                    # export data from Pinnacle3
    root = pinnacle.Pinnacle()
    for pinDvh in root.subDVHGroup().listDVHList():
        dvhData = []
        currentVolume = pinDvh.getVolume()               # total volume of roi of dvh
        roiVolume = currentVolume
        for item in range(0, pinDvh.subData().getNumberOfPoints()):
            currentVolume -= pinDvh.subData().getValueAtIndex(1, item)  # cumulative DVH
            dvhData.append([pinDvh.subData().getValueAtIndex(0, item), currentVolume / roiVolume])
        self.addDVH(pinDvh.subRegionOfInterest().getColor(), \
                pinDvh.queryString("LineStyle.LineStyle"),
                pinDvh.queryInt("LineStyle.LineWidth"), dvhData)

def createPlot(self, pdf):                               # add plot as chart to pdf document
    pdf.addChart(self.data, 'text', self.color, self.width, self.stroke)
```

Example 5: Dose Accumulation

Accessing volume data in the dose grid

```
from scripting import pinnacle

def accumulateDose(sourceTrials, destinationTrial):
    root = pinnacle.Pinnacle()
    names = root.listTrialList().getNames()                                    # get the names of all trials
    if set(sourceTrials) <= set(names) and destinationTrial in names:          # check all trials exist
        destData = root.listTrialList()[destinationTrial].subDoseGrid().getDataFloat() # get dose data
        destDim = root.listTrialList()[destinationTrial].subDoseGrid().getDimension() # get dimensions dose
        for source in sourceTrials:
            sourceGrid = root.listTrialList()[source].subDoseGrid()
            if sourceGrid.getDimension() == destDim:                           # check dimensions match
                destData += sourceGrid.getDataFloat()                          # data is numpy ndarray

if __name__ == '__main__':
    accumulateDose(['Trial_1', 'Trial_2'], 'Trial_3')
```


PHILIPS

Example 6: Batch Plans Processing

```
import os

def evaluatePlans(script):
   root = launchpad.PatientDB()
   for inst in root.listInstitutionList():                                  # loop over all institutions
      root.listInstitutionList().setCurrent(inst.getName())
      root.listInstitutionList().getCurrent().doLoadCurrentPatientFromDB()  # load data from sql database
      for patIndex in range(0, inst.listPatientLiteList().getCount()):      # loop over all patients
         inst.listPatientLiteList().setCurrent(patIndex)
         inst.doLoadCurrentPatientFromDB()                                  # load data from sql database
         root.setInt('WindowList.LPPatientSelect.WidgetList.ListTree.Recreate', 1)
         patient = inst.listPatientLiteList().getCurrent().subPatient()
         for plan in patient.listPlanList():                                # loop over all plans
            patient.listPlanList().setCurrent(plan.getName())
            if (patient.queryInt('IsLocked') == 0):                         # check whether plan is locked
              root.setStartPinnacleBatch(script)

if __name__ == '__main__':
   if not os.environ.has_key('EVALUATION_SCRIPT'):                          # environment variable defining
                                                                            # script to be used in Pinnacle3
os.environ['EVALUATION_SCRIPT'] = os.environ['HOME'] + '/Scripts/characteristics.py'
   evaluatePlans('/EvaluatePinnacle.Script')                                # helping Pinnacle3 script
```


PHILIPS

Pinnacle^3 Dependencies: Observers

Dependencies are mechanism in Pinnacle^3 to get signaled when a property of an object changes

Syntax:

- **addObserver**(Name, Message, Module, Function):
  Create an observer “Name” that is triggered when “Message” is used
  The Python “Function” is called in the “Module”. Argument Module may be an empty string, in which case the function is expected to be already defined in the main module.
- **removeObserver**(Name)
  Remove an observer named Name from the object.
- **getObservers**()
  Get a list of names of existing (Python scripting)observers of that object.
- **getMessages**()
  Get a list of messages of the object that can be used to attach an observer to.

Example 7: Observers Example

```
if 'pinnacle' not in globals():
    print("Loading pinnacle from scripting module")
    from scripting import pinnacle

ipm = pinnacle.Pinnacle().subPluginManager().subInversePlanningManager()
imrtTrial = ipm.subOptimizationManager().subCurrent().listTrialList().getCurrent()

# define callback function
def iteration(message):
# show message triggering call with current optimization iteration
    print('Callback called with argument %s at iteration %d' % (message, imrtTrial.getCurrentIteration()))

# add the observer
imrtTrial.addObserver('IteratorObserver', 'CurrentIteration', 'observerDemo', 'iteration')

# print existing (python) observers of object
print(imrtTrial.getObservers())
```

Code for Examples

- PinnacleStatic/Python/objectives.py (example 2)
- PinnacleStatic/Python/PyQt4.hello world.py (example 3)
- PinnacleStatic/Python/PinnacleDVH.py (example 4)
- PinnacleStatic/Python/dose accumulation.py (example 5)
- PinnacleStatic/Python/evaluatePlans.py (example 6)
- PinnacleStatic/Python/observerDemo.py (example 7)

Installing Standalone Python

Standalone Python **does not** yet come as part of a Pinnacle^3 research install however it can be added easily:

- Complete *PinnacleStatic/Python:*
  - Download Python.tgz from our ftp site
  - Move it to */usr/local/adacbeta/PinnacleStatic/* as root.
  - Extract it using gtar (gtar –xzvf Python.tgz)

- Configure PyCharm:
  - add the standalone python interpreter in configure:
    python interpreters -> add interpreter (+ sign on the right) -> point to *PinnacleStatic/Python/bin/python*

  *(Note: you may have to change permissions on the file as root to be rw for all users; chmod 777 *, chmod 777 .)*

  Make sure LB_LIBRARY_PATH_64 is pointing to /usr/postgres/8.3-community/lib/64

  Add the pycharm.sh directory to your PATH environment variable to run it anywhere

- Add additional search paths:
  - *PinnacleStatic/lib/i386* for offline Pinnacle scripting

- Had to create StartResearch set of plugins


PHILIPS