RAYSTATION 8B

Scripting Guideline

Trace back information:
Workspace Main version a432
Checked in 2018-12-20
Skribenta version 5.2.030

REGULATORY INFORMATION

Disclaimer

Canada: Carbon ion treatment planning, proton Wobbling, proton Line Scanning, BNCT planning, Machine Learning and photon Monte Carlo dose computation are not available in Canada for regulatory reasons. These features are controlled by licenses and these licenses (rayCarbonPlanning, rayCarbonPhysics, rayWobbling, rayLineScanning, rayBoron, rayDeep, rayDeepPlanning, rayDeepSegmentation, rayLearnerBase and rayPhotonMonteCarlo) are not available in Canada.

Japan: Carbon ion treatment planning, proton Wobbling, proton Line Scanning, Monte Carlo for proton PBS dose computation, PBS with aperture blocks, BNCT planning, Machine Learning and photon Monte Carlo dose computation are not available in Japan for regulatory reasons. These features are controlled by licenses and these licenses (rayCarbonPlanning, rayCarbonPhysics, rayWobbling, rayLineScanning, rayBoron, rayDeep, rayDeepPlanning, rayDeepSegmentation, rayLearnerBase and rayPhotonMonteCarlo) are not available in Japan.

The United States: Carbon ion treatment planning, BNCT planning, Machine Learning and photon Monte Carlo dose computation are not available in the United States for regulatory reasons. These features are controlled by licenses and these licenses (rayCarbonPlanning, rayCarbonPhysics, rayBoron, rayDeep, rayDeepPlanning, rayDeepSegmentation, rayLearnerBase and rayPhotonMonteCarlo) are not available in the United States.

Declaration of conformity

CE 0413

Complies with 93/42/EEC Medical Device Directive as amended by M1 to M5. A copy of the corresponding Declaration of Conformity is available on request.

Safety notices

This user documentation contains WARNINGS concerning the safe use of the product. These must be followed.

WARNING!

The general warning sign informs you of a risk for bodily harm. In most cases the risk is related to mistreatment of the patient.

Note: *The note informs of a risk for loss of data or other impairment of the work.*

Copyright

Printed material

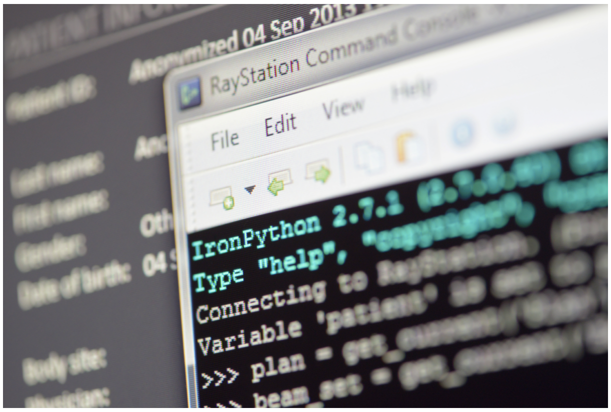
A hard copy of the Instructions For Use and other applicable manuals are available upon request.

Trademarks

RaySearch Laboratories AB (publ) including its subsidiaries is hereafter referred to as RaySearch.

TABLE OF CONTENTS

1 INTRODUCTION



About this guideline

Common processes and use cases in RayStation can be automated using scripting. This document describes how to create scripts and includes several examples. In some cases, line breaks have been inserted in order to make the script fit to the page widthwise.

The scripts can be run from a database or from a file. The database scripts are available for all users and can be categorized as general scripts or as module specific scripts.

- General scripts can be run from all modules.
- Module specific scripts can only be run from a specific module.

The new scriptable features in RayStation 8B are described in *chapter 3 Changes in RayStation 8B*.

In this chapter

This chapter contains the following sections:

Precautions

WARNING!

Validate all scripts for their intended use. Carefully validate all scripts before they are used clinically. Be aware that not all actions are scriptable/recordable. A recorded script may not be equivalent to the GUI steps performed in RayStation when recording. (508971)

WARNING!

Coordinate systems in Scripting. In scripting, coordinates are always interpreted according to the DICOM coordinate system specification. For machine coordinate systems, the IEC standard and the DICOM standard are equivalent but for patient coordinates, they differ, see Chapter *Display of coordinates, movements and scales* in *RSL-D-RS-8B-IFU, RayStation 8B Instructions For Use*. Machine scales are not taken into account. (508973)

WARNING!

Units in Scripting. In scripting, dose is always measured in cGy. Clinic settings for Gy/cGy are not taken into account. (508974)

WARNING!

Dosimetry unit in scripting. In scripting, the primary dosimetry unit is always MU. Configuration of other primary dosimetry units is not taken into account. All parameters related to beam meterset are named MU and contains values in MU. However, there exist script methods GetBeamNP() and SetBeamNP() on ion beams with primary dosimeter unit NP. (126108)

WARNING!

Use exception handling with care. When executing a script, exceptions not caught in the script will be handled by the script framework automatically, causing the script execution to stop and an error message to be displayed.

If exceptions are manually caught in the script (by adding exception handling to the script), it may override the script framework exception handling and as a consequence, the error message will not be displayed.

If exception handling is added to the script, ensure that all important error messages are displayed. (508972)

1.1 PYTHON

The programming language used for scripting in RayStation is CPython 2.7 or 3.6. Two implementations of Python are supported; IronPython and CPython. IronPython was the original Python implementation in RayStation and has better support for creating graphical user interfaces, whereas CPython has better support for third-party extensions. Most of the content in this guideline is applicable for both IronPython and CPython, but some of the examples only work for IronPython. For more information about scripting in CPython, see *chapter 7 Scripting in CPython*.

Python tutorials

Script developers with no previous experience of Python programming are strongly encouraged to go through a Python tutorial. A short introduction of Python aimed at experienced programmers can be found at http://hetland.org/writing/instant-python.html. A more thorough tutorial and the complete documentation can be found at www.python.org.

Install IronPython

IronPython 2.7.1 should be installed together with RayStation.

Install CPython

CPython 2.7.14 and CPython 3.6.5 should be installed together with RayStation.

Scripting environments

In addition to the pre-installed CPython and IronPython interpreters, it is possible to create user-defined scripting environments. These are in essence python virtual environments stored in the database, maintained by admin users but usable by everyone. A scripting environment is based on either CPython 2.7 or 3.6 and provisioned with any selection of CPython packages.

1.2 RUN SCRIPTS IN RAYSTATION

There are two ways to run scripts in RayStation, either a script can be run directly from its file or it can be imported to the clinical database of RayStation and run directly from the scripting tab.

Run a script from a file

To run a script from a file:

1. Click the **Script creation...** button in the **Scripting** tab.

   This opens the **Script creation** dialog.

2. Click the **Open...** button and browse to the location of the script file and click **Open**.

   Once a script has been selected, it will be shown in the **Script creation** dialog and the **Play** button will be enabled.

3. Use the **Python interpreter** drop-down list to select which Python interpreter the script should be executed with.

4. Click **Play** to run the script.

Import a script to the clinical database

To import a script to the clinical database:

1. Click the **Script management...** button in the **Scripting** tab.

   This opens the **Script management** dialog.

2. Click the **Import...** button and browse to the location of the script file and click **OK**.

   Once a script has been selected, it will be shown in the **Script management** dialog.

3. Use the **Python interpreter** drop-down list to select which Python interpreter the script should be executed with.

4. Optional, validate the script before saving it to the database. To validate the script, click the **Validate** button.

5. Click the **Save** button to save the script to the database.

   When the script is saved to the database, it will be reachable directly from the scripting tab, from any instance of RayStation that uses the clinical database to which the script is imported.

   If the script has not been validated, the settings in the **Scripting** tab might need to be changed for the script to be visible. To change the settings, click the **Settings...** button and uncheck the **Show only validated scripts** checkbox. To run an imported script, click the button with a green triangle next to the name of the script.

1.3 COORDINATE SYSTEMS AND UNITS IN THE SCRIPTING INTERFACE

The scripting interface only uses the DICOM coordinate system. All dose units in the scripting interface are given in cGy regardless of the settings in the clinical database. All length, area and volume units are given in cm, cm^2 and cm^3 respectively.

2 EXPLORE THE SCRIPTING INTERFACE

There are basically three different things that can be done with scripting:

- get values of properties for different objects
- set values of properties for different objects
- perform actions on different objects

This chapter will discuss good ways to find the actions that can be performed from scripting and the properties that are exposed in the scripting interface. The easiest way to get started with scripting in RayStation is to explore what can be done using scripting. A few ways to explore is to use the recording functionality in Script creation to create scripts from actions in the GUI, to run the interactive console in RayStation and to have a look at the state tree.

In this chapter

This chapter contains the following sections:

2.1 RECORD ACTIONS

The recording of actions is triggered from the **Script creation** dialog.

Record a script

To record a script:

1. Click the **Script creation…** button. This opens the **Script creation** dialog.

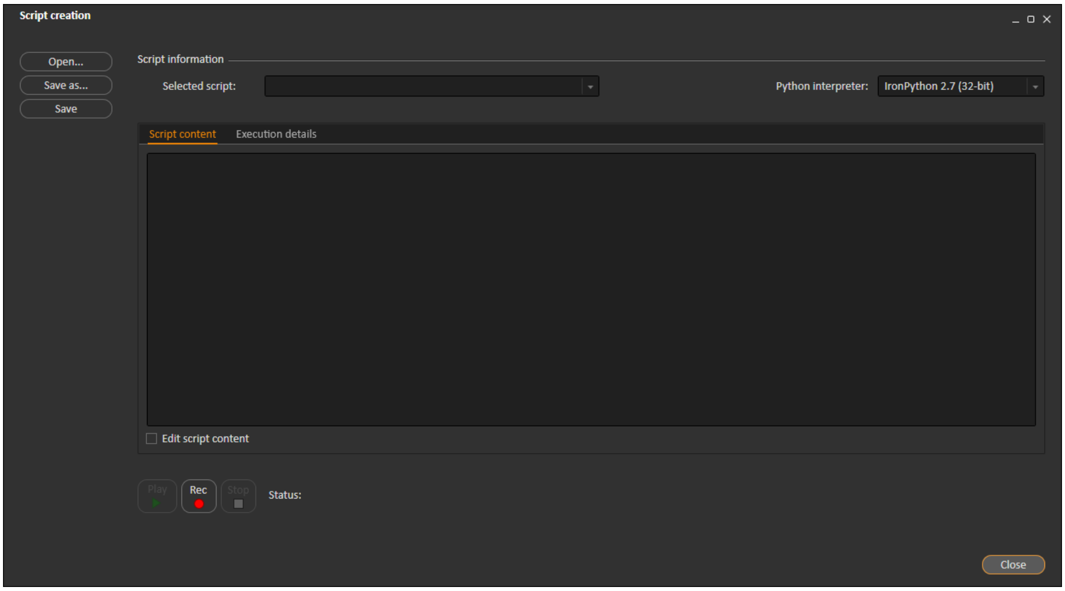
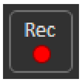
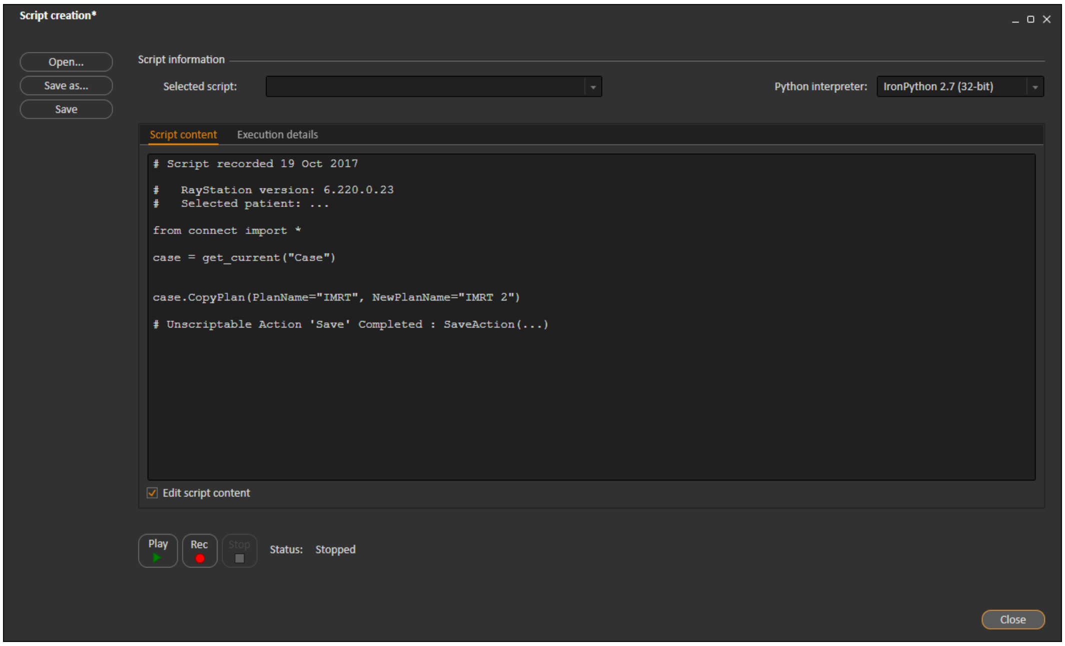


Figure 1. The **Script creation** dialog.

2. Click the **Rec** button (the **Script creation** dialog is automatically closed).



3. Perform a set of actions.
4. Stop the recording by clicking the **Stop** button in the **Script creation** dialog or in the **Scripting** tab.
5. After recording, view the **Script content** and the **Execution details**.

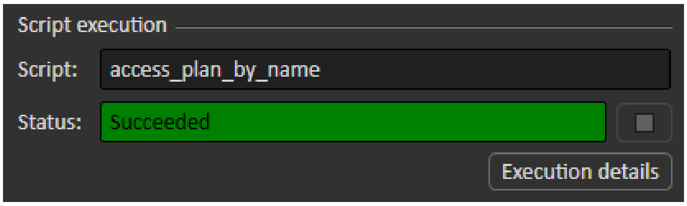


```
# Script recorded 19 Oct 2017

#   RayStation version: 6.220.0.23
#   Selected patient: ...

from connect import *

case = get_current("Case")


case.CopyPlan(PlanName="IMRT", NewPlanName="IMRT 2")

# Unscriptable Action 'Save' Completed : SaveAction(...)
```

Figure 2. The **Script creation** dialog, after a script has been recorded.

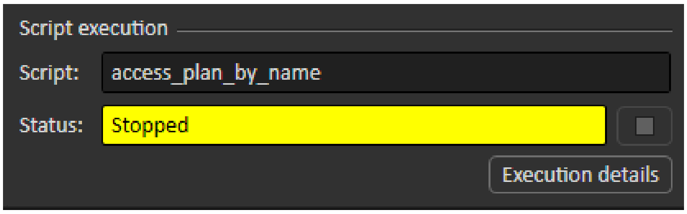
6. If desired, check the **Edit script content** checkbox and edit the script content.
7. Save the script as a file by clicking the **Save** or **Save as...** button. Ctrl + S can also be used.
8. Use the **Python interpreter** drop-down list to select which Python interpreter the script should be executed with. User-created scripting environments can also be selected here.
9. To run the recorded script, click the **Play** button.

The script will be executed and the script execution status and name are displayed in the status field at the bottom of the **Scripting** tab.



Script: access_plan_by_name
Status: Succeeded

10. Click the **Execution details** button. This opens the **Execution details** dialog which consists of two tabs where the **Execution details** tab displays state changes that occurred during the run and the **Execution summary** tab which lists all modified patients.


Script execution

Script: access_plan_by_name

Status: Stopped

To stop a script during run, click the **Stop** button. The script execution status field will indicate that the script was stopped.

Note: *In the execution details, sub modifications may be listed that are part of an overall modification of the patient. The completion of the sequence of modifications are indicated at the end of the last sub modification.*

Unscriptable actions

Recording actions is a good way to learn the scripting syntax for GUI actions. However, there are a few caveats associated with recording actions. Not all actions in RayStation are supported from scripting. If the recorded actions are not scriptable, there will be a warning message that lists the unscriptable actions. These actions will be included in the generated script as comments.

For some of the unscriptable actions there are work-arounds. Some work-arounds for unscriptable actions are described in *chapter 9 Work-arounds for unscriptable actions*. Some unscriptable actions get partly included in the recorded script, as a line containing something similar to `with CompositeAction(...):`. These lines are not followed by the proper indentation and will yield the following error message: `Error: expected an indented block`. These lines need to be removed in order for the recorded script to be executable.

Use a script for a general case

Scripts that are generated by recording actions in the GUI often have hard-coded parameters for simple arguments. If it is desired to use a recorded script for a general case, many of the hard-coded values need to be replaced. An example of a recorded script with comments on changes that can be made to adapt the script for a more general case is described in *section 5.1 Recorded script on page 41*.

2.2 THE INTERACTIVE CONSOLE

The interactive console is an IronPython or CPython interpreter that is connected to RayStation. It can be used to get and set properties through the scripting interface interactively, as well as perform actions, and it is a nice and easy way to explore the scripting interface.

Running the interactive console

The interactive console is started by running the script `run_console.py` in RayStation. The script is located in the ScriptClient directory in the directory where RayStation.exe is located. It is recommended to import this script, as it probably will be run quite often.

When the interactive console is run, all objects currently loaded in the system will be retrieved as variables in the interactive console. If no patient is loaded when the console is run, only the clinic database, machine database and patient database are retrieved. If a patient and a case are loaded but no plan, the patient, machine and clinic databases are retrieved as well as the current patient, the current case and the current examination. If a plan is loaded as well, then the current plan and the current beam set are also retrieved when the console is run. The loaded objects and the names of the corresponding parameters are displayed when the console is run.

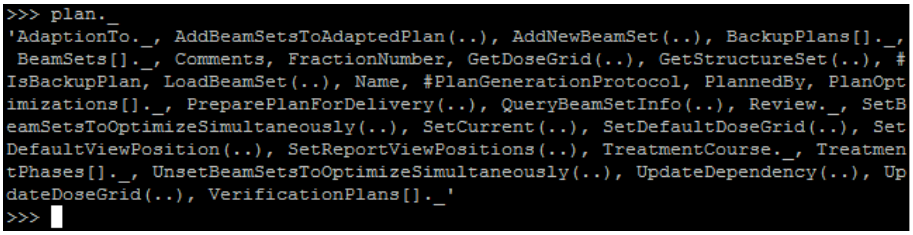
Tab completion

The interactive console supports tab completion, that is, the name of an object can be followed by a period (.) and then followed by pressing the tab key to see the properties and methods of the object. If you want to be able to do tab completion on an item in a list, you need to access the desired list item with tab completion, or access the item as a variable. For example, if you want to do tab completion on a treatment plan named "IMRT" for your current case, you cannot do tab completion if you have accessed the treatment plan like this: `case.TreatmentPlans['IMRT']`. However, if you have accessed the treatment plan by tab completion, which will look like this, `case.TreatmentPlans._IMRT`, or if you have created the variable "imrt", `imrt = case.TreatmentPlans['IMRT']`, you can use tab completion. The `Case` object and the `TreatmentPlan` object will be explained further in *section 4.3 Cases on page 26* and *section 4.4 Treatment plans on page 26* respectively. For tab completion to work in the CPython console, the package pyreadline has to be installed.

Object and method properties

To get the names of all available properties, objects and methods of an object displayed in the console, you can write the name of the object followed by `._` and press enter. Figure 3 shows the use of this feature on a `TreatmentPlan` object. The different markings of the different names show what the names represent.

- A name followed by `._`, for example `AdaptationTo._`, shows that this name represents an object.
- A name followed by `(..)`, for example `AddNewBeamSet(..)`, shows that this name represents a method.
- A name followed by `[]._`, for example `BackupPlans[]._`, shows that this name represents a list of objects.
- A name followed and preceded by nothing, for example `Comments`, shows that the name represents a mutable property.
- A name preceded by `#`, for example `#IsBackupPlan`, shows that the name represents an immutable property.



```
>>> plan._
'AdaptionTo._, AddBeamSetsToAdaptedPlan(..), AddNewBeamSet(..), BackupPlans[]._,
 BeamSets[]._, Comments, FractionNumber, GetDoseGrid(..), GetStructureSet(..), #
IsBackupPlan, LoadBeamSet(..), Name, #PlanGenerationProtocol, PlannedBy, PlanOpt
imizations[]._, PreparePlanForDelivery(..), QueryBeamSetInfo(..), Review._, SetB
eamSetsToOptimizeSimultaneously(..), SetCurrent(..), SetDefaultDoseGrid(..), Set
DefaultViewPosition(..), SetReportViewPositions(..), TreatmentCourse._, Treatmen
tPhases[]._, UnsetBeamSetsToOptimizeSimultaneously(..), UpdateDependency(..), Up
dateDoseGrid(..), VerificationPlans[]._'
>>>
```

Figure 3. Example of the ._ notation for a `TreatmentPlan` object.

To get all the properties of an object listed in the console, use `._print`. This command displays the value of each property and the number of items in each list of the object.

To get documentation on an object or method in the console, use `._help`. If this syntax is used on an object, it will list documentation for all its objects, properties and methods. If the syntax is used on a method, it will list the documentation of the method. If this syntax is used on a list or a property, it will cause a `SystemError` or `AttributeError` respectively.

The help() method which provides documentation for built-in Python types can also be used to get the documentation for objects and methods from RayStation.

A good way to get started with scripting is to try code snippets from this document in the interactive console.

2.3 THE STATE TREE

The state tree displays the state of the currently loaded patient in a tree view. For each object, the scripting interface properties and methods are shown and if you click on an object or a method, its documentation is shown. By exploring the state tree you learn how different objects are connected in the scripting interface and it is the easiest way to view documentation for the scripting methods.

Running the state tree

The state tree can be run either from the interactive console or as a stand-alone script. The console is kept open and active while the state tree is shown, so it is possible to get and set properties and perform actions while having the state tree open. Updates done from the console will show up in the state tree if you re-open the object you have changed. The following command opens the state tree from the console:

```
statetree.RunStateTree()
```

When the state tree is opened from a script, the script must not finish before the state tree has been closed. To achieve that, the following syntax should be used.

```
import statetree
statetree.RunStateTree(True)
```

It is also possible to display a state tree for the machine of the currently loaded beam set. The syntax for running the machine state tree in the console is as follows:

```
statetree.RunMachineStateTree()
```

The following code opens the machine state tree from a script.

```
import statetree
statetree.RunMachineStateTree(None, True)
```

There are also state tree windows for the patient, machine and clinic databases. They are opened with the commands `statetree.RunPatientDBStateTree()`, `statetree.RunMachineDBStateTree()` and `statetree.RunClinicDBStateTree()`. As for `RunStateTree()`, "`True`" should be given as an input argument when the commands are executed from a script.

You can right-click on an item in the state tree and copy the path to that object in the scripting interface. If you find the object you are looking for in the state tree and you want to add this object to a script, right-click on the object and paste the object path into your script.

3 CHANGES IN RAYSTATION 8B

In RayStation 7 and RayStation 8A it is not possible to access the list of templates, only single items, using scripting. Therefore, for each template type a corresponding scriptable method has been given where either a list of info objects or a list of names describing the template is returned. The following methods has been added:

- `patient_db.GetPatientModelTemplateInfo()` replaces `patient_db.TemplatePatientModels`
- `patient_db.GetClinicalGoalTemplateInfo()` replaces `patient_db.TemplateTreatmentOptimizations`
- `patient_db.GetOptimizationFunctionTemplateInfo()` replaces `patient_db.TemplateTreatmentOptimizations`
- `patient_db.GetBeamListTemplateInfo()` replaces `patient_db.TemplateTreatmentSetups`
- `patient_db.GetMbsModelTemplateInfo()` replaces `patient_db.TemplateMbsModels`
- `patient_db.GetResponseParametersTemplateInfo()` replaces `patient_db.TemplateResponseParameters`
- `patient_db.GetTemplateMaterialNames()` replaces `patient_db.TemplateMaterials`
- `patient_db.GetColorMapTemplateInfo()` replaces `patient_db.TemplateColorMaps`
- `patient_db.GetArcTrajectoryTemplateNames()` replaces `patient_db.TemplateArcTrajectories`
- `patient_db.GetPlanExplorationTemplateInfo()` replaces `patient_db.TemplatePlanExplorations`

Replace the old methods by following the context below, for example:

- Prior:

  ```
  templates = patient_db.TemplatePatientModels
  template = templates[i]
  ```

- New:

  ```
  templateInfo = patient_db.GetPatientModelTemplateInfo()
  template = patient_db.LoadTemplatePatientModel(templateName =
  templateInfo[i][‘Name’], lockMode = ‘Read’)
  ```

There is a special case for TemplateMaterials:

```
templateNames = patient_db.GetTemplateMaterialNames()
```

```
templateList = patient_db.GetTemplateMaterial()
```

```
template = [x for x in templateList.Materials if x.Name == templateNames[i]]
```

This is also affecting the recording of templates, and some manual editing of the resulting recorded script is required, for example:

- Prior:

  ```
  plan = get_current("Plan")
  ```

  ```
  db = get_current("PatientDB")
  ```

  ```
  plan.PlanOptimizations[0].ApplyOptimizationTemplate
  ```

  ```
  (Template=db.TemplateTreatmentOptimizations['HN PTV'])
  ```

- New:

  ```
  plan = get_current("Plan")
  ```

  ```
  db = get_current("PatientDB")
  ```

  ```
  template = db.LoadTemplateOptimizationFunctions(templateName = ‘HN PTV’,
  lockMode = ‘Read’)
  ```

  ```
  plan.PlanOptimizations[0].ApplyOptimizationTemplate(Template=template)
  ```

4 ACCESS COMMON OBJECTS

This chapter describes how to access common objects such as for example patients, treatment plans and dose distributions.

In this chapter

This chapter contains the following sections:

4.1 THE GET_CURRENT METHOD

The `get_current` method is the method in RayStation scripting that enables you to get handles to RayStation objects. Every script that interacts with RayStation will include at least one `get_current` method. The following is a list of all available `get_current` arguments:

```
patient = get_current('Patient')
# 'patient' is the currently loaded patient.
case = get_current('Case')
# 'case' is the currently loaded case.
plan = get_current('Plan')
# 'plan' is the currently loaded treatment plan.
beam_set = get_current('BeamSet')
# 'beam_set' is the currently loaded beam set.
examination = get_current('Examination')
# 'examination' is the currently loaded examination.
patient_db = get_current('PatientDB')
# 'patient_db' is the current patient database.
machine_db = get_current('MachineDB')
# 'machine_db' is the current machine database.
clinic_db = get_current('ClinicDB')
# 'clinic_db' is the current clinic database.
```

If you try to access an object type for which there is no currently selected object, for example if you write `plan = get_current("Plan")` when no plan is loaded, you will receive the following error: `SystemError: Invalid objectHandle`.

4.2 PATIENTS

The currently open patient can be accessed using the `get_current` method. Unlike other scripting objects, you can only get a handle to the patient that is currently loaded. This has the effect that you cannot have a working handle to more than one patient at a time. In order to work with a patient that is not currently loaded, you must load it. This is done using patient database methods.

Prerequisites before loading a patient

To be able to load a patient, you must have access to its patient information. The patient information is accessed by a query to the patient database using the method `QueryPatientInfo`. This method requires one parameter, `Filter`, which is a Python dictionary. If `Filter` is an empty dictionary, the query method will return information on all patients in the database. Otherwise, the method will return information on all patients in the database matching the filter.

The possible filter keys are `Id, FirstName, LastName, PatientId, BodySite, Gender, DisplayName, BirthDate, Physician` and `IsImmutable`. The `Filter` value property supports regular expressions for string parameters. If regular expressions are used, only the patients whose properties match the filter parameter exactly will be returned, whereas if a normal string parameter is used, all patients whose properties contain the filter string will be returned.

The following code snippet shows some examples of how to query the patient database:

```
patient_db = get_current("PatientDB")
# Get info on all patients in the database.
all_patients = patient_db.QueryPatientInfo(Filter={})
```

```
# Get info on all mutable patients in the database.
mutable = patient_db.QueryPatientInfo(Filter={'IsImmutable': False})
# Get info on all patients whose last name is exactly Smith.
smiths = patient_db.QueryPatientInfo(Filter={'LastName':'^Smith$'})
# Get info on all patients whose last name contains Smith.
contains_smith = patient_db.QueryPatientInfo(Filter={'LastName':'Smith'})
```

Loading a patient

To load a patient, use the `LoadPatient` method of the patient database. It has one required parameter, `PatientInfo`. The next code snippet shows how to load a patient found in the database whose last name is Smith. If no patient with last name Smith is found, or if more than one patient with last name Smith is found, an exception is raised.

```
patient_db = get_current("PatientDB")
last_name = 'Smith'
info = patient_db.QueryPatientInfo(Filter={'LastName':
                                   '^{0}$'.format(last_name)})
if len(info) == 1: # Check that info contains exactly one item.
   patient = patient_db.LoadPatient(PatientInfo=info[0])
   # The patient with last name 'Smith' is now loaded and 'patient'
   # is the handle to that patient.
else:
   # No patient, or more than one patient, with last name 'Smith' found.
   # Raise an exception.
   raise Exception("No patient or more than one patient with last
   name '{0}' in the database".format(last_name))
```

Note: *When a new patient is loaded from the scripting interface, all unsaved changes will be lost. Thus, make sure to save the changes before loading a new patient. This is done using the method patient.Save().*

Index service

In order to query for and load a patient from a secondary database in an index service, some additional arguments are needed.

```
patient_db = get_current("PatientDB")
# Query for patient info for patient with ID 123456
# in both primary and secondary database
info = patient_db.QueryPatientInfo(Filter={'PatientID':'123456'},
                                           UseIndexService=True)

# This will throw an error message if patient is located
# in a secondary database
patient = patient_db.LoadPatient(PatientInfo=info[0])

# If AllowPatientUpgrade is set to True,
# then the patient will be copied/moved to the
# primary database, if located in a secondary database
patient = patient_db.LoadPatient(PatientInfo=info[0],
                                 AllowPatientUpgrade=True)
```

4.3 CASES

As with patients, the currently loaded case is accessed using the `get_current` method. It is possible to access a case for the currently loaded patient which is not the currently loaded case. All cases of a patient can be found in the list `Cases` of the `Patient` object. A case in this list can be accessed using either the name or the index of the case. The following code snippet shows how to access a case named "Original" for the currently loaded patient:

```
patient = get_current('Patient')
case_name = 'Original'
# Get a handle to the case named 'Original'
try:
   case = patient.Cases[case_name]
except:
   print 'No case named {0} exists for \
   the current patient'.format(case_name)
```

It is possible to access data for cases that are not the currently loaded case from scripting. However, it can be useful to load another case. This is done using the method `SetCurrent()` of the `Case` object to load. The following code snippet shows how to load the first case of the currently loaded patient:

```
patient = get_current('Patient')
# Get a handle to the first case of the patient
case_to_load = patient.Cases[0]
# Load the case
case_to_load.SetCurrent()
```

Note: *The SetCurrent() method can only be used if all changes are saved. Use patient.Save() to save changes.*

4.4 TREATMENT PLANS

As with patients and cases, to access the currently loaded treatment plan, use the `get_current` method. It is possible to access a treatment plan for the currently loaded case which is not the currently loaded treatment plan. All treatment plans of a case can be found in the list `TreatmentPlans` of the `Case` object. A treatment plan in this list can be accessed using either the name or the index of the treatment plan. The following code snippet shows how to access a plan named "IMRT" for the currently loaded case:

```
case = get_current("Case")
plan_name = 'IMRT'
# Get a handle to the plan with name 'IMRT'.
try:
   plan = case.TreatmentPlans[plan_name]
except:
   print "No plan named {0} exists for the current \
   case".format(plan_name)
```

Although it is possible to access and modify data of a plan which is not the plan that is currently loaded in the system, you might want to be able to load a plan in the GUI, for example for viewing doses and dose statistics after optimization or dose calculation. Loading a `TreatmentPlan` object, as with a `Case` object, is done using the `SetCurrent()`-method.

The next code snippet shows how to load a plan named "SMLC". If no plan with this name exists for the current case, an exception is raised.

```
case = get_current("Case")
plan_name = 'SMLC'
try:
   # Try to access the plan by its name.
   plan_to_load = case.TreatmentPlans[plan_name]
except:
   # Could not find a plan with name 'SMLC'.
   raise Exception("No plan named {0} exists for \
   the current case".format(plan_name)
plan_to_load.SetCurrent()
```

Note: *The SetCurrent() method can only be used if all changes are saved. Use patient.Save() to save changes.*

4.5 BEAM SETS AND BEAMS

The currently loaded beam set can be accessed using the `get_current` method. Other beam sets are accessed through the `BeamSets` list of the `TreatmentPlan` object that the beam set belongs to. As for cases and treatment plans, you can access the items of this list by using either the names of the beam sets or by using their indices.

It is possible to access beam sets that belong to other plans than the currently loaded plan, as long as the plans belong to the currently loaded case.

The following code snippet shows how to access the first beam set of the first treatment plan of the currently loaded case:

```
case = get_current("Case")
# Get the first beam set of the first treatment plan of the
# currently loaded case.
beam_set = case.TreatmentPlans[0].BeamSets[0]
```

It is also possible to load a beam set that is not currently loaded. As for cases and plans, the `SetCurrent()`-method is used. Note that it is not possible to load a beam set that does not belong to the currently loaded plan.

The next code snippet shows how to load a beam set named "5 beams" for the current plan. If no beam set with this name exists for the current plan, an exception is raised.

```
plan = get_current("Plan")
beam_set_name = '5 beams'
# Try to access the beam set named '5 beams'.
try:
   beam_set_to_load = plan.BeamSets[beam_set_name]
except:
   # No beam set named '5 beams' exists for the current plan.
   raise Exception("No beam set named {0} exists for \
   the current plan".format(beam set name))
beam_set_to_load.SetCurrent()
```

Note: *The SetCurrent() method can only be used if all changes are saved. Use patient.Save() to save changes.*

It should be noted that `BeamSet` objects do not have a `Name` property. Instead, the `DicomPlanLabel` property is used.

The beams are accessed from the `Beams` lists of the beam set objects. The beams can be accessed by either using their names or by using their indices.

4.6 IMAGE SETS AND STRUCTURE SETS

All image sets of a case are stored in the list `Examinations` of the `Case` object. As for cases, treatment plans and beam sets, examinations in this list can be accessed by either using their names or their indices.

From an `Examination` object, you can access the image stack and get information about for example the slice positions and the pixel size of the image set.

The following code snippet shows how to access the slice locations and the maximum and minimum of the stored pixel values of the currently selected examination:

```
examination = get_current("Examination")
slice_positions = examination.Series[0].ImageStack.SlicePositions
max_value = examination.Series[0].ImageStack.MaxStoredValue
min_value = examination.Series[0].ImageStack.MinStoredValue
```

Every `Examination` object has an associated structure set. The `StructureSet` objects are located in the list `StructureSets` of the `PatientModel` object of the `Case` object. The structure sets can be accessed either by using the names of their related examinations or by using their indices. A structure set holds geometric information defined on the associated image set, for example the geometries of the regions and points of interest.

To access the examination that a structure set is defined for, the `OnExamination` property can be used:

```
case = get_current('Case')
# Access the first structure set for the currently loaded case.
structure_set = case.PatientModel.StructureSets[0]
related_examination = structure_set.OnExamination
```

The items in the `StructureSets` list can be accessed by the name of its corresponding examination:

```
case = get_current("Case")
examination = get_current("Examination")
# Get the structure set associated with 'examination'.
structure_set = case.PatientModel.StructureSets[examination.Name]
```

`TreatmentPlan` objects have a method for easy access of the structure set associated to the plan:

```
plan = get_current("Plan")
# Access the structure set used by the plan.
structure_set = plan.GetStructureSet()
```

4.7 REGISTRATIONS

The list `Registrations` of the `Case` object holds the frame of reference registrations for the case. The registrations in this list can only be accessed by their indices, since a frame of reference registration has no name property.

A frame of reference registration has no easy access to the registered examinations. It has the properties `FromFrameOfReference` and `ToFrameOfReference` that can be used to find its associated examinations. Since multiple examinations can share a frame of reference, a frame of reference registration can map multiple examinations.

The following code snippet shows how to find all examinations mapped by the first frame of reference registration of the currently loaded patient:

```
case = get_current("Case")
for_registration = case.Registrations[0]
# Frame of reference of the "To" examination.
to_for = for_registration.ToFrameOfReference
# Frame of reference of the "From" examination.
from_for = for_registration.FromFrameOfReference
# Find all examinations with frame of reference that matches 'to_for'.
to_examinations = [e for e in case.Examinations if
                    e.EquipmentInfo.FrameOfReference == to_for]
# Find all examinations with frame of reference that matches 'from_for'.
from_examinations = [e for e in case.Examinations if
                      e.EquipmentInfo.FrameOfReference == from_for]
```

A frame of reference registration has all its associated structure registrations in the list `StructureRegistrations`. The items in this list can be accessed by name or by index.

4.8 DOSE DISTRIBUTIONS

The most common dose distributions in RayStation are fraction doses, plan doses and evaluation doses.

The fraction dose distribution of a beam set is a property of the beam set object. This dose distribution holds the dose per fraction for the given beam set. The total dose of a treatment plan can be found from the `TreatmentPlan` object. This dose distribution is the sum of the doses from all fractions of the plan. If the plan only contains one beam set, this dose is the fraction dose of the beam set scaled with the total number of fractions of the plan.

The following code snippet shows how to access the fraction dose of the currently loaded beam set and the total dose of the currently loaded plan:

```
beam_set = get_current("BeamSet")
# Get fraction dose for beam set.
fraction_dose = beam_set.FractionDose
plan = get_current("Plan")
# Get total dose for plan.
total_dose = plan.TreatmentCourse.TotalDose
```

Evaluation doses are a little trickier to find. They are stored in:

```
case.TreatmentDelivery.FractionEvaluations[i].DoseOnExamination[j].DoseEvaluations
```

`DoseEvaluations` is a list of evaluation doses for fraction i on examination j.

For all dose distributions, the voxel dose values are stored in `dose_distribution.DoseValues.DoseData`. In IronPython, `DoseData` is a C# three dimensional float array. The easiest way to work with these values is to convert them to a Python list as follows:

```
dose = list(dose_distribution.DoseValues.DoseData)
```

In CPython, `DoseData` is a three dimensional numpy array that is easy to work with as it is.

The voxel dose values are ordered so that the x-index of the voxel increases fastest, then the y-index and then the z-index. The coordinates refer to the DICOM coordinate system. How to export dose values to a text file is described in *section 5.6 Import and export dose values on page 72*.

4.9 REGIONS OF INTEREST AND POINTS OF INTEREST

The Regions Of Interest (ROIs) and the Points Of Interest (POIs) are found in lists in the `PatientModel` object.

```
case = get_current("Case")
rois = case.PatientModel.RegionsOfInterest
pois = case.PatientModel.PointsOfInterest
```

The items of these lists can be accessed either by their names or by their indices. The `RegionOfInterest` and `PointOfInterest` objects in these lists hold common information about the ROIs and POIs, for example name, color and type. They do not contain information regarding the geometry of the objects for the different examinations. The geometry objects are found in the `StructureSet` objects:

```
case = get_current("Case")
examination = get_current("Examination")
# Get ROI geometries for the structure set of the current examination.
roi_geometries =
   case.PatientModel.StructureSets[examination.Name].RoiGeometries
# Get POI geometries for the structure set of the current examination.
poi_geometries =
   case.PatientModel.StructureSets[examination.Name].PoiGeometries
```

The geometry objects in these lists can be accessed by either the name of the corresponding ROI/POI or by their indices. The ROI/POI geometry objects have the object `OfRoi/OfPoi` to access the region/point of interest object that the geometry belongs to.

Each ROI also has a geometry representation in the dose grids associated with the examination on which the ROI is defined. A dose grid is a discretized container for storing dose on an examination. The dose grid representation of an ROI geometry specifies which voxels in the grid that hold dose for the ROI geometry and how much of the volume of the voxel that is encompassed by the ROI geometry.

The dose grid representations of the ROI geometries are accessed from `DoseDistribution` objects and are represented in the dose grid of the dose distribution. The dose grid ROIs can be used to access the voxel indices of the ROI geometry in the dose grid and the fraction of the total volume of the ROI in a given dose grid voxel. The following code snippet shows how to access the dose grid representation of an ROI named "Bladder" from the fraction dose of the currently loaded beam set:

```
beam_set = get_current("BeamSet")
fraction_dose = beam_set.FractionDose
# Get dose grid representation of ROI "Bladder".
bladder_dg = fraction_dose.GetDoseGridRoi(RoiName="Bladder")
# Get voxel indices of dose grid representation of "Bladder".
voxel_indices = bladder_dg.RoiVolumeDistribution.VoxelIndices
# Get relative volumes of the corresponding voxel indices for
# dose grid representation of ROI "Bladder".
relative_volumes = bladder_dg.RoiVolumeDistribution.RelativeVolumes
```

A dose grid ROI object has the object `OfRoiGeometry` that can be used to access the ROI geometry that the dose grid ROI corresponds to.

4.10 OPTIMIZATION FUNCTIONS AND PARAMETERS

The optimization problems of a treatment plan are stored in the list `PlanOptimizations` of the `TreatmentPlan` object. There is one optimization problem object for every beam set of a treatment plan.

The objective functions of the optimization problem of the `i:th` beam set of the plan, `po = plan.PlanOptimizations[i]`, are stored in `po.Objective.ConstituentFunctions` and the constraint functions are stored in `po.Constraints`. The parameters for the individual functions are stored either in the `DoseFunctionParameters` property (for physical optimization functions) or in the `ResponseFunctionParameters` (for biological optimization functions).

The settings of the optimization problem, for example settings regarding dose calculation and segment conversion during optimization and settings for the beams, are stored in the `OptimizationParameters` object. The structure is a bit complicated, the easiest way to get an overview of it is to navigate to the optimization parameters in the state tree.

The following code snippet shows how to set the function type and dose level of the first objective function in the first optimization problem of the currently open treatment plan:

```
plan = get_current("Plan")
# Access the optimization problem of the first beam set.
po = plan.PlanOptimizations[0]
# Access first objective constituent function.
cf = po.Objective.ConstituentFunctions[0]
# Check that cf is physical and has a 'FunctionType' property.
if cf.DoseFunctionParameters != None and hasattr(cf.DoseFunctionParameters,
                                                 'FunctionType'):
   # Edit function type and dose level.
   cf.DoseFunctionParameters.FunctionType = 'MaxDose'
   cf.DoseFunctionParameters.DoseLevel = 4000
```

The next code snippet shows how to set some optimization settings for the first optimization problem of the currently selected plan:

```
plan = get_current("Plan")
# Access the optimization problem of the first beam set.
```

```
po = plan.PlanOptimizations[0]
opt_param = po.OptimizationParameters
# Set iterations before segment conversions to 10
opt_param.DoseCalculation.IterationsInPreparationsPhase = 10
tss = opt_param.TreatmentSetupSettings[0]
# SMLC settings
if plan.BeamSets[0].DeliveryTechnique == 'SMLC':
   # Set the maximum number of segments for all beams combined to 60
   tss.SegmentConversion.MaxNumberOfSegments = 60
# VMAT settings
if plan.BeamSets[0].DeliveryTechnique == 'DynamicArc':
   for bs in tss.BeamSettings:
      bs.ArcConversionPropertiesPerBeam.NumberOfArcs = 2
# Non-VMAT settings
else:
   # Do not allow beam split for any of the beams
   for bs in tss.BeamSettings:
      bs.AllowBeamSplit = False
```

An example of how to use scripting to copy optimization functions from a plan is described in *section 5.17 Copy optimization functions on page 80*.

4.11 LOAD AND UNLOAD TEMPLATE FROM SCRIPT

Before using a template in a script you need to call the corresponding load template method:

```
tat = patient_db.LoadTemplateArcTrajectory(templateName = '<template_name>',
      lockMode = 'Read'|'Write'| None)

tbl = patient_db.LoadTemplateBeamList(templateName = '<template_name>',
      lockMode = 'Read'|'Write'| None)

tc = patient_db.LoadTemplateColormap(templateName = '<template_name>',
      lockMode = 'Read'|'Write'| None)

tcg = patient_db.LoadTemplateClinicalGoals(templateName = '<template_name>',
      lockMode = 'Read'|'Write'| None)

tm = patient_db.GetTemplateMaterial()

tmm = patient_db.LoadTemplateMbsModel(templateName = '<template_name>')

tof = patient_db.LoadTemplateOptimizationFunctions(templateName =
      '<template_name>',lockMode = 'Read'|'Write'| None)

tpe = patient_db.LoadTemplatePlanExplorations(templateName =
      '<template_name>', lockMode = 'Read'|'Write'| None)

tpm = patient_db.LoadTemplatePatientModel(templateName = '<template_name>',
      lockMode = 'Read'|'Write'| None)

trp = patient_db.LoadTemplateResponseParameters(tissueName = '<tissue_name>',
      endpointOrTumorStage = '<endpoint_or_tumor_stage>')
```

Note: *Using `lockMode = None` or omitting the `lockMode` will result in a read lock by default.*

When the template is not needed anymore in the script it can be unloaded with:

```
t.Unload()
```

Note: *The `t.Unload()` command is not available for TemplateMaterial.*

4.12 DICOM IMPORT

All query methods are now hierarchical, meaning that either three levels (Patient -> Study -> Series) or four levels (Patient -> Study -> Series -> Instance) of queries must be done in order to produce data appropriate for passing to the import methods.

Note: *Methods that query on a given path are now NON-RECURSIVE. This behavior must be instead be handled by the API user.*

PatientDB:

- `QueryPatientsFromRepository`
- `QueryStudiesFromRepository`
- `QuerySeriesFromRepository`
- `QueryInstancesFromRepository`
- `QueryPatientsFromPath`
- `QueryStudiesFromPath`
- `QuerySeriesFromPath`
- `QueryInstancesFromPath`
- `ImportPatientFromRepository`
- `ImportPatientFromPath`

Patient:

- `ImportDataFromRepository`
- `ImportDataFromPath`

4.13 PATIENT, MACHINE AND CLINIC DATABASE

The patient database, machine database and clinic database are accessed by the commands `get_current("PatientDB")`, `get_current("MachineDB")` and `get_current("ClinicDB")` respectively.

The state tree does not show documentation for the databases. Thus, to explore the properties and methods of the databases you need to use the interactive console.

Patient database

The patient database has methods for querying the database of patient, examination and plan information and for querying DICOM paths and repositories of patients, series and studies. How to query the database for patient information is described in *section 4.2 Patients on page 24*. It also has methods for loading patients, for DICOM import of patients and for creating patient backups.

How to DICOM import patients is described in *section 5.3 DICOM import on page 47*, and how to create a patient backup is described in *section 5.2 Create a backup of a patient on page 46*.

From the patient database it is also possible to access template color maps, template CT filters, template materials, template MBS models, template patient models, template response parameters, template treatment optimizations and template treatment setups. All template objects are read-only.

Machine database

From the machine database it is possible to access the treatment machines and names and commissioning times for commissioned CT and CBCT imaging systems. All properties in the machine database are read-only.

There are three different query methods available for the machine database. These are `QueryCommissionedMachineInfo, QueryUncommissionedMachineInfo` and `QueryTemplateMachineInfo`. The methods return info for commissioned, uncommissioned and template machines respectively. The syntax is the same for all three methods; one parameter is required, `Filter`, which is a Python dictionary. If `Filter` is empty, the query method will return information on all machines of the queried type in the database. Otherwise, the method will return information on all machines in the database with properties that match the filter. The possible filter keys are `Name`, `IsDynamicArcCapable, IsDynamicMlcCapable, IsStaticArcCapable, CommissionedBy, HasMlc, IsLinac, IsIonMachine, HasRangeShifters` and `IsElectronCapable`.

The next code snippet shows how to query the machine database for all commissioned machines that are dynamic arc capable, access the first of these machines and get its maximum arc dose rate, its maximum jaw speed and its number of leaf pairs.

```
machine_db = get_current("MachineDB")
arc_machines = machine_db.QueryCommissionedMachineInfo(Filter=
               {'IsDynamicArcCapable':True})
name = arc_machines[0]['Name']
machine = machine_db.GetTreatmentMachine(machineName=name,
          lockMode=None)
# Get maximum arc dose rate for the first photon energy.
max_dose_rate = machine.ArcProperties.ArcPropertiesPerEnergy[0].MaxDoseRate
# Get maximum jaw speed.
max_jaw_speed = machine.Physics.JawPhysics.MaxJawSpeed
# Get number of leaf pairs.
num_of_leaf_pairs = machine.Physics.MlcPhysics.NumOfLeafPairs
```

The following code snippet shows how to access the names and the commission times of all commissioned CT and CBCT imaging systems in the machine database:

```
machine_db = get_current('MachineDB')
# Get a list of the names and the commission times of all commissioned
# CT systems.
ct_systems = machine_db.GetCtImagingSystemsNameAndCommissionTime()
# Get a list of the names and the commission times of all commissioned
# CBCT systems.
cbct_systems = machine_db.GetCbctImagingSystemsNameAndCommissionTime()
```

Clinic database

The clinic database enables access to the site settings set in Clinic Settings. All properties in the clinic database are read-only. The site settings for the clinic database are accessed from the method `GetSiteSettings()`. The site settings contain e.g. general default settings, default settings for dose and report templates, and settings for auto recovery and DICOM. From the site settings, it is also possible to access templates for plan reports and DICOM filters. The following code snippet shows how to access the names of the import and export DICOM filters, some dose default settings, and the names of the plan report templates:

```
clinic_db = get_current('ClinicDB')

# Access site settings
site_settings = clinic_db.GetSiteSettings()

# Get the names of DICOM import and export filters
dicom_filters = site_settings.DicomSettings.DicomFilters
import_filter_names = [f.Name for f in dicom_filters if f.Type == 'Import']
export_filter_names = [f.Name for f in dicom_filters if f.Type == 'Export']

# Get default settings for dose grid voxels and dose unit
dose_default_settings = site_settings.DoseDefaultSettings
x_size = dose_default_settings.DoseGridVoxelSizeX
y_size = dose_default_settings.DoseGridVoxelSizeY
z_size = dose_default_settings.DoseGridVoxelSizeZ
dose_unit = dose_default_settings.DoseUnit

# Get names of all plan report templates
report_templates = site_settings.ReportTemplates
plan_report_template_names = [t.Name for t in report_templates
                              if t.Type == 'TreatmentPlanReport']
```

4.14 NOTES ON RAYSTATION SCRIPTING SYNTAX

Method arguments

All method calls in the RayStation scripting interface need to be made with named arguments. Method arguments supplied without names will result in the following error:

```
SystemError: Method must be called with named arguments
```

Some methods have default arguments. To check if a method has default arguments, look up the documentation of the method in the state tree and see if a default value is specified for any of the method parameters. Recorded actions will list all method arguments, including the default arguments.

Omitting a required argument in a method call will cause a `SystemError`. Depending on the method and the omitted argument, the error message will differ.

ExpandoObjects

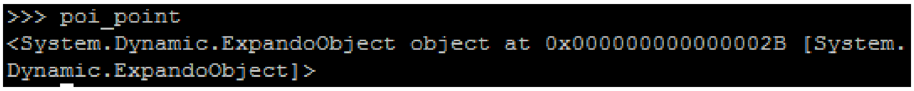
Some properties, for example points, are represented as `ExpandoObjects` in the IronPython scripting interface.

If you try to view for example the position of a point of interest in the console, you will get the output shown in Figure 4. To view the coordinates of the point of interest, you need to display the x, y and z properties of the object.

To set a property that is an `ExpandoObject`, use a Python dictionary that contains the properties of the `ExpandoObject` as keys. It is not possible to set the properties of an existing `ExpandoObject` and it is not possible to use an `ExpandoObject` to set a property that is an `ExpandoObject`.

The following code snippet shows how to display and set the position of a point of interest named 'POI1' in the currently loaded examination:

```
case = get_current("Case")
examination = get_current("Examination")
poi_name = 'POI1'
# Access the structure set of the current examination.
ex_name = examination.Name
structure_set = case.PatientModel.StructureSets[ex_name]
# Access the POI geometry.
poi_geometry = structure_set.PoiGeometries[poi_name]
# Access the POI position.
point = poi_geometry.Point
# Print the position.
print "POI position: {0}, {1}, {2}".format(point.x, point.y, point.z)
# Shift the POI position 0.1 cm in every direction.
new_point = {'x':point.x+0.1, 'y':point.y+0.1, 'z':point.z+0.1}
# Apply the new position.
poi_geometry.Point = new_point
```



```
>>> poi_point
<System.Dynamic.ExpandoObject object at 0x000000000000002B [System.
Dynamic.ExpandoObject]>
```

Figure 4. In IronPython, the position of a point of interest is stored as an `ExpandoObject`.

In CPython, the properties which are stored as `ExpandoObjects` in IronPython are instead stored using the Python class `ExpandoDictionary`. An `ExpandoDictionary` is a subclass of the built-in class `dict`, which allows access of values using dot-syntax. It is possible to set a property which is an `ExpandoDictionary` using either an `ExpandoDictionary` or a `dict`. Altering an `ExpandoDictionary` will not alter the underlying object in RayStation. The example script in the previous section works in both IronPython and CPython.

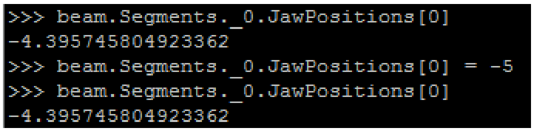
Setting list properties

If you want to set a property which is a list or an array, for example jaw positions or leaf positions of a photon beam, you need to set the whole list. If you try to set a single item, you will not get an error message, but the property will not have changed. Figure 5 shows what happens when you try to set the first item of the jaw positions list for a segment of a photon beam.

The following code snippet shows how to set the jaw positions of the first segment of the first beam in the currently loaded beam set:

```
beam_set = get_current("BeamSet")
```

```
beam = beam_set.Beams[0]
# Access the jaw positions of the first segment
jaw_positions = beam.Segments[0].JawPositions
# Change the position of the first (left) jaw
jaw_positions[0] = -5
# Set the jaw positions
beam.Segments[0].JawPositions = jaw_positions
```



```
>>> beam.Segments._0.JawPositions[0]
-4.395745804923362
>>> beam.Segments._0.JawPositions[0] = -5
>>> beam.Segments._0.JawPositions[0]
-4.395745804923362
```

Figure 5. Setting an item in the jaw position list has no effect.

The connect module

All scripts run in RayStation must import the `connect` module, for example using the following line of code:

```
from connect import *
```

The `connect` module defines the `get_current`-method, described in section 3.1, and the method `CompositeAction`. CompositeAction can be used to group several actions together in the undo history, and is described in detail in *The CompositeAction method* below.

The CompositeAction method

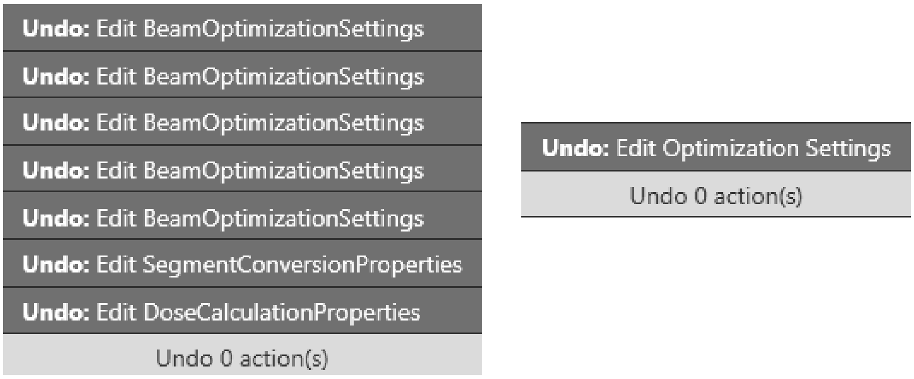
The `CompositeAction` method takes one string parameter and is used to group several actions as one undo-event. The string parameter sets the name of the grouped undo-event. The next code snippet shows how this method can be used to group the update of the optimization settings shown in section 3.9 to one undo-event:

```
plan = get_current("Plan")
po = plan.PlanOptimizations[0]
opt_param = po.OptimizationParameters
tss = opt_param.TreatmentSetupSettings[0]
# Group as one undo-event
with CompositeAction('Edit Optimization Settings'):
   # Set iterations before segment conversions to 10
   opt_param.DoseCalculation.IterationsInPreparationsPhase = 10
   # Set the maximum number of segments for all beams combined to 60
   tss.SegmentConversion.MaxNumberOfSegments = 60
   # Do not allow beam split for any of the beams
   for bs in tss.BeamSettings:
      bs.AllowBeamSplit = False
```

Figure 6 shows screen shots of the undo stack for a treatment plan with five beams without and with the use of `CompositeAction`.

It should be noted that `try-except`-clauses do not work perfectly with `CompositeAction`. If a `try-except`-clause is used to circumvent a RayStation exception inside a

`CompositeAction`-clause, no error messages will occur but the `CompositeAction` will cancel all actions and no changes will be made.



Undo: Edit BeamOptimizationSettings
Undo: Edit BeamOptimizationSettings
Undo: Edit BeamOptimizationSettings
Undo: Edit BeamOptimizationSettings
Undo: Edit BeamOptimizationSettings
Undo: Edit SegmentConversionProperties
Undo: Edit DoseCalculationProperties
Undo 0 action(s)

Undo: Edit Optimization Settings
Undo 0 action(s)

Figure 6. The undo stack for a treatment plan with five beams, without and with the use of `CompositeAction`.

5 EXAMPLE CODE

This chapter contains code examples. Slight changes have been made in order to make the code fit to the page widthwise. For example, some line breaks have been added after the periods in variable names. However, when writing the code these line breaks must not be included.

In this chapter

This chapter contains the following sections:

5.1 RECORDED SCRIPT

In this section a script generated by recording actions in the GUI is shown. Starting from an SMLC plan with no beams and no optimization functions, the following actions are performed:

- Add a beam with isocenter at the center of the ROI named "PTV" and with gantry, couch and collimator angle set to 0º.
- Make four copies of the created beam, making the total number of beams for the current beam set five.
- For each of the copied beams, edit the gantry angle to make the beams equidistantly spaced.
- Rename the beams so that each beam is named "Beam(i)", where (i) is the gantry angle of the beam.
- Add a uniform dose objective function to the ROI named "PTV" with dose level 7000 and weight 100.
- Add a maximum EUD objective function to the ROI named "Rectum" with dose level 3000, EUD parameter A 2 and weight 1.
- Add a maximum EUD objective function to the ROI named "Bladder" with dose level 3000, EUD parameter A 2 and weight 1.
- Add a dose fall-off objective function to the ROI named "External" with high dose level 7000, low dose level 3000, low dose distance 2 and weight 50.
- Start optimization.
- Compute dose.

Example of a recorded script

The recorded script is shown below.

```
# Script recorded

# RayStation version:
# Selected patient: ...

from connect import *

plan = get_current("Plan")
beam_set = get_current("BeamSet")

with CompositeAction('Add beam (Beam 0, Beam Set: SMLC)'):

   retval_0 = beam_set.CreatePhotonBeam(Energy=6,
              IsocenterData={ 'Position': { 'x': -1.03, 'y': 12.79,'z': 1.26 },
                                          'NameOfIsocenterToRef': "",
                                          'Name': "Iso 2",
                                          'Color': "98, 184, 234" },
              Name="Beam 0", Description="", GantryAngle=0, CouchAngle=0,
              CollimatorAngle=0)

   retval_0.SetBolus(BolusName="")

   beam_set.Beams['Beam 0'].BeamMU = 0

   # CompositeAction ends.

beam_set.CopyBeam(BeamName="Beam 0")

beam_set.CopyBeam(BeamName="Beam 1")

beam_set.CopyBeam(BeamName="Beam 2")

beam_set.CopyBeam(BeamName="Beam 3")

# Unscriptable Action 'Edit Beam (Beam 1, Beam Set: SMLC)' Completed :
EditPhotonBeamAction(...)

# Unscriptable Action 'Edit Beam (Beam 2, Beam Set: SMLC)' Completed :
EditPhotonBeamAction(...)

# Unscriptable Action 'Edit Beam (Beam 3, Beam Set: SMLC)' Completed :
EditPhotonBeamAction(...)

# Unscriptable Action 'Edit Beam (Beam 4, Beam Set: SMLC)' Completed :
EditPhotonBeamAction(...)

with CompositeAction('Add Optimization Function'):

   retval_1 = plan.PlanOptimizations[0].AddOptimizationFunction(FunctionType=
              "UniformDose", RoiName="PTV", IsConstraint=False,
              RestrictAllBeamsIndividually=False,RestrictToBeam=None,
              IsRobust=False, RestrictToBeamSet=None, UseRbeDose=False)

   plan.PlanOptimizations[0].Objective.ConstituentFunctions[0].
   DoseFunctionParameters.DoseLevel = 7000

   plan.PlanOptimizations[0].Objective.ConstituentFunctions[0].
   DoseFunctionParameters.Weight = 100
```

```
    # CompositeAction ends.

with CompositeAction('Add Optimization Function'):

    retval_2 = plan.PlanOptimizations[0].AddOptimizationFunction(FunctionType=
               "MaxEud", RoiName="Rectum", IsConstraint=False,
               RestrictAllBeamsIndividually=False, RestrictToBeam=None,
               IsRobust=False, RestrictToBeamSet=None, UseRbeDose=False)

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[1].
    DoseFunctionParameters.DoseLevel = 3000

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[1].
    DoseFunctionParameters.EudParameterA = 2

    # CompositeAction ends.

with CompositeAction('Add Optimization Function'):

    retval_3 = plan.PlanOptimizations[0].AddOptimizationFunction(FunctionType=
               "MaxEud", RoiName="Bladder", IsConstraint=False,
               RestrictAllBeamsIndividually=False, RestrictToBeam=None,
               IsRobust=False, RestrictToBeamSet=None, UseRbeDose=False)

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[2].
    DoseFunctionParameters.DoseLevel = 3000

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[2].
    DoseFunctionParameters.EudParameterA = 2

    # CompositeAction ends.

with CompositeAction('Add Optimization Function'):

    retval_4 = plan.PlanOptimizations[0].AddOptimizationFunction(FunctionType=
               "DoseFallOff", RoiName="External", IsConstraint=False,
               RestrictAllBeamsIndividually=False, RestrictToBeam=None,
               IsRobust=False, RestrictToBeamSet=None, UseRbeDose=False)

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[3].
    DoseFunctionParameters.HighDoseLevel = 7000

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[3].
    DoseFunctionParameters.LowDoseLevel = 3000

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[3].
    DoseFunctionParameters.LowDoseDistance = 2

    plan.PlanOptimizations[0].Objective.ConstituentFunctions[3].
    DoseFunctionParameters.Weight = 50

    # CompositeAction ends.

plan.PlanOptimizations[0].RunOptimization()

beam_set.ComputeDose(ComputeBeamDoses=True, DoseAlgorithm="CCDose",
                     ForceRecompute=False)
```

This script is executable in its current form. However, since the action `EditPhotonBeam` is unscriptable, the gantry angles of the beams will not be edited. Even though this action is unscriptable, it is possible to edit the beams by changing the `Name` and `GantryAngle` properties of the beams directly. The following code snippet will edit the gantry angles of the beams in the current beam set to make the beams equidistant and update the beam names. This piece of code should be added to the recorded script to make it execute as intended.

```
for i, beam in enumerate(beam_set.Beams):
   beam.GantryAngle = i * 360 / beam_set.Beams.Count
   beam.Name = 'Beam ' + str(beam.GantryAngle)
```

Improvements to the recorded script example

There are additional changes that can be made to make the script more general and more readable:

- Define the parameter `ptv_name` at the beginning of the script to enable the user to easily set the name of the PTV ROI.
- Use the method `GetCenterOfRoi()` on the `RoiGeometry` object of the PTV ROI to get the coordinates of the isocenter. The hard-coded values will most likely give a very bad location of the isocenter for another case.
- Use the method `CreateDefaultIsocenterData` to create the isocenter data instead of using the hard-coded values for color and name.
- Replace the hard-coded value for the beam energy with the first photon beam energy of the machine.
- Get the name of the external ROI from the property `OutlineRoiGeometry` of the structure set of the plan instead of using the hard-coded name for the external ROI.
- Check that ROIs named "Rectum" and "Bladder" exist before trying to add objective functions for these ROIs.
- Replace the hard-coded value `CCDose` as the `DoseAlgorithm` parameter in the method `ComputeDose` with the accurate dose algorithm of the current beam set.
- Remove default arguments of actions.
- Use the `retval` parameters to set the properties of the objective constituent functions instead of accessing them with their hard-coded list indices.

With these changes, all the user needs to do to use the script for any empty SMLC plan is to set the parameter `ptv_name` at the beginning of the script. The updated script is shown below. Unnecessary line breaks have been removed.

```
from connect import *
# Import module sys to enable exiting the script if input is missing.
import sys

case = get_current("Case")
plan = get_current("Plan")
beam_set = get_current("BeamSet")
machine_db = get_current("MachineDB")

# Define ptv_name parameter.
ptv_name = 'PTV'
```

```
# Get the center of the PTV ROI for the structure set of the plan.
structure_set = plan.GetStructureSet()
try:
   ptv_center = structure_set.RoiGeometries[ptv_name].GetCenterOfRoi()
except:
   print 'Cannot access center of ROI {0}. Exiting script.'.format(ptv_name)
   sys.exit()

# Convert the ptv_center variable to a dictionary
iso_position = {'x':ptv_center.x, 'y':ptv_center.y, 'z':ptv_center.z}

# Create default isocenter data using iso_position.
iso_data = beam_set.CreateDefaultIsocenterData(Position=iso_position)

# Get the first energy of the machine used for this beam set.
machine_name = beam_set.MachineReference.MachineName
machine = machine_db.GetTreatmentMachine(machineName=machine_name,
                                         lockMode=None)
energy = machine.PhotonBeamQualities[0].NominalEnergy

retval_0 = beam_set.CreatePhotonBeam(Energy=energy,
                                     IsocenterData=iso_data,
                                     Name="Beam 0", Description="",
                                     GantryAngle=0, CouchAngle=0,
                                     CollimatorAngle=0)
beam_set.CopyBeam(BeamName="Beam 0")
beam_set.CopyBeam(BeamName="Beam 1")
beam_set.CopyBeam(BeamName="Beam 2")
beam_set.CopyBeam(BeamName="Beam 3")

for i, beam in enumerate(beam_set.Beams):
   beam.GantryAngle = i * 360 / beam_set.Beams.Count
   beam.Name = str(int(beam.GantryAngle))

po = plan.PlanOptimizations[0]
with CompositeAction('Add Optimization Function'):
   retval_1 = po.AddOptimizationFunction(FunctionType="UniformDose",
                                         RoiName=ptv_name)
   retval_1.DoseFunctionParameters.DoseLevel = 7000
   retval_1.DoseFunctionParameters.Weight = 100

roi_names = [r.Name for r in case.PatientModel.RegionsOfInterest]
# Check that an ROI named 'Rectum' exists and has a defined geometry in the current
# structure set before adding an objective function.
rectum = 'Rectum'
if rectum in roi_names and \
   structure_set.RoiGeometries[rectum].PrimaryShape != None:
   with CompositeAction('Add Optimization Function'):
      retval_2 = po.AddOptimizationFunction(FunctionType="MaxEud",
                 RoiName=rectum)
      retval_2.DoseFunctionParameters.DoseLevel = 3000
      retval_2.DoseFunctionParameters.EudParameterA = 2

# Check that an ROI named 'Bladder' exists and has a defined geometry in the current
# structure set before adding an objective function.
bladder = 'Bladder'
if bladder in roi_names and \
    structure_set.RoiGeometries[bladder].PrimaryShape != None:
    with CompositeAction('Add Optimization Function'):
       retval_3 = po.AddOptimizationFunction(FunctionType="MaxEud",
                                             RoiName=bladder)
       retval_3.DoseFunctionParameters.DoseLevel = 3000
       retval_3.DoseFunctionParameters.EudParameterA = 2

external_name = structure_set.OutlineRoiGeometry.OfRoi.Name
with CompositeAction('Add Optimization Function'):
   retval_4 = po.AddOptimizationFunction(FunctionType="DoseFallOff",
                                         RoiName=external_name)
   retval_4.DoseFunctionParameters.HighDoseLevel = 7000
   retval_4.DoseFunctionParameters.LowDoseLevel = 3000
   retval_4.DoseFunctionParameters.LowDoseDistance = 2
   retval_4.DoseFunctionParameters.Weight = 50

po.RunOptimization()
algorithm = beam_set.AccurateDoseAlgorithm.DoseAlgorithm
beam_set.ComputeDose(ComputeBeamDoses=True,DoseAlgorithm=algorithm)
```

5.2 CREATE A BACKUP OF A PATIENT

The following script shows how to create a backup of the current patient. This is a two-step process. In the first step, the patient info must be accessed. After this, a backup can be created. The patient info for the current patient is accessed by querying the patient database with a filter on the patient ID. If there are more than one patient with the same ID, the patient name is used to find the correct patient info. There is an option to create an anonymized backup by setting the `Anonymize` parameter to True. In this script, the backup is not anonymized.

```
# Create a backup of the currently loaded patient
patient_db = get_current('PatientDB')
patient = get_current('Patient')
backup_path = r'C:\Users\elihyn\Desktop'

# Access the patient info for the current patient
id = patient.PatientID
patient_info = patient_db.QueryPatientInfo(Filter={'PatientID':'^'+id+'$'})

if patient_info.Count > 1:
   # More than one patient with the same ID
   # Check name
   patient_name = patient.PatientName
   if '^' in patient_name:
      last_name, first_name = patient_name.split('^')
   else:
      last_name, first_name = patient_name, ''
   patient_info = next(pi for pi in patient_info if pi['LastName'] ==
last_name and pi['FirstName'] == first_name)
else:
   patient_info = patient_info[0]

# Backup patient, do not anonymize
patient_db.BackupPatient(PatientInfo=patient_info,
                         TargetPath=backup_path,
                         Anonymize=False)
```

5.3 DICOM IMPORT

RayStation scripting supports DICOM import by file and Query/Retrieve. DICOM import is not recordable.

Three groups of methods are available for import:

- Query data
- Import data as new patient
- Import data to existing patient

All query methods have the functionality to filter data for input to the import methods. The filtering is done through a parameter called 'SearchCriteria', which is a dictionary that is mandatory for every query method. The keys that can be used in the dictionary varies depending on the level of the query. Some of the keys are mandatory in which the query will match exactly on the value, whereas some keys are optional in which DICOM wildcard matching in the value is supported. To see the supported keys of all the queries, see *section 5.3.1 Query methods on page 47*.

Note: *DICOM wildcards for Patient ID and Patient's Name can only be used on patient level queries.*

5.3.1 Query methods

These methods are used for search and filtering DICOM data from file or SCP.

QueryPatientsFromPath

Searches DICOM files and returns matching patients.

Parameter	Description
Path	The path containing DICOM data.
SearchCriterias	The search criteria to filter patients. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID • PatientsName

<u>This method returns:</u>

A list of dictionaries containing patient information. Each dictionary contains the following patient information:

- PatientID
- PatientsName
- PatientsSex
- NumberOfRelatedStudies
- NumberOfRelatedSeries
- NumberOfRelatedInstances

QueryStudiesFromPath

Searches DICOM files and returns matching studies.

Parameter	Description
Path	The path containing DICOM data.
SearchCriterias	The search criteria to filter studies. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID (M) • PatientsName • StudyDate

This method returns:

A list of dictionaries containing study information. Each dictionary contains the following study information:

- PatientID
- PatientsName
- StudyInstanceUID
- StudyDate
- StudyID
- AccessionNumber
- StudyDescription
- ReferringPhysiciansName
- ModalitiesInStudy
- NumberOfStudyRelatedSeries
- NumberOfStudyRelatedInstances

QuerySeriesFromPath

Searches DICOM files and returns matching series.

Parameter	Description
Path	The path containing DICOM data.
SearchCriterias	The search criteria to filter series. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID (M) • PatientsName • StudyInstanceUID (M) • Modality • SeriesNumber

<u>This method returns:</u>

A list of dictionaries containing series information. Each dictionary contains the following series information:

- PatientID
- PatientsName
- StudyInstanceUID
- SeriesNumber
- SeriesInstanceUID
- SeriesDate
- SeriesTime
- SeriesDescription
- Modality
- FrameOfReferenceUID
- NumberOfSeriesRelatedInstances

QueryInstancesFromPath

Searches DICOM files and returns matching instances.

Parameter	Description
Path	The path containing DICOM data.
SearchCriterias	The search criteria to filter instances. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID (M) • PatientsName • StudyInstanceUID (M) • SeriesInstanceUID (M) • SOPClassUID • SOPInstanceUID

This method returns:

A list of dictionaries containing instance information. Each dictionary contains the following instance information:

- PatientID
- PatientsName
- StudyInstanceUID
- SeriesInstanceUID
- SOPClassUID
- SOPInstanceUID
- ImageType
- AcquisitionNumber

QueryPatientsFromRepository

Queries DICOM SCP and returns matching patients.

Parameter	Description
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
SearchCriterias	The search criteria to filter patients. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID • PatientsName

This method returns:

A list of dictionaries containing patient information. Each dictionary contains the following patient information:

- PatientID
- PatientsName
- PatientsSex
- NumberOfRelatedStudies
- NumberOfRelatedSeries
- NumberOfRelatedInstances

QueryStudiesFromRepository

Queries DICOM SCP returns matching studies.

Parameter	Description
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
SearchCriterias	The search criteria to filter studies. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID (M) • PatientsName (M) • StudyDate

<u>This method returns:</u>

A list of dictionaries containing study information. Each dictionary contains the following study information:

- PatientID
- PatientsName
- StudyInstanceUID
- StudyDate
- StudyID
- AccessionNumber
- StudyDescription
- ReferringPhysiciansName
- ModalitiesInStudy
- NumberOfStudyRelatedSeries
- NumberOfStudyRelatedInstances

QuerySeriesFromRepository

Queries DICOM SCP and returns matching series.

Parameter	Description
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
SearchCriterias	The search criteria to filter series. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID (M) • PatientsName (M) • StudyInstanceUID (M) • Modality • SeriesNumber

<u>This method returns:</u>

A list of dictionaries containing series information. Each dictionary contains the following series information:

- PatientID
- PatientsName
- StudyInstanceUID
- SeriesNumber
- SeriesInstanceUID
- SeriesDate
- SeriesTime
- SeriesDescription
- Modality
- FrameOfReferenceUID
- NumberOfSeriesRelatedInstances

QueryInstancesFromRepository

Queries DICOM SCP and returns matching instances.

Parameter	Description
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
SearchCriterias	The search criteria to filter instances. The keys that are not mandatory support DICOM matching. Possible keys: • PatientID (M) • PatientsName (M) • StudyInstanceUID (M) • SeriesInstanceUID (M) • SOPClassUID • SOPInstanceUID

This method returns:

A list of dictionaries containing instance information. Each dictionary contains the following instance information:

- PatientID
- PatientsName
- StudyInstanceUID
- SeriesInstanceUID
- SOPClassUID
- SOPInstanceUID
- ImageType
- AcquisitionNumber

5.3.2 Import data to new patient

These methods are used for importing new patients from DICOM files or SCP. New patient import always requires at least one image set.

ImportPatientFromPath

Imports all DICOM data from file that matches the given UIDs and creates a new patient.

Parameter	**Description**
Path	The path containing DICOM data.
SeriesOrInstances	List of dictionaries containing UIDs of the series or instances to import. Each dictionary must contain the following keys: • PatientID • StudyInstanceUID • SeriesInstanceUID If single instances should be imported, an additional key is required: • SOPInstanceUID
ImportFilter	Name of the DICOM import filter to be applied. If the import should be done without any filter, this parameter does not need to be specified.

ImportPatientFromRepository

Imports all DICOM data from a repository that matches the given UIDs and creates a new patient.

Parameter	Description
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
SeriesOrInstances	List of dictionaries containing UIDs of the series or instances to import. Each dictionary must contain the following keys: • PatientID • StudyInstanceUID • SeriesInstanceUID If single instances should be imported, an additional key is re-quired: • SOPInstanceUID
ImportFilter	Name of the DICOM import filter to be applied. If the import should be done without any filter, this parameter does not need to be specified.

5.3.3 Import data to existing patient

These methods are used for importing DICOM data from file or SCP to an existing patient.

ImportDataFromPath

Imports all DICOM data from file that matches the given UIDs to an existing patient.

Parameter	Description
Path	The path containing DICOM data.
SeriesOrInstances	List of dictionaries containing UIDs of the series or instances to import. Each dictionary must contain the following keys: • PatientID • StudyInstanceUID • SeriesInstanceUID If single instances should be imported, an additional key is required: • SOPInstanceUID
CaseName	Name of the target case or null/empty if a new case shall be created.
ImportFilter	Name of the DICOM import filter to be applied. If the import should be done without any filter, this parameter does not need to be specified.
AllowMismatchingPatientID	Boolean option to allow importing data with PatientID mismatching from the target patient. This parameter does not need to be specified if default behavior is desired. The default value for this option is `False`.

ImportDataFromRepository

Imports all DICOM data from repository that matches the given UIDs to an existing patient.

Parameter	Description
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
SeriesOrInstances	List of dictionaries containing UIDs of the series or instances to import. Each dictionary must contain the following keys: • PatientID • StudyInstanceUID • SeriesInstanceUID If single instances should be imported, an additional key is required: • SOPInstanceUID
CaseName	Name of the target case or null/empty if a new case shall be created.
ImportFilter	Name of the DICOM import filter to be applied. If the import should be done without any filter, this parameter does not need to be specified.
AllowMismatchingPatientID	Boolean option to allow importing data with PatientID mismatching from the target patient. This parameter does not need to be specified if default behaviour is desired. The default value for this option is `False`.

5.3.4 Examples

Example: Query DICOM data and import to a new patient

The following code snippet contains two examples of how to query DICOM data for a patient with ID "JD123", and if there is exactly one patient with this ID, then all CT series are imported. Two different approaches of how to filter the queried series on modality == CT are demonstrated.

The first example shows how to query and import DICOM data from path. The modality filtering is done after all series of the patient is returned by the query.

The second example shows how to query and import DICOM data from a repository. The modality filtering is done by adding 'Modality' to the search criteria when querying series. Also, in this example a DICOM import filter named 'CT filter' is used during import.

```
### Example 1:
patient_db = get_current('PatientDB')

# Folder to import from
path = r'C:\SSCP_Inbox'

# Patient ID for search criterias
patient_id = 'JD123'

# Query patients from path by Patient ID
matching_patients = patient_db.QueryPatientsFromPath(Path = path,
                    SearchCriterias = {'PatientID' : patient_id})
assert len(matching_patients) == 1, "Found more than 1 patient with
ID {}".format(patient_id)
matching_patient = matching_patients[0]

# Query all the studies of the matching patient
studies = patient_db.QueryStudiesFromPath(Path = path,
                                          SearchCriterias = matching_patient)

# Query all the series from all the matching studies
series = []
for study in studies:
   series += patient_db.QuerySeriesFromPath(Path = path,
                                            SearchCriterias = study)

# Filter queried series to only contain CT series
seriesToImport = [s for s in series if s['Modality'] == 'CT']

# Import the CT series
warnings = patient_db.ImportPatientFromPath(Path = path,
                                            SeriesOrInstances = seriesToImport)
print warnings

### Example 2:
patient_db = get_current('PatientDB')

# Set up connection parameter defining the SCP repository to query and import from
connection = {'Node': "localhost",
              'Port': 11112,
              'CallingAE': 'RAYSTATION',
              'CalledAE': 'PACS'}

# Patient ID for search criterias
patient_id = 'JD123'

# Name of DICOM import filter to use during import
import_filter = 'CT filter'

# Query patients from repository Patient ID
matching_patients = patient_db.QueryPatientsFromRepository(Connection = connection,
                   SearchCriterias = {'PatientID' : patient_id})
assert len(matching_patients) == 1, "Found more than 1 patient with
ID {}".format(patient_id)
```

```
matching_patient = matching_patients[0]

# Query all the studies from the matching patient
studies = patient_db.QueryStudiesFromRepository(Connection = connection,
          SearchCriterias = matching_patient)

# Query all the series with CT modality from all the matching studies
series = []
for study in studies:
   study['Modality'] = 'CT'
   series += patient_db.QuerySeriesFromRepository(Connection = connection,
                                                  SearchCriterias = study)

# Import all the queried CT series
warnings = patient_db.ImportPatientFromRepository(Connection = connection,
           SeriesOrInstances = seriesToImport,
           ImportFilter = import_filter)
print warnings
```

Example: Import DICOM data to current patient

It is also possible to import DICOM data to the current patient. The methods to use for this are:

- ImportDataFromPath - to import data from DICOM files in a path
- ImportDataFromRepository - to import data from a repository

Both these methods are defined on the Patient object. The parameters for these methods are the same as for the corresponding methods to import a new patient, except they got two additional parameters, `CaseName` and `AllowMismatchingPatientID`.

The following code snippet contains two examples of how to query DICOM data, import CT image sets to a new patient, and then import an additional MR image set to the newly created patient.

The first example shows how to query and import DICOM data from path. In this example, the MR image set is imported to the same case that was created during the first import.

The second example shows how to query and import DICOM data from a repository. In this example, the MR image set is imported to a separate case, a DICOM import filter named 'MR filter' is used and a mismatching patient ID is allowed.

```
### Example 1
patient_db = get_current('PatientDB')

# Folder to import from
path = r'C:\SSCP_Inbox'

# Patient ID for search criterias
patient_id = "JD123"

# Query patients from path by Patient ID
matching_patients = patient_db.QueryPatientsFromPath(Path = path,
                    SearchCriterias = {'PatientID' : patient_id})
assert len(matching_patients) == 1, "Found more than 1 patient with
ID {}".format(patient_id)
matching_patient = matching_patients[0]
```

```
# Query all the studies of the matching patient
studies = patient_db.QueryStudiesFromPath(Path = path,
                                          SearchCriterias = matching_patient)

# Query all the series from all the matching studies
series = []
for study in studies:
   series += patient_db.QuerySeriesFromPath(Path = path,
                                            SearchCriterias = study)

# Filter queried series to only contain CT series
seriesToImport = [s for s in series if s['Modality'] == 'CT']

# Import all the queried CT series
warnings = patient_db.ImportPatientFromPath(Path = path,
                                            SeriesOrInstances = seriesToImport)
print warnings

patient = get_current('Patient')
case = get_current('Case')

# Filter queried series to only contain MR series
seriesToImport = [s for s in series if s['Modality'] == 'MR']

# Import MR series from path to current patient
warnings = patient.ImportDataFromPath(Path = path,
                                      SeriesOrInstances = seriesToImport,
                                      CaseName = case.CaseName)
print warnings

# Save state
patient.Save()


### Example 2
patient_db = get_current('PatientDB')

# Set up connection parameter defining the SCP repository to query and import from
connection = {'Node': "localhost",
              'Port': 11112,
              'CallingAE': 'RAYSTATION',
              'CalledAE': 'PACS'}

# Patient ID for search criterias
patient_id = "JD123"

# Name of DICOM import filter to use during import to current
import_filter = 'MR filter'

# Query patients from repository Patient ID
matching_patients = patient_db.QueryPatientsFromRepository(Connection = connection,
                   SearchCriterias = {'PatientID' : patient_id})
assert len(matching_patients) == 1, "Found more than 1 patient with
ID {}".format(patient_id)
matching_patient = matching_patients[0]

# Query all the studies from the matching patient
studies = patient_db.QueryStudiesFromRepository(Connection = connection,
          SearchCriterias = matching_patient)

# Query all the series from all the matching studies
series = []
for study in studies:
  series += patient_db.QuerySeriesFromRepository(Connection = connection,
                                                 SearchCriterias = study)

# Filter queried series to only contain CT series
seriesToImport = [s for s in series if s['Modality'] == 'CT']

# Import all the queried CT series
warnings = patient_db.ImportPatientFromRepository(Connection = connection,
           SeriesOrInstances = seriesToImport)
print warnings

patient = get_current('Patient')

# Filter queried series to only contain MR series
seriesToImport = [s for s in series if s['Modality'] == 'MR']

# Import everything from path to current patient
warnings = patient.ImportDataFromRepository(Connection = connection,
                                            SeriesOrInstances = seriesToImport,
                                            ImportFilter = import_filter,
                                            AllowMismatchingPatientID = True)
print warnings

# Save state
patient.Save()
```

5.4 DICOM EXPORT

The following section will show how to perform DICOM export with scripting.

When exporting it is important to take notice of the warnings. By default the export action will stop and crash if there are any warnings present. In order to read the warnings the script should be implemented in a certain way. See *section 5.4.4 Example: DICOM Export on page 67* for more information.

5.4.1 General considerations

It is possible to do the same kind of export as can be done from the RayStation DICOM export dialog. It is possible to export to disk or to an SCP server, but not both at the same time.

There are two types of warnings; blocking and non-blocking warnings:

- **Blocking warnings**. The blocking warnings cannot be ignored and the script will always stop if blocking warnings are present.
- **Non-blocking warnings**. Non-blocking warnings can be ignored if desired. This can be done by setting the parameter `IgnorePreConditionWarnings` to `True`. However, it is important to once again implement the script in such a way that the user can see the warnings that have been ignored. See *section 5.4.4 Example: DICOM Export on page 67* for more information.

When executing the script it is important to assign the result of the script method to a variable. That variable is a JSON formatted string containing the following properties:

- **Comment** - a comment stating if the export was successful or not.
- **Warnings** - these are the non-blocking warnings that the user must read. If the parameter `IgnorePreConditionWarnings` is set to `False`, then these are the warnings that will block the export. When the user has handled them it is possible to choose to proceed with the export despite the warnings.
- **ExportNotifications** - contains notes about the successful export.

It is recommended to execute the script within a Try-Except block. If the script is stopped because of a non-blocking warning, the script will throw an exception and the warnings will be available in the JSON formatted string exception variable. The user can then handle the warnings from there and try again if desired. Should there be a blocking warning present, the exception variable will not contain a JSON formatted string, instead the execution log should clearly describe why the script was blocked.

5.4.2 Export settings parameters

Mandatory parameters are `ExportFolderPath` or all of the following parameters: `AEHostName`, `AEPort`, `CallingAETitle` and `CalledAETitle`. These are the parameters that define where the script should export to.

The export settings parameters are described in the following table.

Parameter	Description
ExportFolderPath	Path to a directory on disk. The directory needs to exist.
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
Anonymize	Boolean indicating if the export should anonymize the export. This will trigger the use of the parameters `AnonymizedName` and `AnonymizedId`.
AnonymizedName	The anonymized name to use if the parameter `Anonymize` is set to `True`.
AnonymizedId	The anonymized id to use if the parameter `Anonymize` is set to `True`.

Parameter	Description
DicomFilter	This parameter specifies the export filter, and it should be specified by name.
IgnorePreConditionWarnings	A boolean parameter that enables the script to proceed with the export even though non-blocking warnings are present.

5.4.3 Export type parameters

This section describes different kinds of export types and how to specify them.

The export types can be specified individually or several at once. All of these parameters are lists and that means that for instance if you want to export two examinations just add another with a comma separating them.

Code snippet: `Examinations = [examination1.Name, examination2.Name]`

Examinations

`Examinations` is a list of the examinations that shall be exported. The list specified by examination names.

Code snippet: `Examinations = [examination.Name]`

RtStructureSetsForExaminations

`RtStructureSetsForExaminations` is a list of the examination names for which the structure set shall be exported. The list specified by examination names.

Code snippet: `RtStructureSetsForExaminations = [examination.Name]`

RtStructureSetsReferencedFromBeamSets

`RtStructureSetsReferencedFromBeamSets` is a list of beam set identifiers that determines the referenced structure set that shall be exported. The identifier shall be specified as PlanName:DicomPlanLabel (e.g., "Plan 1:BS 1" for plan "Plan 1" with beam set "BS 1").

Code snippet: `RtStructureSetsReferencedFromBeamSets = ["%s:%s"%(plan.Name, beam_set.DicomPlanLabel)]`

Alternative code snippet: `RtStructureSetsReferencedFromBeamSets = [beam_set.BeamSetIdentifier()]`

BeamSets

`BeamSets` is a list of beam set identifiers that shall be exported. The identifier shall be specified as PlanName:DicomPlanLabel (ex. "Plan 1:BS 1" for plan "Plan 1" with beam set "BS 1").

Code snippet: `BeamSets = ["%s:%s"%(plan.Name, beam_set.DicomPlanLabel)]`

Alternative code snippet: `BeamSets = [beam_set.BeamSetIdentifier()]`

BeamSetDoseForBeamSets

`BeamSetDoseForBeamSets` is a list of beam set identifiers for which the beam set dose shall be exported. The identifier shall be specified as PlanName:DicomPlanLabel (e.g., "Plan 1:BS 1" for plan "Plan 1" with beam set "BS 1").

Code snippet: `BeamSetDoseForBeamSets = ["%s:%s"%(plan.Name, beam_set.DicomPlanLabel)]`

Alternative code snippet: `BeamSetDoseForBeamSets = [beam_set.BeamSetIdentifier()]`

BeamDosesForBeamSets

`BeamDosesForBeamSets` is a list of beam set identifiers for which all beam doses shall be exported. The identifier shall be specified as PlanName:DicomPlanLabel (e.g., "Plan 1:BS 1" for plan "Plan 1" with beam set "BS 1").

Code snippet: `BeamDosesForBeamSets = ["%s:%s"%(plan.Name, beam_set.DicomPlanLabel)]`

Alternative code snippet: `BeamDosesForBeamSets = [beam_set.BeamSetIdentifier()]`

SpatialRegistrationForExaminations

`SpatialRegistrationForExaminations` is a list of examination pairs for which the registration object shall be exported. The pair is specified as fromExaminationName:toExaminationName.

Code snippet: `SpatialRegistrationForExaminations = ["%s:%s"%(fromExamination.Name, toExamination.Name)]`

TreatmentBeamDrrImages

`TreatmentBeamDrrImages` is a list of beam set identifiers that determine all treatment beam DRRs that shall be exported. The identifier shall be specified as PlanName:DicomPlanLabel (e.g., "Plan 1:BS 1" for plan "Plan 1" with beam set "BS 1")

Code snippet: `TreatmentBeamDrrImages = ["%s:%s"%(plan.Name, beam_set.DicomPlanLabel)]`

Alternative code snippet: `TreatmentBeamDrrImages = [beam_set.BeamSetIdentifier()]`

If you want to specify a single beam or a specific `DrrSetting` other than `Default`, the identifier shall be specified as PlanName:DicomPlanLabel:BeamName:DrrSettingName (e.g., "Plan 1:BS 1:B 1:DRR 1" for plan "Plan 1", with beam set "BS 1", and beam "B 1" and `DrrSetting` "DRR 1"). The last two arguments are optional.

If a beam is not specified, all beams in the beam set will be exported. If the `DrrSetting` is not specified, the setting named "Default" will be used.

Code snippets:

```
TreatmentBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                            "", "")]
# All beams with Default DrrSetting.
```

```
TreatmentBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                           "", "DRR 1")]
# All beams with DrrSetting named "DRR 1".
```

```
TreatmentBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                           beam.Name, "")]
# Only the selected beam with Default DrrSetting.
```

```
TreatmentBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                           beam.Name, "DRR 1")]
# Only the selected beam with DrrSetting named "DRR 1".
```

SetupBeamDrrImages

`SetupBeamDrrImages` is a list of beam set identifiers for which all setup beam DRRs shall be exported.

The identifier shall be specified as PlanName:DicomPlanLabel (e.g., "Plan 1:BS 1" for plan "Plan 1" with beam set "BS 1").

Code snippet: `SetupBeamDrrImages = ["%s:%s"%(plan.Name, beam_set.DicomPlanLabel)]`

Alternative code snippet: `SetupBeamDrrImages = [beam_set.BeamSetIdentifier()]`

If you want to specify a single beam or a specific `DrrSetting` other than `Default`, the identifier shall be specified as PlanName:DicomPlanLabel:BeamName:DrrSettingName. For example, "Plan 1:BS 1:B 1:DRR 1" for plan "Plan 1", with beam set "BS 1", and beam "B 1" and DrrSetting "DRR 1". The last two arguments are optional.

If a beam is not specified, all beams in the beam set will be exported. If the `DrrSetting` is not specified, the setting named "Default" will be used.

Code snippes:

```
SetupBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                       "", "")]
# All beams with Default DrrSetting.
```

```
SetupBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                       "", "DRR 1")]
# All beams with DrrSetting named "DRR 1".
```

```
SetupBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                       beam.Name, "")]
# Only the selected beam with Default DrrSetting.
```

```
SetupBeamDrrImages = ["%s:%s:%s:%s"%(plan.Name, beam_set.DicomPlanLabel,
                       beam.Name, "DRR 1")]
# Only the selected beam with DrrSetting named "DRR 1".
```

5.4.4 Example: DICOM Export

The following code snippet shows how to perform a DICOM export:

```
from connect import *
import json
```

```
# Example on how to read the JSON error string.
def LogWarning(warning):
   try:
      jsonWarnings = json.loads(str(warning))
# If the json.loads() works then the script was stopped due to
# a non-blocking warning.
      print " "
      print "WARNING! Export Aborted!"
      print "Comment:"
      print " ",
      print jsonWarnings["Comment"]
      print "Warnings:"
# Here the user can handle the warnings. Continue on known warnings,
# stop on unknown warnings.
      for w in jsonWarnings["Warnings"]:
         print " ",
         print w
   except ValueError as error:
      print "Error occurred. Could not export."
# The error was likely due to a blocking warning, and the details should be stated
# in the execution log.

# This prints the successful result log in an ordered way.
def LogCompleted(completed):
   try:
      jsonWarnings = json.loads(str(completed))
      print " "
      print "Completed!"
      print "Comment:"
      print " ",
      print jsonWarnings["Comment"]
      print "Warnings:"
      for w in jsonWarnings["Warnings"]:
         print " ",
         print w
      print "Export notifications:"
# Export notifications is a list of notifications that the user should read.
      for w in jsonWarnings["ExportNotifications"]:
         print " ",
         print w
   except ValueError as error:
      print "Error reading completion messages."

case = get_current('Case')
examination = get_current('Examination')
plan = get_current('Plan')
beamset = get_current('BeamSet')

for_registration = case.Registrations[0]
# Frame of reference of the "To" examination.
to_for = for_registration.ToFrameOfReference
# Frame of reference of the "From" examination.
from_for = for_registration.FromFrameOfReference
# Find all examinations with frame of reference that matches 'to_for'.
to_examinations = [e for e in case.Examinations if
e.EquipmentInfo.FrameOfReference == to_for]
# Find all examinations with frame of reference that matches 'from_for'.
from_examinations = [e for e in case.Examinations if
e.EquipmentInfo.FrameOfReference == from_for]

try:
# It is not necessary to assign all of the parameters, you only need to assign the
# desired export items. In this example we try to export with
# IgnorePreConditionWarnings=False. This is an option to handle possible warnings.
   result = case.ScriptableDicomExport(ExportFolderPath='c:\\temp\\testexport\\',
   Examinations=[examination.Name],
   RtStructureSetsForExaminations=[examination.Name],
   BeamSets=[beamset.BeamSetIdentifier()],
   BeamSetDoseForBeamSets=[beamset.BeamSetIdentifier()],
   BeamDosesForBeamSets=[beamset.BeamSetIdentifier()],
   SpatialRegistrationForExaminations=["%s:%s"%(to_examinations[0].Name,
                                        from_examinations[0].Name)],
   TreatmentBeamDrrImages=[beamset.BeamSetIdentifier()],
   SetupBeamDrrImages=[beamset.BeamSetIdentifier()],
   RtStructureSetsReferencedFromBeamSets = [beamset.BeamSetIdentifier()],
   DicomFilter="",
   IgnorePreConditionWarnings=False)

# It is important to read the result event if the script was successful.
# This gives the user a chance to see possible warnings that were ignored, if for
# example the IgnorePreConditionWarnings was set to True by mistake. The result
# also contains other notifications the user should read.
   LogCompleted(result)
except SystemError as error:
# The script failed due to warnings or errors.
   LogWarning(error)
   print " "
   print "Trying to export again with IgnorePreConditionWarnings=True"
   print " "
   result = case.ScriptableDicomExport(ExportFolderPath='c:\\temp\\testexport\\',
   Examinations=[examination.Name],
   RtStructureSetsForExaminations=[examination.Name],
   BeamSets=[beamset.BeamSetIdentifier()],
   BeamSetDoseForBeamSets=[beamset.BeamSetIdentifier()],
   BeamDosesForBeamSets=[beamset.BeamSetIdentifier()],
   SpatialRegistrationForExaminations=["%s:%s"%(to_examinations[0].Name,
                                        from_examinations[0].Name)],
   TreatmentBeamDrrImages=[beamset.BeamSetIdentifier()],
   SetupBeamDrrImages=[beamset.BeamSetIdentifier()],
   RtStructureSetsReferencedFromBeamSets = [beamset.BeamSetIdentifier()],
   DicomFilter="",
   IgnorePreConditionWarnings=True)

# It is very important to read the result event if the script was successful.
# This gives the user a chance to see any warnings that have been ignored.
   LogCompleted(result)
```

5.5 DICOM QA EXPORT

DICOM QA export differs from regular scripted DICOM export as it uses another method to start the export and has different parameters. But the general considerations described in *section 5.4.1 General considerations on page 63* still apply.

5.5.1 QA export setting parameters

Mandatory parameters are `ExportFolderPath` or all of the following parameters: `AEHostName`, `AEPort`, `CallingAETitle` and `CalledAETitle`. These are the parameters that define where the script should export to.

The export setting parameters are described in the following table.

Parameter	Description
ExportFolderPath	Path to a directory on disk. The directory needs to exist.
Connection	Dictionary of connection parameters to DICOM SCP: • Node - IP address or hostname of remote SCP • Port - Port number of remote SCP • CallingAE - AE title of local SCU • Called AE - AE title of remote SCP
IgnorePreConditionWarnings	A boolean parameter that enables the script to proceed with the export even though non-blocking warnings are present.
QaPlanIdentity	Optional. Specifies if the patient module of the patient or phantom should be used in all of the exported files. Set to "Phantom" or "Patient". If left empty, the setting defined in Clinic Settings will be used.

5.5.2 QA export type parameters

This section describes different kinds of export types and how to specify them. The export types can be specified individually or several at once. To export a type, set the parameter to `True`. If nothing is set, they will not be exported.

Parameter	Description
ExportExamination	Set to `True` if the examination for the verification plan should be exported. `QaPlanIdentity` needs to be set to `Phantom`.
ExportExaminationStructure-Set	Set to `True` if the examination structure set for the verification plan should be exported. `QaPlanIdentity` needs to be set to `Phantom`.

Parameter	Description
ExportBeamSet	Set to `True` if the beam set for the verification plan should be exported.
ExportBeamSetDose	Set to `True` if the beam set dose for the verification plan should be exported.
ExportBeamSetBeamDose	Set to `True` if the beam set beam dose for the verification plan should be exported.

5.5.3 QA export code example

```
from connect import *
import json

# Example on how to read the JSON error string.
def LogWarning(warning):
   try:
      jsonWarnings = json.loads(str(warning))
# If the json.loads() works then the script was stopped due to
# a non-blocking warning.
      print " "
      print "WARNING! Export Aborted!"
      print "Comment:"
      print " ",
      print jsonWarnings["Comment"]
      print "Warnings:"
# Here the user can handle the warnings. Continue on known warnings,
# stop on unknown warnings.
      for w in jsonWarnings["Warnings"]:
         print " ",
         print w
   except ValueError as error:
      print "Error occurred. Could not export."
# The error was likely due to a blocking warning, and the details should be stated
# in the execution log.

# This prints the successful result log in an ordered way.
def LogCompleted(completed):
   try:
      jsonWarnings = json.loads(str(completed))
      print " "
      print "Completed!"
      print "Comment:"
      print " ",
      print jsonWarnings["Comment"]
      print "Warnings:"
      for w in jsonWarnings["Warnings"]:
         print " ",
         print w
      print "Export notifications:"
# Export notifications is a list of notifications that the user should read.
      for w in jsonWarnings["ExportNotifications"]:
```

5

```
         print " ",
         print w
   except ValueError as error:
      print "Error reading completion messages."

plan = get_current('Plan')
beamset = get_current('BeamSet')

# Get the verification plans. This gets all verification plans on a plan.
verificationPlans = plan.VerificationPlans
# In this example we are only interested in the
# first QA plan for the currently selected Beam Set
for verificationPlan in verificationPlans:
   if verificationPlan.OfRadiationSet == beamset:
      try:
# It is not necessary to assign all of the parameters, you only need to assign the
# desired export items. In this example we try to export with
# IgnorePreConditionWarnings=False. This is an option to handle possible warnings.
         result =
verificationPlan.ScriptableQADicomExport(ExportFolderPath='c:\\temp\\testexport\\',
                                    ExportExamination=True,
                                    ExportExaminationStructureSet=True,
                                    ExportBeamSet=True,
                                    ExportBeamSetDose=True,
                                    ExportBeamSetBeamDose=True,
                                    IgnorePreConditionWarnings=False)
# It is important to read the result event if the script was successful.
# This gives the user a chance to see possible warnings that were ignored, if for
# example the IgnorePreConditionWarnings was set to True by mistake. The result
# also contains other notifications the user should read.
         LogCompleted(result)
      except SystemError as error:
# The script failed due to warnings or errors.
         LogWarning(error)
         print " "
         print "Trying to export again with IgnorePreConditionWarnings=True"
         print " "

# It is very important to read the result event if the script was successful.
# This gives the user a chance to see any warnings that have been ignored.
         LogCompleted(result)
# Break because we found the first verification plan in the list
# on the currently selected Beam Set.
      break
```

5.6 IMPORT AND EXPORT DOSE VALUES

How to access dose distributions in RayStation is described in *section 4.8 Dose distributions on page 29*. This section contains example code for import and export of dose values. The first piece of code shows how to write the dose values of the fraction dose of the currently loaded beam set to a text file:

```
import platform

beam_set = get_current("BeamSet")
```

```
fraction_dose = beam_set.FractionDose
dose = list(fraction_dose.DoseValues.DoseData)
if platform.python_implementation() == "IronPython":
   dose = list(fraction_dose.DoseValues.DoseData)
   # Create the file "doses.txt" and open it for writing
   with open("doses.txt", "w") as dose_file:
      # Write the dose values, separated by line breaks
      for d in dose:
         dose_file.write(str(d)+"\n")
else:
   fraction_dose.DoseValues.DoseData.tofile("doses.txt",sep="\r\n")
```

The next example shows how to read dose values from a text file and set these dose values as the fraction dose of the currently loaded beam set. It is assumed that the dose values are separated by line breaks:

```
# Open file for reading.
with open("doses.txt","r") as dose_file:
   # Read everything from the file and split it to a list based on
   # escape character.
   dose_lines = dose_file.read().splitlines()
# Convert dose values from string to float.
dose = [float(d) for d in dose_lines]
# Set the dose values of the fraction dose
# of the currently loaded beam set.
beam_set = get_current("BeamSet")
beam_set.FractionDose.SetDoseValues(Array=dose, CalculationInfo="")
```

The dose values set by `SetDoseValues` are marked as non-clinical.

5.7 CREATE REPORT FOR A BEAM SET

The following script shows how to create a report for the currently loaded beam set. The name of the template to use for the report is accessed from the clinic database. The method for creating a report has a parameter for ignoring warnings from the report creation. If this parameter is set to `True`, no report is created if there are any warnings from the report creation. In this script, the method is first run without ignoring the warnings, and if there are any warnings, these are displayed using the `await_user_input` method. The user can then select to continue to run the script, in which case the warnings will be ignored and the report will be created, or to stop the script execution, in which case no report is created. The `await_user_input` method is described in *section 6.3 Pause a script and wait for user input on page 86*.

```
beam_set = get_current('BeamSet')
clinic_db = get_current('ClinicDB')

filename = r'C:\script_files\BeamSetReport.pdf'

# Access the first template for a plan report in the clinic database
site_settings = clinic_db.GetSiteSettings()
template = next(t for t in site_settings.ReportTemplates
                if t.Type == 'TreatmentPlanReport')

try:
   # Try to create report, do not ignore warnings
   beam_set.CreateReport(templateName=template.Name, filename=filename,
                         ignoreWarnings=False)
except SystemError as e:
   # Display the warnings to the user
   await_user_input('Please review the following warnings: {0}'.format(e))
   # Create report, ignore warnings
   beam_set.CreateReport(templateName=template.Name, filename=filename,
                         ignoreWarnings=True)
```

5.8 GET DOSE IMAGES FOR TOTAL PLAN DOSE

The following code snippet shows how to generate total plan dose images in the coronal, transversal and sagittal direction for the currently loaded plan, with the images centered on the position of the isocenter of the first beam of the first beam set of the plan:

```
# Get total plan dose images in all directions for the current plan
# Use the isocenter of the first beam of the first beam set
# as center position
iso = plan.BeamSets[0].Beams[0].Isocenter.Position
position = {'x':iso.x, 'y':iso.y, 'z':iso.z}

# Get images of total plan dose in coronal, transversal
# and sagittal direction
# The images will be focused on the isocenter
# The image size will be 300 x 300 points
# Focus will be on the ROI named 'External'
orientations = ['Coronal', 'Transversal', 'Sagittal']
points = [position, position, position]
focus = [True, True, True]
size = {'x':300, 'y':300}
output = plan.GetTotalPlanDoseImages(Orientations=orientations,
                                     Points=points,
                                     FocusOnIsocenter=focus,
                                     ImageSize=size,
                                     FocusOnRoi='External')

# Print the file locations of all the generated images
for filenames in dict(output).itervalues():
   for name in filenames:
      print name
```

5.9 COMPUTE DIFFERENTIAL DVH

The RayStation GUI does not display differential DVHs. However, from the scripting interface it is possible to compute the differential DVH values and either write them to a text file or plot them using external plotting tools. The following code snippet shows how to compute the differential DVHs and write the values to a text file:

```
import platform
import math
plan = get_current("Plan")
# Set the bin size and the output path for the text file.
bin_size = 10 # [cGy]
file_path = r'C:\output'
```

```
structure_set = plan.GetStructureSet()
# Get the names of all ROIs that have geometries in the structure set of the plan.
roi_names = [r.OfRoi.Name for r in structure_set.RoiGeometries
             if r.PrimaryShape != None]
plan.TreatmentCourse.TotalDose.UpdateDoseGridStructuresAndRecomputeInvalidatedDoses()
# Get plan dose.
if platform.python_implementation() == "IronPython":
   plan_dose = [d for d in plan.TreatmentCourse.TotalDose.DoseValues.DoseData]
else:
   plan_dose = plan.TreatmentCourse.TotalDose.DoseValues.DoseData.flatten()
# Will use the same number of bins for all ROIs.
num_of_bins = int(math.ceil(max(plan_dose)/bin_size))
# Loop over ROIs and compute differential DVHs.
# Write to text file where the filename is: 'file_path\' + ROI name + '.txt'
for roi in roi_names:
   # Get dose grid ROI representation.
   dgr = plan.TreatmentCourse.TotalDose.GetDoseGridRoi(RoiName=roi)
   roi_dose = [plan_dose[vi] for vi in dgr.RoiVolumeDistribution.VoxelIndices]
   differential_dvh = [0] * num_of_bins
   for rd in roi_dose:
      bin_number = int(rd / bin_size)
      differential_dvh[bin_number] += 1
   # Write computed DVH to text file.
   with open(file_path + r'\{0}.txt'.format(roi),'w') as dvh_file:
      for dd in differential_dvh:
         dvh_file.write(str(dd)+"\n")
```

5.10 COMPUTE EQUIVALENT UNIFORM DOSE

Equivalent uniform dose can be used in optimization functions in RayStation, but the GUI does not display the values of the equivalent uniform dose for parameters that differ from 1 (average dose).

The following code snippet shows a method that can be used to compute equivalent uniform dose for any value of the a-parameter:

```
import platform
def compute_eud(dose, roi_name, parameter_a):
   # Get the dose values from the dose distribution.
   if platform.python_implementation() == "IronPython":
      dose_values = [d for d in in dose.DoseValues.DoseData]
   else:
      dose_values = dose.DoseValues.DoseData.flatten()
   # Get the dose grid representation of the ROI.
   dgr = dose.GetDoseGridRoi(RoiName=roi_name)
   # Get indices and relative volumes.
   indices = dgr.RoiVolumeDistribution.VoxelIndices
   relative_volumes = dgr.RoiVolumeDistribution.RelativeVolumes
   dose_sum = 0
   # Scale factor to prevent overflow with large parameter_a values.
   scalefactor = 1/max(dose_values)
   # Sum the dose values and scale with scalefactor and parameter_a.
   for i, v in zip(indices, relative_volumes):
      d = dose_values[i] * scalefactor
      dose_sum += v * (d ** parameter_a)
```

```
    return 1/scalefactor * dose_sum ** (1 / float(parameter_a))

plan = get_current('Plan')
roi_name = 'Rectum'
parameter_a = 3
# Compute EUD of the current plan dose for ROI Rectum with parameter_a = 3
# and print the result.
eud = compute_eud(plan.TreatmentCourse.TotalDose, roi_name, parameter_a)
print 'EUD with parameter a {0} for ROI {1}: {2} cGy'.format(parameter_a,
      roi_name, eud)
```

5.11 RETRIEVE DOSE VALUES AT A GIVEN PATIENT DEPTH

The following script shows how to retrieve dose values at a given patient depth along the beam lines for each beam in the currently loaded beam set. The script computes the coordinates of the given depth for each beam from the gantry and couch angles and then retrieves the dose values for these points from either the beam dose distribution or the fraction dose distribution.

```
# Import wpf and System.Windows to be able to use MessageBox to display the values.
import sys, math, wpf
from System.Windows import *
depth = 1.5
beam_dose = True
scale_with_number_of_fractions = True

def get_converted_beam_angles(beam_set):
   # Method that converts gantry and couch angles for each beam
   # depending on the patient position.
   patient_position = beam_set.PatientPosition
   if patient_position == 'HeadFirstSupine':
      return [[math.pi / 180 * b.GantryAngle,
               math.pi / 180 * b.CouchAngle] for b in beam_set.Beams]
   elif patient_position == 'HeadFirstProne':
      return [[math.pi / 180 * (b.GantryAngle + 180),
               math.pi / 180 * (-b.CouchAngle)] for b in beam_set.Beams]
   elif patient_position == 'FeetFirstSupine':
      return [[math.pi / 180 * b.GantryAngle,
               math.pi / 180 * (b.CouchAngle + 180)] for b in beam_set.Beams]
   elif patient_position == 'FeetFirstProne':
      return [[math.pi / 180 * (b.GantryAngle + 180),
               math.pi / 180 * (-b.CouchAngle - 180)] for b in beam_set.Beams]

def get_beam_points(beam_set, depth, sad):
   # Method that retrieves the points positioned at the given
   # depth along the beam for each beam.
   angles = get_converted_beam_angles(beam_set)
   # The distance from the isocenter to the given depth for each beam.
   dist = [sad-b.GetSSD()-depth for b in beam_set.Beams]
   isocenters = [b.Isocenter.Position for b in beam_set.Beams]
   # The coordinates of the points at the given depth for each beam.
   return [{'x':isocenters[i].x+dist[i]*math.cos(angles[i][1])*
            math.sin(angles[i][0]), 'y':isocenters[i].y-dist[i]*
            math.cos(angles[i][0]), 'z':isocenters[i].z-dist[i]*
            math.sin(angles[i][1])*math.sin(angles[i][0])}
            for i, b in enumerate(beam_set.Beams)]
```

```
def get_dose_at_points(plan, beam_set, sad, depth):
   # Method that retrieves the dose at the given depth
   # for each beam.
   doses_per_beam = {}
   scale = beam_set.FractionationPattern.NumberOfFractions if \
           scale_with_number_of_fractions else 1
   # Get the coordinates of the depth for each beam.
   points = get_beam_points(beam_set, depth, sad)
   # Retrieve the dose values for the points.
   pointFoR = beam_set.FrameOfReference
   # Frame of reference defining the DICOM patient system for which the
   # patient specific coordinates in the beams are defined in.
   if beam_dose:
      for b, point in zip(beam_set.Beams, points):
          bd = next(d for d in beam_set.FractionDose.BeamDoses
                    if d.ForBeam.Name == b.Name)
          dose = bd.InterpolateDoseInPoint(Points=points,
PointFrameOfReference=pointFoR)
          doses_per_beam[b.Name] = dose * scale
   else:
      doses = plan.TreatmentCourse.TotalDose.InterpolateDoseInPoints
              (Points=points, PointFrameOfReference=pointFoR)
      for b, dose in zip(bs, doses):
          doses_per_beam[b.Name] = dose
   return doses_per_beam

# The script execution starts here.
# Try to load all the required objects.
try:
   plan = get_current('Plan')
   beam_set = get_current('BeamSet')
   machine_db = get_current('MachineDB')
except:
   print 'No beam set is currently loaded. Cannot access beam doses.'
   sys.exit()

# Get the machine used.
machine = machine_db.GetTreatmentMachine
         (machineName=beam_set.MachineReference.MachineName,lockMode=None)
# Get the source-to-axis distance for the machine.
# This is needed to compute the distance from the isocenter to the given depth
# for the beams.
sad = machine.Physics.SourceAxisDistance
# Retrieve the dose at the given depth for all beams in the beam set.
dose_at_points = get_dose_at_points(plan, beam_set, sad, depth)
# Format the retrieved dose values for display.
text = ['Point dose at depth {0} cm for beam {1}: {2:.0f} cGy'.format(depth, b.Name,
        dose_at_points[b.Name]) for b in beam_set.Beams]
# Display the retrieved dose values.
MessageBox.Show('\n'.join(text))
```

5.12 GET THE COORDINATE OF THE MAXIMUM DOSE OF A DOSE DISTRIBUTION

The method `GetCoordinateOfMaxDose` works on a dose distribution object, and returns the coordinate of the maximum dose of that dose distribution. The following code snippet shows how to get the coordinate of the maximum dose of the currently loaded plan:

```
# Get the coordinate of the maximum dose of the plan dose
dose = plan.TreatmentCourse.TotalDose
point = dose.GetCoordinateOfMaxDose()

# Print coordinate
print point.x, point.y, point.z
```

5.13 CREATE ROI FROM DOSE

It is possible to generate the geometry of an ROI from a given threshold dose level. The method is called `CreateRoiGeometryFromDose` and works on the `RegionOfInterest` object. The method takes two parameters, `DoseDistribution`, which is the dose distribution that shall be used for geometry creation, and `ThresholdLevel`, which is the dose level (in cGy) that shall be used as the threshold level. The following code snippet shows how to create an ROI of type “Control” and then create its geometry from all voxels in the current plan dose that has dose level above 6000 cGy:

```
# Access the plan dose
plan_dose = plan.TreatmentCourse.TotalDose
# Define the threshold level
threshold_level = 6000

# Create a new ROI and create its geometry from the plan dose
# and the threshold level
# Define the name of the ROI
roi_name = 'Control ' + str(threshold_level)

roi = case.PatientModel.CreateRoi(Name=roi_name, Color='Blue',
                                  Type='Control')
roi.CreateRoiGeometryFromDose(DoseDistribution=plan_dose,
                              ThresholdLevel=threshold_level)
```

5.14 GET THE COORDINATE WITH THE HIGHEST GRAY LEVEL INSIDE AN ROI

The method `GetCoordinateForMaxGrayLevel` returns the coordinate of the voxel which has the highest gray level among all voxels in the ROI geometry. If there are multiple voxels with the same gray level, the first voxel is returned. The following code snippet prints the coordinate with the highest gray level in the ROI "PTV_Prostate" on the examination "CT 1":

```
# Get the coordinate with the highest gray level
case = get_current('Case')
geometry = case.PatientModel.StructureSets['CT 1'].
           RoiGeometries['PTV_Prostate']
point = geometry.GetCoordinateForMaxGrayLevel()

# Print the coordinate
print point.x, point.y, point.z
```

5.15 SET TREAT OR PROTECT ROIS AND BEAM MARGINS

There are three scripting methods to handle treat and protect ROIs for photon beams. The first method is for setting a treat or protect ROI. It is called `SetTreatOrProtectRoi` and works on the `Beam` object. It has one parameter, `RoiName`, which is a string that specifies the name of the ROI that shall be set as a treat or protect ROI. The second method is for removing a treat or protect ROI. It is called `RemoveTreatOrProtectRoi`, works on the `Beam` object and has one parameter, `RoiName`, which is a string that specifies which treat or protect ROI that shall be removed. The third method is for setting beam margins for a beam that has at least one treat or protect ROI defined. It is called `SetBeamMargins` and works on the `Beam` object. It has four parameters, `TopMargin`, `BottomMargin`, `LeftMargin` and `RightMargin`, which are used to define the top, bottom, left and right margins respectively. The following code snippet shows how to set an ROI named “PTV” as the treat ROI for a beam and setting the right and left beam margins to 1 cm:

```
# Define the ROI name
roi_name = 'PTV'

# Set the ROI as a Treat or Protect ROI for the beam
beam.SetTreatOrProtectRoi(RoiName=roi_name)

# Set beam margins for the beam
beam.SetBeamMargins(LeftMargin=1, RightMargin=1, TopMargin=0, BottomMargin=0)
```

5.16 EDIT COLOR TABLES

The color maps can be edited from the scripting interface. The color maps are found in the `CaseSettings` object under the `Case` object. In order to work with color objects in the scripting interface, the `System.Drawing` namespace from .NET needs to be imported.

The following code snippet shows how to add and edit levels of the dose color table:

```
import platform
# Import the System.Drawing namespace.
import clr
clr.AddReference('System.Drawing')
import System.Drawing
# Get the dose color table for the current case.
case = get_current('Case')
dose_color_table = case.CaseSettings.DoseColorMap.ColorTable
# Create a color from ARGB-values.
color = System.Drawing.Color.FromArgb(255,255,0,0)
# Add this color at 10 percent level in the dose color table.
dose_color_table[10] = color
# Change the 50 percent level to the 45 percent level.
# Get the color.
color_50 = dose_color_table[50]
# Remove the entry at 50.
if platform.python_implementation() == "IronPython":
   dose_color_table.Remove(50)
else:
   del dose_color_table[50]
# Add a new entry with the same color at 45 percent level.
dose_color_table[45] = color_50
```

```
# Set the new color table.
case.CaseSettings.DoseColorMap.ColorTable = dose_color_table
```

5.17 COPY OPTIMIZATION FUNCTIONS

This example shows how to copy optimization functions from one plan to another plan. For simplicity, it is assumed that both plans belong to the same patient. It is possible to copy optimization functions from a plan for one patient to a plan for another patient, but then a new patient must be loaded and the existence of ROIs with the correct names must be checked which will make the script unnecessarily complicated.

There is no action to copy an optimization function from one plan to another, so the script must retrieve the necessary data and create new functions for the new plan. In the following example, it is assumed that only physical optimization functions are present.

```
case = get_current("Case")
# Get handles to "Original plan" and "Copy plan".
original_plan = case.TreatmentPlans["SMLC"]
new_plan = case.TreatmentPlans["SMLC 2"]
# Define a function that will retrieve the necessary information
# and put it in a dictionary.
def get_arguments_from_function(function):
   dfp = function.DoseFunctionParameters
   arg_dict = {}
   arg_dict['RoiName'] = function.ForRegionOfInterest.Name
   arg_dict['IsRobust'] = function.UseRobustness
   arg_dict['Weight'] = dfp.Weight
   if hasattr(dfp, "FunctionType"):
      if dfp.FunctionType == 'UniformEud':
         arg_dict['FunctionType'] = 'TargetEud'
      else:
         arg_dict['FunctionType'] = dfp.FunctionType
      arg_dict['DoseLevel'] = dfp.DoseLevel
      if 'Eud' in dfp.FunctionType:
         arg_dict['EudParameterA'] = dfp.EudParameterA
      elif 'Dvh' in dfp.FunctionType:
         arg_dict['PercentVolume'] = dfp.PercentVolume
   elif hasattr(dfp, 'HighDoseLevel'):
      # Dose fall-off function does not have function type attribute.
      arg_dict['FunctionType'] = 'DoseFallOff'
      arg_dict['HighDoseLevel'] = dfp.HighDoseLevel
      arg_dict['LowDoseLevel'] = dfp.LowDoseLevel
      arg_dict['LowDoseDistance'] = dfp.LowDoseDistance
   elif hasattr(dfp, 'PercentStdDeviation'):
      # Uniformity constraint does not have function type.
      arg_dict['FunctionType'] = 'UniformityConstraint'
      arg_dict['PercentStdDeviation'] = dfp.PercentStdDeviation
   else:
      # Unknown function type, raise exception.
      raise ('Unknown function type')
   return arg_dict

# Define a function that will use information in arg_dict to set
# optimization function parameters.
```

```
def set_function_arguments(function, arg_dict):
   dfp = function.DoseFunctionParameters
   dfp.Weight = arg_dict['Weight']
   if arg_dict['FunctionType'] == 'DoseFallOff':
      dfp.HighDoseLevel = arg_dict['HighDoseLevel']
      dfp.LowDoseLevel = arg_dict['LowDoseLevel']
      dfp.LowDoseDistance = arg_dict['LowDoseDistance']
   elif arg_dict['FunctionType'] == 'UniformityConstraint':
      dfp.PercentStdDeviation = arg_dict['PercentStdDeviation']
   else:
      dfp.DoseLevel = arg_dict['DoseLevel']
      if 'Eud' in dfp.FunctionType:
         dfp.EudParameterA = arg_dict['EudParameterA']
      elif 'Dvh' in dfp.FunctionType:
         dfp.PercentVolume = arg_dict['PercentVolume']

# Loop over all objectives and constraints in the original plan
# to retrieve information. It is assumed that the original plan
# only contains one beam set.
po_original = original_plan.PlanOptimizations[0]
arguments = [] # List to hold arg_dicts of all functions.
# Get arguments from objective functions.
if po_original.Objective != None:
   for cf in po_original.Objective.ConstituentFunctions:
      arg_dict = get_arguments_from_function(cf)
      arg_dict['IsConstraint'] = False
      arguments.append(arg_dict)
# Get arguments from constraint functions.
for cf in po_original.Constraints:
   arg_dict = get_arguments_from_function(cf)
   arg_dict['IsConstraint'] = True
   arguments.append(arg_dict)

# Add optimization functions to the new plan.
po_new = new_plan.PlanOptimizations[0]
for arg_dict in arguments:
   with CompositeAction('Add Optimization Function'):
      f = po_new.AddOptimizationFunction(FunctionType=arg_dict['FunctionType'],
          RoiName=arg_dict['RoiName'], IsConstraint=arg_dict['IsConstraint'],
          IsRobust=arg_dict['IsRobust'])
      set_function_arguments(f, arg_dict)
```

5.18 ADD A BIOLOGICAL OPTIMIZATION FUNCTION

The following code snippet shows how to add a biological optimization function for the first target of the current case that has a defined tissue name. The code uses an auxiliary method to access the available tumor stages for this tissue and uses the first tumor stage in the list as the tumor stage for the function.

```
# Access the first target ROI that has a tissue defined
rois = case.PatientModel.RegionsOfInterest
# Access the beam settings of the first beam
target = next(r for r in rois if
              r.OrganData.OrganType == 'Target' and
              r.OrganData.ResponseFunctionTissueName != '')
# Access the tumor stages for this target
tumor_stages = patient_db.GetTumorStagesForRoi(Roi=target)

# Add a biological objective function for this target
# using the first tumor stage
po = plan.PlanOptimizations[0]
tumor_stage = tumor_stages[0]
f = po.AddBiologicalOptimizationFunction(RoiName=target.Name,
                                         EndpointOrTumorStage=tumor_stage,
                                         UseRepair=False,
                                         UseRepopulation=False)
```

5.19 CLINICAL GOAL EVALUATION

In RayStation, there are methods for getting the value of the clinical goal for the total plan dose, and also for evaluating and getting the clinical goal value for an evaluation dose. The following code snippet shows how to access the goal value for the total plan dose, as well as evaluating the clinical goal for an evaluation dose and accessing the goal value.

```
case = get_current('Case')
plan = get_current('Plan')

# Access the first clinical goal defined for the plan
goal = plan.TreatmentCourse.EvaluationSetup.EvaluationFunctions[0]

# Access the current value of the goal on the total plan dose
# of the current plan
value_plan = goal.GetClinicalGoalValue()

# Access the first evaluation dose of the case
fraction_evaluation = case.TreatmentDelivery.FractionEvaluations[0]
evaluation_dose =
fraction_evaluation.DoseOnExaminations[0].DoseEvaluations[0]


# Evaluate the clinical goal on the evaluation dose
is_fulfilled =
  goal.EvaluateClinicalGoalForEvaluationDose(DoseDistribution=evaluation_dose,
                                         ScaleFractionDoseToBeamSet=True)

# Access the current value of the goal on the evaluation dose
value_eval =
  goal.GetClinicalGoalValueForEvaluationDose(DoseDistribution=evaluation_dose,
                                         ScaleFractionDoseToBeamSet=True)
```

5.20 EDIT PARAMETERS RELATED TO BEAM COMPUTATION SETTINGS FOR PROTON PBS PLANS

It is possible to edit parameters related to Beam Computation Settings for proton pencil beam scanning plans. The following code snippet shows how to set the energy layer spacing for the first beam in the first optimization problem of the plan to “Constant, 0.5 cm in water”:

```
# Access the optimization parameters of the first optimization problem
opt_param = plan.PlanOptimizations[0].OptimizationParameters
```

```
# Access the beam settings of the first beam
beam_settings = opt_param.TreatmentSetupSettings[0].BeamSettings[0]

# Set EnergyLayerSeparation to 0.5
# Set energy selection mode to 'ConstantLayerSeparation'
with CompositeAction('Edit spot pattern'):
   beam_settings.ScannedBeamProperties.EnergyLayerSeparation = 0.5
   beam_settings.ScannedBeamProperties.EnergySelectionMode =
      "ConstantLayerSeparation"
```

All edits of Beam Computation Settings are recordable, so to find the syntax for editing any other setting, just record your changes in the GUI. It should however be noted that for changes that require edits of multiple parameters, i.e. Energy layer spacing, Spot spacing, Lateral target margins and Layer repainting, the parameters must be changed in a specific order. This is typically not the order you get when you record the change. Thus, if you record a parameter change and you get an error similar to `SystemError: ValidationError: EnergyLayerSeparation must be defined if EnergySelectionMode is ConstantLayerSeparation.`, then switch the order in which you change your parameters and the script will run without errors.

5.21 ADDING ENERGY LAYERS FOR PBS ION BEAMS

The methods `AddEnergyLayer` and `AddEnergyLayerWithSpots` that were previously used for adding energy layers from scripting are no longer supported. The new method to use is `CreatePBSIonSegment`. It can be used both to add an energy layer with spots with arbitrary positions, just like the previous methods, and it can also be used to add an energy layer with a rectangular grid of spots, by specifying the size of the rectangular grid and the spot spacing. The syntax for creating an energy layer with spots in a rectangular grid can be found by recording adding an energy layer from the Energy Layers tab in Plan Optimization. The following code snippet shows how to add an energy layer with arbitrarily positioned spots.

```
# Access the first beam of the current beam set
beam = beam_set.Beams[0]

# Define positions of two spots, the first one
# at position (-0.1, -0.1)
# and the second one at position (0.1, 0,1)
spot_positions = [{'x':-0.1, 'y':-0.1}, {'x':0.1, 'y':0.1}]

# Define the spot weights for the spots, let the first be 0.2
# and the second 0.15
spot_weights = [0.2, 0.15]

# Define the nominal energy of the energy layer
nominal_energy = 200

# Add the energy layer
beam.CreatePBSIonSegment(NominalEnergy=nominal_energy,
                         SpotPositions=spot_positions,
                         SpotWeights=spot_weights)
```

5.22 ACCESS NOZZLE SETTING PARAMETERS FOR PROTON WOBBLING BEAMS

The nozzle setting parameters for proton wobbling beams cannot be accessed directly from the scripting interface. Therefore, access methods have been added for these parameters. Note that these methods only work for proton wobbling beams, if they are invoked for other types of ion beams, an error message will be shown. The following code snippet shows how to access the nominal energy, the first scatterer parameters and the maximum range for a proton wobbling beam:

```
beam_set = get_current('BeamSet')
beam = beam_set.Beams[0]

# Get the nominal energy
nominal_energy = beam.GetNominalEnergyForWobblingBeam()

# Get the first scatterer parameters
scatterer_parameters = beam.GetFirstScattererParametersForWobblingBeam()

# 'scatterer_parameters' is a dictionary containing the following keys:
# - Name: the name of the scatterer
# - Setting: the setting of the scatterer as a binary string
# - Physical thickness: the physical thickness of the scatterer
# - Thickness: the thickness of the scatterer, in cm water

# Get the maximum range
max_range = beam.GetMaximumRangeForMitsubishiAndSumitomoBeam()
```

5.23 SET RBE MODEL REFERENCE FOR A BEAM SET

For carbon beam sets, it is possible to set or remove an RBE model reference. The following code snippet shows how to set the RBE model of the current beam set to a model named ‘LEM1 Original’, and then remove the RBE model:

```
beam_set = get_current('BeamSet')
# Set the model 'LEM1 Original' as the RBE model of the current beam set
beam_set.SetRbeModelReference(Name='LEM1 Original')

# Remove the RBE model from the beam set
beam_set.SetRbeModelReference()
```

6 USER INTERFACE SCRIPTING

RayStation supports User Interface (UI) scripting which enables for example switching modules and sub-modules as well as pausing a script and waiting for user input.

6.1 OVERVIEW

The idea of User Interface (UI) scripting is to enable a better interaction between the scripting interface and the Graphical User Interface (GUI). This includes support for opening different modules and sub-modules from a script, selecting which views to show in a workspace, as well as pausing a script to ask the user to revise content or perform some manual tasks.

Note that UI scripting is only intended to be used for editing what you see in the GUI. Starting actions and accessing and editing values should be carried out using the regular scripting interface.

The User Interface is accessed using the command `ui = get_current('ui')`. Depending on the state of the GUI, the variable `ui` will have different sub-objects. The following is a short description of the most important sub-objects that can be accessed when the GUI shows an open module with no extra windows opened:

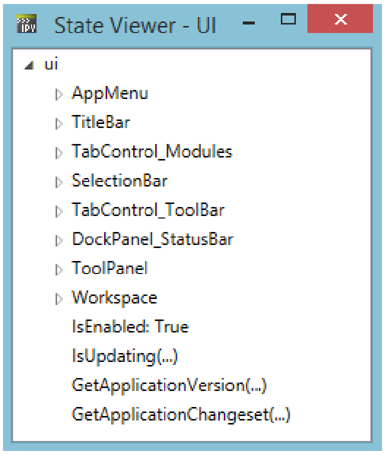
- `TitleBar`: the top bar of the RayStation window. From `TitleBar.MenuItem`, it is possible to access the different RayStation modules. The `MenuItem` object is also used to select which sub-module to switch to. For examples of how to switch modules and sub-modules using User Interface scripting, see *section 6.4 Examples on page 87*.
- `Workspace`: the main workspace of the current sub-module. From `Workspace` it is possible to switch between different views in the current workspace. See *Change a workspace view on page 88* for an example.
- `TabControl_ToolBar`: the toolbar for each sub-module. Most of the toolbar buttons are used to start actions that should not be started from the User Interface scripting, but e.g., selecting a deformable registration in the Deformable Registration sub-module is done from `TabControl_ToolBar`. See *Open a deformable registration on page 88* for an example.

6.2 UI STATE TREE

The module `statetree` (see *section 2.3 The state tree on page 18*) can generate a state tree for the User Interface. The command `statetree.RunUiStateTree()` creates a UI state tree from the console. The syntax for creating a UI state tree from a script is shown below.

```
import statetree
statetree.RunUiStateTree(True)
```

It should be noted that the User Interface state tree does not update when the GUI state is changed. Thus, if you for example change module and want to see what User Interface objects can be accessed from this module, you will have to open up a new User Interface state tree.



```
◢ ui
    ▷ AppMenu
    ▷ TitleBar
    ▷ TabControl_Modules
    ▷ SelectionBar
    ▷ TabControl_ToolBar
    ▷ DockPanel_StatusBar
    ▷ ToolPanel
    ▷ Workspace
      IsEnabled: True
      IsUpdating(...)
      GetApplicationVersion(...)
      GetApplicationChangeset(...)
```

Figure 7. The User Interface (UI) state tree

6.3 PAUSE A SCRIPT AND WAIT FOR USER INPUT

The connect module (see *section 4.14 Notes on RayStation scripting syntax on page 35*) contains the method `await_user_input`. This method is used to pause the script execution and allow user input from the GUI. It is then possible to continue the script execution from where the script was paused by clicking the continue button in the Script execution panel. The method takes a string parameter as input. This string is shown to the user as a pop-up dialog when the script is paused.

The following shows a script that creates MBS ROI:s for a prostate case and then adds a plan to the case. The script is paused after the MBS ROI:s are created and prompts the user to review the ROI:s before continuing the script execution. Figure 8 shows the dialog that is shown when the script is paused and Figure 9 shows the Script execution panel when the script is paused.

```
case = get_current("Case")
examination = get_current("Examination")
pm = case.PatientModel
# Create MBS ROIs.
pm.MBSAutoInitializer(MbsRois=[{'CaseType':"PelvicMale", 'ModelName':"Bladder",
                                'RoiName':"Bladder", 'RoiColor':"255, 255, 0"},
                               {'CaseType':"PelvicMale", 'ModelName':"Prostate",
                                'RoiName':"Prostate", 'RoiColor':"244, 164, 96"},
                               {'CaseType':"PelvicMale", 'ModelName':"Rectum",
                                'RoiName':"Rectum", 'RoiColor':"139, 69, 19"}],
                      CreateNewRois=True, Examination=examination,
                      UseAtlasBasedInitialization=True)
pm.AdaptMbsMeshes(Examination=examination,
                  RoiNames=["Bladder", "Prostate", "Rectum"])
# Prompt user for input.
await_user_input('Review the created MBS ROI:s')
```

```
# Script continues here.
# Add a new plan.
with CompositeAction('Add New Plan'):
   plan = case.AddNewPlan(PlanName='My scripted plan',
                          ExaminationName=examination.Name)
   plan.SetDefaultDoseGrid(VoxelSize={'x':0.3, 'y':0.3, 'z':0.3})
   beam_set = plan.AddNewBeamSet(Name='My scripted plan',
                                 ExaminationName=examination.Name,
                                 MachineName="Machine", Modality="Photons",
                                 TreatmentTechnique="SMLC",
                                 PatientPosition="HeadFirstSupine",
                                 NumberOfFractions=39)
```

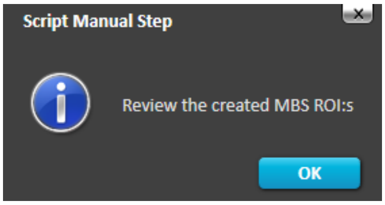


Figure 8. The dialog that is shown when the script is paused.

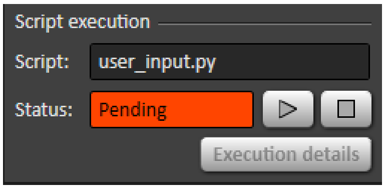


Figure 9. The Script execution panel that is shown when the script is paused.

6.4 EXAMPLES

Change module

The modules are found in the `MenuItem` list under `ui.TitleBar`. The modules can be accessed both by name and by index. The following code snippet shows how to change module to the Plan Design module.

```
# Access the user interface.
ui = get_current('ui')
# Access the MenuItem list.
menu_item = ui.TitleBar.MenuItem
# Open the Plan Design module.
menu_item['Plan Design'].Button_Plan_Design.Click()
```

Change sub-module

Switching to another sub-module is a two-step process. First, you need to perform a click on the module button to access its list of sub-modules. Then, it is possible to access the sub-modules under the `Popup` object of the `MenuItem` object. The sub-modules can be accessed both by name and by index. The following code snippet shows how to open the Image Registration sub-module in the Patient Modeling module.

```
# Access the user interface.
ui = get_current('ui')
# Access the MenuItem list.
menu_item = ui.TitleBar.MenuItem
# Open the popup window of the Patient Modeling module.
menu_item['Patient Modeling'].Click()
# Open Image Registration.
menu_item['Patient Modeling'].Popup.MenuItem['Image Registration'].Click()
```

Change a workspace view

The workspace part of a module has the sub-object `TabControl`. This is a list of all the tab controls in the workspace. The tab controls can be accessed using either indices or names. The name of a tab control is the name of the first tab in the tab control. The following code snippet shows how to open the Plan Setup module and show the Room View instead of the default 3D view.

```
# Access the user interface.
ui = get_current('ui')
# Open Plan Setup.
menu_item = ui.TitleBar.MenuItem
menu_item['Plan Design'].Click()
menu_item['Plan Design'].Popup.MenuItem['Plan Setup'].Click()
# Show the Room View instead of the default 3D view.
ui.Workspace.TabControl['3D'].TabItem['Room View'].Select()
```

Open a deformable registration

Selecting a deformable registration is done using the `TabControl_ToolPanel` object directly under the User Interface object. The following code snippet shows how to enter the Deformable Registration module and select to view the first deformable registration of the first deformable registration group of the currently loaded case. Note that the regular scripting interface is used to access the name of the registration group.

```
# Access the user interface.
ui = get_current('ui')
# Open Deformable Registration.
menu_item = ui.TitleBar.MenuItem
menu_item['Patient Modeling'].Click()
menu_item['Patient Modeling'].Popup.MenuItem['Deformable Registration'].Click()
# Click on Select Registration.
current_registration = ui.TabControl_ToolBar.ToolBarGroup['CURRENT REGISTRATION']
current_registration.RayPanelDropDownItem['Select registration'].
                            DropDownButton_Select_Registration.Click()
# Select a hybrid deformable registration.
context_menu = current_registration.
                    RayPanelDropDownItem['Select registration'].ContextMenu
hybrid_tree_item = context_menu.MenuItem[0].DefRegistrationTree.
                        TreeItem['Hybrid registrations']
# Get the name of the first structure registration group.
case = get_current('Case')
structure_reg_group = case.PatientModel.StructureRegistrationGroups[0]
```

```
structure_reg_group_name = structure_reg_group.Name
# Get the reference and the target of the first registration in this group.
reg = structure_reg_group.DeformableStructureRegistrations[0]
ref_name = reg.FromExamination.Name
target_name = reg.ToExamination.Name
# Select this registration.
tree_item_1 = "'{0}' Reference: '{1}'".format(structure_reg_group_name, ref_name)
tree_item_2 = "Target: '{0}'".format(target_name)
hybrid_tree_item.TreeItem[tree_item_1].TreeItem[tree_item_2].Select()
```

7 SCRIPTING IN CPYTHON

In RayStation CPython 2.7 and 3.6 has been added as an alternative scripting language alongside IronPython 2.7. The integration with .NET is handled using the Python package pythonnet. CPython has the advantage that it is compatible with many more third party packages. Scripting in CPython does not yet have all the functionality that scripting in IronPython has. It is for example not possible to generate GUIs in WPF or Winforms.

Differences in syntax between CPython and IronPython

The scripting syntax in CPython has been made as similar as possible to the scripting syntax in IronPython, but in some cases, IronPython scripts will need to be changed to work in CPython. The two most important changes in CPython scripting are:

1. Syntax from .NET no longer works on Python objects. Python syntax does however work in both IronPython and CPython. One example is that the length of a list can be accessed using `len(my_list)` in both CPython and IronPython, while the syntax `my_list.Count` only works in IronPython.

2. Arrays are numpy arrays instead of .NET arrays. This makes computations much easier, but it also changes a lot of syntax. For example, the syntax to flatten a multi-dimensional array (concatenate higher dimensions to create a vector) is:

   ```
   plan_dose = plan.TreatmentCourse.TotalDose.DoseValues.DoseData.flatten()
   ```

   instead of:

   ```
   plan_dose = [d for d in
   plan.TreatmentCourse.TotalDose.DoseValues.DoseData].
   ```

Making scripts work in both CPython and IronPython

The command `platform.python_implementation()` returns the name of the current Python implementation. This command can be used to make a script work with both IronPython and CPython, in cases where the interpreters have different syntax. The example below shows how this is done for flattening of an array.

```
import platform

if platform.python_implementation() == "IronPython":

    plan_dose = [d for d in
plan.TreatmentCourse.TotalDose.DoseValues.DoseData]

else: # CPython
```

```
plan_dose = plan.TreatmentCourse.TotalDose.DoseValues.DoseData.flatten()
```

Syntax differences between CPython 2 and CPython 3

There is often one syntax that works in both CPython 2 and CPython 3 and a version-specific syntax that works only in CPython 2. Some examples are listed in the table below:

Syntax	Description
`print 'text'`	Works only in CPython 2
`print('text')`	Works in both CPython 2 and CPython 3
`except Exception, exc:`	Works only in CPython 2
`except Exception as exc:`	Works in both CPython 2 and CPython 3
`xrange(100)`	Works only in CPython 2
`range`	works in both CPython 2 and CPython 3 (but is slower than xrange in CPython 2)
`len(my_array)/2`	Results in integer division in CPython 2 but in a double value in CPython 3.

In order to execute different code depending on the CPython version see *Making scripts work in both CPython and IronPython on page 91*.

8 ADVANCED TOPICS

This chapter describes some more advanced topics.

In this chapter

This chapter contains the following sections:

8.1 CREATE SCRIPTS WITH GUI

Sometimes the user should be able to supply information to a script. One way to facilitate this is to create a GUI. Since IronPython is part of the .NET framework, both WinForms and Windows Presentation Foundation (WPF) can be used to create GUIs. Scripting using CPython does not yet have support for creation of GUIs using WinForms and WPF.

8.1.1 GUI using WinForms

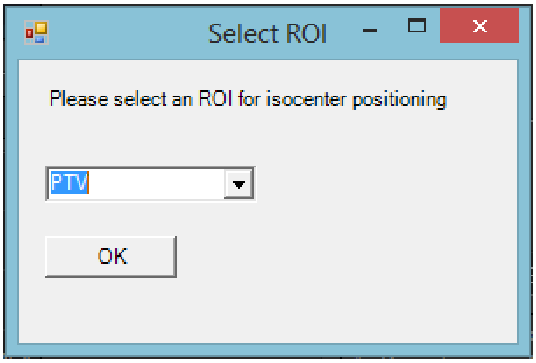
WinForms is the simpler package to use for creating GUI:s in IronPython. The GUI code can be incorporated in the same file as the code that interacts with RayStation.

Tutorials for WinForms can be found online. Most of them are written for C#, but it should be fairly easy to convert the examples to IronPython.

The WinForms example shown here creates a GUI that lets the user select an ROI and uses the center of this ROI to create an isocenter for the current beam set.

```
# WinForms example
# Creates a GUI that prompts the user to select an ROI
# The center of this ROI will be used as to create
# isocenter data for the current beam set
from connect import *

import clr
clr.AddReference("System.Windows.Forms")
clr.AddReference("System.Drawing")

from System.Windows.Forms import Application, Form, Label, ComboBox, Button
from System.Drawing import Point, Size

# Define Forms class that will prompt the user to select an
# ROI for creating isocenter data for the current beam set
class SelectROIForm(Form):

   def __init__(self, plan):
      # Set the size of the form
      self.Size = Size(300, 200)
      # Set title of the form
      self.Text = 'Select ROI'

      # Add a label
      label = Label()
      label.Text = 'Please select an ROI for isocenter positioning'
      label.Location = Point(15, 15)
      label.AutoSize = True
      self.Controls.Add(label)

      # Add a ComboBox that will display the ROI:s to select
      # Define the items to show up in the combobox,
      # we only want to show ROI:s with geometries
      # defined in the structure set of the current plan
      structure_set = plan.GetStructureSet()
      roi_names = [rg.OfRoi.Name for rg in
                   structure_set.RoiGeometries if rg.PrimaryShape != None]
      self.combobox = ComboBox()
```

```
        self.combobox.DataSource = roi_names
        self.combobox.Location = Point(15, 60)
        self.combobox.AutoSize = True
        self.Controls.Add(self.combobox)

        # Add button to press OK and close the form
        button = Button()
        button.Text = 'OK'
        button.AutoSize = True
        button.Location = Point(15, 100)
        button.Click += self.ok_button_clicked
        self.Controls.Add(button)
    def ok_button_clicked(self, sender, event):
        # Method invoked when the button is clicked
        # Save the selected ROI name
        self.roi_name = self.combobox.SelectedValue
        # Close the form
        self.Close()

# Access current plan and show the form
plan = get_current('Plan')

# Create an instance of the form and run it
form = SelectROIForm(plan)
Application.Run(form)

# Create isocenter data, use the center of the selected ROI as position
structure_set = plan.GetStructureSet()
roi_center = structure_set.RoiGeometries[form.roi_name].GetCenterOfRoi()
beam_set = get_current('BeamSet')
isocenter_data = beam_set.CreateDefaultIsocenterData(Position={'x':roi_center.x,
                                                              'y':roi_center.y,
                                                              'z':roi_center.z})
```



Please select an ROI for isocenter positioning

Figure 10. Screen shot of the GUI created from the WinForms example.

8.1.2 GUI using WPF

A WPF application in IronPython consists of two files, one Python file where the window or application class is defined and one XAML file where the graphics are added. To be able to create a simple GUI and integrate it with RayStation scripting, no advanced knowledge of WPF is required. The easiest way to get started is to look at a few examples and copy existing files and make changes.

Most WPF tutorials are written for C# or Visual Basic, but it is quite simple to convert the C# or Visual Basic code to IronPython. One fairly simple tutorial can be found at http://www.wpftutorial.net.

GUI that displays dose value for a given relative volume for an ROI

The first example shows how to create a GUI where the user can get the dose at a relative volume for an ROI. The considered dose distribution is the fraction dose of the currently loaded beam set. The first file is the Python file, `gui_example.py`:

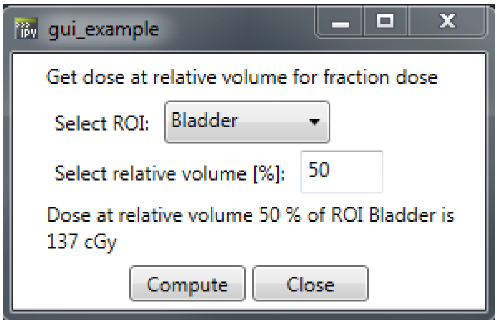
```
# gui_example.py
# A GUI example for displaying the relative volume dose of an ROI.
import wpf

from System.Windows import *
from System.Windows.Controls import *

class MyWindow(Window):
   def __init__(self, plan, beam_set):
      # Load xaml component.
      wpf.LoadComponent(self, 'gui_example.xaml')
      # Get the ROIs with non-empty geometries of the plan.
      roi_list = [r.OfRoi.Name for r in plan.GetStructureSet().RoiGeometries
                  if r.PrimaryShape != None]
      # Set ROIs to select from.
      self.SelectROI.ItemsSource = roi_list
      # Get the dose distribution.
      self.dose = beam_set.FractionDose

   def ComputeClicked(self, sender, event):
      ''' Gets the dose at the selected relative volume for the selected ROI '''
      # Get ROI name from combobox.
      roi_name = self.SelectROI.SelectedItem
      if roi_name == "":
         # No ROI name selected, cannot get dose.
         return
      try:
         # Get relative volume from textbox.
         rel_vol = float(self.RelVol.Text)
         rel_vol *= 0.01 # Convert from percent.
         if rel_vol < 0 or rel_vol > 1:
            # Must have relative volume between 0 and 1.
            return
      except:
         # Wrong input if it cannot be cast to float.
         return
      # Get the dose.
      dose = self.dose.GetDoseAtRelativeVolumes
             (RoiName=roi_name, RelativeVolumes=[rel_vol])
      # Change the text.
      text = "Dose at relative volume {0} % of ROI {1} is {2:.0f} cGy"
      self.RelVolText.Text = text.format(self.RelVol.Text, roi_name, dose[0])
      # Show the panel.
```

```
        self.RelVolPanel.Visibility = Visibility.Visible

    def CloseClicked(self, sender, event):
        # Close window.
        self.DialogResult = True

# Run in RayStation.
from connect import *
plan = get_current("Plan")
beam_set = get_current("BeamSet")
# Create window, arguments specified by __init__ method.
window = MyWindow(plan, beam_set)
# Show window.
window.ShowDialog()
```

The XAML file, `gui_example.xaml`, loaded by the Python file:

```
<!--gui_example.xaml
   The corresponding XAML file of gui_example.py-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="gui_example" Height="Auto" Width="300"
      SizeToContent="Height">
   <DockPanel>
      <TextBlock Text="Get dose at relative volume for fraction dose"
                 Margin="20,5,5,5" DockPanel.Dock="Top"/>
      <StackPanel Orientation="Horizontal" DockPanel.Dock="Top">
         <Label Content="Select ROI:" Margin="20,2,2,2"/>
         <ComboBox Name="SelectROI" Width="100" Margin="2"/>
      </StackPanel>
      <StackPanel Orientation="Horizontal" DockPanel.Dock="Top">
         <Label Content="Select relative volume [%]:"
                Margin="20,2,2,2"/>
         <TextBox Width="50" Margin="2" Name="RelVol"/>
      </StackPanel>
      <StackPanel Orientation="Horizontal" DockPanel.Dock="Top"
                  Visibility="Collapsed" Name="RelVolPanel">
         <TextBlock Margin="20,2,2,2" Name="RelVolText"
                    TextWrapping="Wrap" Width="260"/>
      </StackPanel>
      <StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom"
                  HorizontalAlignment="Center">
         <Button Content="Compute" Margin="2,5,2,5" Width="70"
                 Click="ComputeClicked"/>
         <Button Content="Close" Margin="2,5,2,5" Width="70"
                 Click="CloseClicked"/>
      </StackPanel>
   </DockPanel>
</Window>
```

A screenshot of the created GUI is shown in Figure 11.



Get dose at relative volume for fraction dose

Select ROI: Bladder

Select relative volume [%]: 50

Dose at relative volume 50 % of ROI Bladder is
137 cGy

Figure 11. Screenshot of the GUI created from `gui_example.py` and `gui_example.xaml`.

GUI for deleting evaluation doses

The next example shows the code for a GUI for deleting evaluation doses. The first file is the Python file, `delete_evaluation_doses.py`:

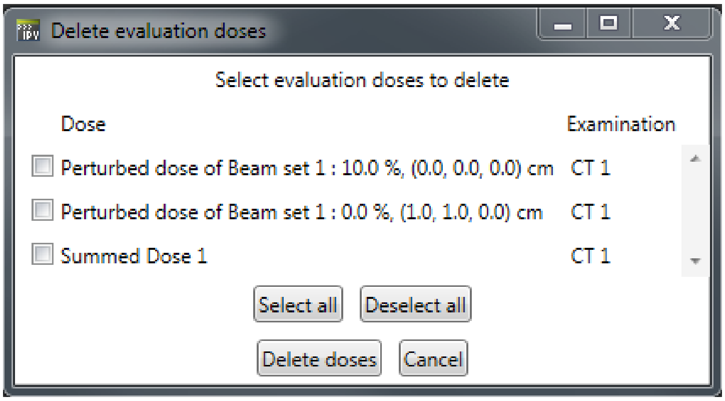
```
# delete_evaluation_doses.py
# GUI application for deleting evaluation doses.
import wpf

from System.Windows import *
from System.Windows.Controls import *

class MyWindow(Window):
   def __init__(self, case):
      wpf.LoadComponent(self, 'delete_evaluation_doses.xaml')
      # Set window as topmost window.
      self.Topmost = True
      # Start up window at the center of the screen.
      self.WindowStartupLocation = WindowStartupLocation.CenterScreen
      self.doses = {}
      # Find all evaluation doses of the patient.
      for fe in case.TreatmentDelivery.FractionEvaluations:
         for doe in fe.DoseOnExaminations:
            examination_name = doe.OnExamination.Name
            for de in doe.DoseEvaluations:
               self.add_row(de, examination_name)

   def DeleteClicked(self, sender, event):
      ''' Method that finds all checked doses and deletes them and closes
          the window '''
      for c in self.doseGrid.Children:
         if c.GetValue(Grid.ColumnProperty) == 0 and c.IsChecked:
            row = c.GetValue(Grid.RowProperty)
            try:
               self.doses[row].DeleteEvaluationDose()
            except:
               pass
      self.DialogResult = True

   def CancelClicked(self, sender, event):
      ''' Method that cancels the window '''
      self.DialogResult = False

   def SelectAllClicked(self, sender, event):
      ''' Method that checks the checkboxes for all doses in the window '''
      for c in self.doseGrid.Children:
         if c.GetValue(Grid.ColumnProperty) == 0:
            c.IsChecked = True

   def DeselectAllClicked(self, sender, event):
      ''' Method that unchecks the checkboxes for all doses in the window '''
      for c in self.doseGrid.Children:
         if c.GetValue(Grid.ColumnProperty) == 0:
            c.IsChecked = False

   def add_row(self, dose_distribution, examination_name):
      ''' Method that adds a row with a checkbox and a text describing
          the associated evaluation dose to the window '''
      row = self.doseGrid.RowDefinitions.Count
      self.doseGrid.RowDefinitions.Add(RowDefinition())
      cb = CheckBox()
      cb.Margin = Thickness(10,5,5,5)
      cb.SetValue(Grid.RowProperty, row)
      cb.SetValue(Grid.ColumnProperty,0)
      # Find out which dose type we have and set dose text accordingly.
      if dose_distribution.PerturbedDoseProperties != None:
         # Perturbed dose.
         rds = dose_distribution.PerturbedDoseProperties.RelativeDensityShift
         density = "{0:.1f} %".format(rds*100)
         iso = dose_distribution.PerturbedDoseProperties.IsoCenterShift
         isocenter = "({0:.1f}, {1:.1f}, {2:.1f}) cm".
                     format(iso.x, iso.z, -iso.y)
         beam_set_name = dose_distribution.ForBeamSet.DicomPlanLabel
         dose_text = "Perturbed dose of {0} : {1}, {2}".
                     format(beam_set_name, density, isocenter)
      elif dose_distribution.Name != "":
         # This is usually a summed dose.
         dose_text = dose_distribution.Name
      elif hasattr(dose_distribution, "ByStructureRegistration"):
         # Mapped dose.
         reg_name = dose_distribution.ByStructureRegistration.Name
         name = dose_distribution.OfDoseDistribution.ForBeamSet.DicomPlanLabel
         dose_text = "Deformed dose of {0} by registration {1}".
                     format(name, reg_name)
      else:
         # Neither perturbed, summed or mapped dose.
         dose_text = dose_distribution.ForBeamSet.DicomPlanLabel
      cb.Content = dose_text
      self.doseGrid.Children.Add(cb)
      tbe = TextBlock()
      tbe.Text = examination_name
      tbe.Margin = Thickness(5)
      tbe.TextAlignment = TextAlignment.Left
      tbe.VerticalAlignment = VerticalAlignment.Center
      tbe.SetValue(Grid.RowProperty, row)
      tbe.SetValue(Grid.ColumnProperty,1)
```

```
        self.doseGrid.Children.Add(tbe)
        self.doses[row] = dose_distribution

# Get current patient and display dialog.
from connect import *
case = get_current("Case")
dialog = MyWindow(case)
dialog.ShowDialog()
```

The second file is the XAML file, `delete_evaluation_doses.py`, loaded by the Python file:

```
<!-- delete_evaluation_doses.xaml
   XAML code for delete_evaluation_doses.py-->
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Delete evaluation doses" MaxHeight="600" Width="300"
   SizeToContent="WidthAndHeight">
   <DockPanel Name="mainDockPanel">
      <TextBlock Text="Select evaluation doses to delete" TextWrapping="Wrap"
                 Margin="5" TextAlignment="Center" DockPanel.Dock="Top"/>
      <DockPanel DockPanel.Dock="Top">
         <TextBlock Text="Dose" Margin="27,5,5,5" Grid.Row="0"
                    Grid.Column="0" DockPanel.Dock="Left"/>
         <TextBlock Text="Examination" Margin="5,5,20,5" Grid.Row="0"
                    Grid.Column="1" DockPanel.Dock="Right"
                    TextAlignment="Right"/>
      </DockPanel>
      <StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Center"
                  Orientation="Horizontal">
         <Button Content="Delete doses" Margin="5" Click="DeleteClicked"/>
         <Button Content="Cancel" Margin="5" Click="CancelClicked"/>
      </StackPanel>
      <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
                  HorizontalAlignment="Center">
         <Button Content="Select all" Margin="5" Click="SelectAllClicked"/>
         <Button Content="Deselect all" Margin="5" Click="DeselectAllClicked"/>
      </StackPanel>
      <ScrollViewer>
         <Grid DockPanel.Dock="Top" Name="doseGrid">
            <Grid.ColumnDefinitions>
               <ColumnDefinition/>
               <ColumnDefinition MinWidth="70"/>
            </Grid.ColumnDefinitions>
         </Grid>
      </ScrollViewer>
   </DockPanel>
</Window>
```

A screenshot of the GUI is shown in Figure 12.


Delete evaluation doses

Select evaluation doses to delete

Dose	Examination
☐ Perturbed dose of Beam set 1 : 10.0 %, (0.0, 0.0, 0.0) cm	CT 1
☐ Perturbed dose of Beam set 1 : 0.0 %, (1.0, 1.0, 0.0) cm	CT 1
☐ Summed Dose 1	CT 1

Figure 12. Screenshot of the GUI created by `delete_evaluation_doses.py` and `delete_evaluation_doses.xaml`.

Importing a script with a WPF GUI

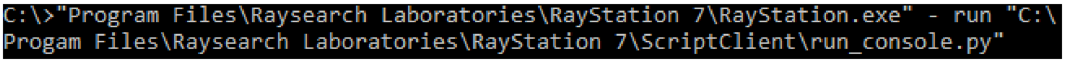
Scripts with GUIs created with WPF always contain at least two files. To import a script with a WPF GUI, you can let the python file generate the XAML-file from within the script.

With the latter method, you only need to import the python file, and when the python file is run, it will generate its own XAMLfile at its current location. The following code snippet shows how it can be done for `delete_evaluation_doses.py`. Only the relevant part of the file is shown here.

```
# delete_evaluation_doses.py
# GUI application for deleting evaluation doses.
import wpf

from System.Windows import *
from System.Windows.Controls import *
xaml_filename = "delete_evaluation_doses.xaml"

# Copy the contents of the XAML file and paste them as a string in
# the parameter xaml_string.
# Replace all line breaks with "\n".
# The parameter here shall contain the contents of delete_evaluation_doses.xaml.
xaml_string = "<!-- delete_evaluation_doses.xaml\n XAML code for
               delete_evaluation_doses.py-->\n"

# Create a file
with open(xaml_filename, "w") as f:
   f.write(xaml_string)

class MyWindow(Window):
   def __init__(self, patient):
      wpf.LoadComponent(self, xaml_filename)
      # The rest of the code follows here,
      # it is the same as in the earlier example.
```

8.2 RUN SCRIPTS FROM COMMAND LINE

It is possible to start RayStation from a command console and execute a script directly from the command line. The syntax is the path to the RayStation executable, followed by `-run`, and then the path to the script to run. The following is an example of the command line syntax to start RayStation and run the script `run_console.py`:



```
C:\>"Program Files\Raysearch Laboratories\RayStation 7\RayStation.exe" - run "C:\
Progam Files\Raysearch Laboratories\RayStation 7\ScriptClient\run_console.py"
```

The Python interpreter can be specified by appending the option `-interpreter` followed by the name of the interpreter. If the Python interpreter is not specified, the script is executed using the default interpreter, IronPython 2.7 (32-bit).

8.3 CREATE EXCEL FILES FROM IRONPYTHON

Microsoft Excel can be controlled programmatically from the .NET framework. Thus, it is possible to create Excel worksheets from IronPython. This can currently not be done from CPython, but there are third party packages for CPython that can be used to export data to Excel.

The first code snippet shows how to create an Excel worksheet and add data to a cell:

```
import clr

# Import libraries needed to communicate with Excel.
clr.AddReference("Office")
clr.AddReference("Microsoft.Office.Interop.Excel")
import Microsoft.Office.Interop.Excel as excel

# Open Excel application.
app = excel.ApplicationClass(Visible=True)
# Add a workbook.
workbook = app.Workbooks.Add(excel.XlWBATemplate.xlWBATWorksheet)
# Access the first worksheet of this workbook.
worksheet = workbook.Worksheets[1]
# Write 'Hello' in the cell A1.
worksheet.Range("A1").Value = 'Hello'
```

The next example is more advanced and shows how to retrieve dose statistics from the currently open plan and write the data to an Excel file:

```
import clr
import System.Array

clr.AddReference("Office")
clr.AddReference("Microsoft.Office.Interop.Excel")

import Microsoft.Office.Interop.Excel as excel

# Utility function to create 2-dimensional array.
def create_array(m, n):
   dims = System.Array.CreateInstance(System.Int32, 2)
   dims[0] = m
   dims[1] = n
   return System.Array.CreateInstance(System.Object, dims)
```

```
plan = get_current('Plan')

plan_dose = plan.TreatmentCourse.TotalDose

# Find all ROIs with defined geometry in the current plan.
structure_set = plan.GetStructureSet()
roi_names = [r.OfRoi.Name for r in structure_set.RoiGeometries
             if r.PrimaryShape != None]

# Create an Excel file.
try:
   # Open Excel with new worksheet.
   app = excel.ApplicationClass(Visible=True)
   workbook = app.Workbooks.Add(excel.XlWBATemplate.xlWBATWorksheet)
   worksheet = workbook.Worksheets[1]

   # Create an array that holds header information.
   header_row = create_array(1,5)
   header_row[0,0] = 'ROI'
   header_row[0,1] = 'Volume [cc]'
   header_row[0,2] = 'D99 [cGy]'
   header_row[0,3] = 'Average [cGy]'
   header_row[0,4] = 'D1 [cGy]'

   # Add header row to work sheet.
   startcell = worksheet.Cells(1, 1)
   header_range = worksheet.Range(startcell, startcell.Cells(1,5))
   header_range.Value = header_row

   # Create data array to hold ROI data.
   data_array = create_array(len(roi_names),5)

   # Add data for each ROI.
   for idx, roi in enumerate(roi_names):
      # Edit this if other dose statistics are desired.
      volume = plan_dose.GetDoseGridRoi(RoiName=roi).
               RoiVolumeDistribution.TotalVolume
      d99, d1 = plan_dose.GetDoseAtRelativeVolumes
                (RoiName=roi, RelativeVolumes=[.99,.01])
      average = plan_dose.GetDoseStatistic(RoiName=roi, DoseType='Average')
      data_array[idx,0] = roi
      data_array[idx,1] = volume
      data_array[idx,2] = d99
      data_array[idx,3] = average
      data_array[idx,4] = d1

   # Add ROI data array to work sheet.
   startcell = worksheet.Cells(2,1)
   data_range = worksheet.Range(startcell, startcell.Cells(len(roi_names),5))
   data_range.Value = data_array

   # Auto-fit the width of all columns.
   worksheet.Columns.AutoFit()

   filename = r"C:\script_files\DoseStatistics.xlsx"
   app.DisplayAlerts = False
   workbook.SaveAs(filename)
finally:
   # The following is needed for the Excel process to end when user closes
   # a worksheet.
   System.Runtime.InteropServices.Marshal.FinalReleaseComObject(worksheet)
   System.Runtime.InteropServices.Marshal.FinalReleaseComObject(workbook)
   System.Runtime.InteropServices.Marshal.FinalReleaseComObject(app)
   worksheet = None
   workbook = None
   excel = None
   System.GC.WaitForPendingFinalizers()
   System.GC.Collect()
```

8.4 CREATE PDF FILES FROM IRONPYTHON

PDF files can be generated from IronPython using the libraries PdfSharp and MigraDoc. These libraries are the same that are used for creating the reports in RayStation, so they are already included with your RayStation installation. These libraries cannot currently be used in CPython, but there are other thrid party packages for CPython that can be used to create PDF files.

The first example shows how to load the necessary libraries and generate a PDF file with the text "Hello World":

```
import sys, System

# Retrieve the path to RayStation.exe.
# This will only work if the script is run from within RayStation.
script_path = System.IO.Path.GetDirectoryName(sys.argv[0])
path = script_path.rsplit('\\',1)[0]
sys.path.append(path)

# Add references to libraries for PDF creation.
import clr
clr.AddReference("MigraDoc.DocumentObjectModel.dll")
clr.AddReference("MigraDoc.Rendering.dll")
clr.AddReference("PdfSharp.dll")

# Import objects for PDF creation.
from MigraDoc.DocumentObjectModel import Document
from MigraDoc.Rendering import PdfDocumentRenderer
from PdfSharp import Pdf

# Import object to open the created PDF document.
from System.Diagnostics import Process

# Create a PDF document.
doc = Document()
# Add a section to the document.
sec = doc.AddSection()
# Add a paragraph with the text "Hello world".
sec.AddParagraph('Hello world')

# Render the document.
renderer = PdfDocumentRenderer(True, Pdf.PdfFontEmbedding.Always)
renderer.Document = doc
renderer.RenderDocument()
# Save the file.
filename = r'C:\script_files\test.pdf'
```

```
renderer.PdfDocument.Save(filename)
# Display the PDF file.
process = Process()
process.StartInfo.FileName = filename
process.Start()
process.WaitForExit()
```

The next example is similar to the Excel example and will create a PDF file with dose statistics for the current plan:

```
import sys, System

filename = r'C:\script_files\test.pdf'

# Retrieve the path to RayStation.exe.
script_path = System.IO.Path.GetDirectoryName(sys.argv[0])
path = script_path.rsplit('\\',1)[0]
sys.path.append(path)

# Import objects for PDF creation.
import clr
clr.AddReference("MigraDoc.DocumentObjectModel.dll")
clr.AddReference("MigraDoc.Rendering.dll")
clr.AddReference("PdfSharp.dll")
from MigraDoc.DocumentObjectModel import Document, Colors, Unit,
                                         ParagraphAlignment, Orientation
from MigraDoc.DocumentObjectModel.Tables import Table
from MigraDoc.Rendering import PdfDocumentRenderer
from PdfSharp import Pdf

# Import object to open the created PDF document.
from System.Diagnostics import Process

# Get ROIs for dose statistics.
# Take all ROIs shown in the GUI that have defined geometries for the current plan.
structure_set = plan.GetStructureSet()
roi_names = [r.OfRoi.Name for r in structure_set.RoiGeometries
             if r.PrimaryShape != None]
plan_dose = plan.TreatmentCourse.TotalDose

# Create a PDF document.
doc = Document()

# Add a "normal" style and a "heading" style for the document.
# Normal style.
style = doc.Styles["Normal"]
style.Font.Name = "Verdana"
style.ParagraphFormat.SpaceAfter = 4
# Header style.
style = doc.Styles["Heading1"]
style.Font.Size = 20
style.Font.Bold = True
style.Font.Name = "Verdana"
style.Font.Color = Colors.DeepSkyBlue
style.ParagraphFormat.Alignment = ParagraphAlignment.Center
style.ParagraphFormat.SpaceAfter = 10
```

```
# Add a section and set landscape orientation.
sec = doc.AddSection()
pagesetup = doc.DefaultPageSetup.Clone()
pagesetup.Orientation = Orientation.Landscape
sec.PageSetup = pagesetup

# Add a title.
sec.AddParagraph("Dose Statistics", "Heading1")

# Create a table to hold the dose statistics.
table = Table()
for i in range(5):
   col = table.AddColumn(Unit.FromCentimeter(3))
      if i == 0:
         col.Format.Alignment = ParagraphAlignment.Left
      else:
         col.Format.Alignment = ParagraphAlignment.Right

# Add header row.
# Edit this if other dose statistics are desired.
header_row = table.AddRow()
header_row.Cells[0].AddParagraph('ROI')
header_row.Cells[1].AddParagraph('Volume [cc]')
header_row.Cells[2].AddParagraph('D99 [cGy]')
header_row.Cells[3].AddParagraph('Average [cGy]')
header_row.Cells[4].AddParagraph('D1 [cGy]')

# Add data rows for the ROIs.
# Edit this if other dose statistics are desired.
for roi in roi_names:
   row = table.AddRow()
   volume = plan_dose.GetDoseGridRoi(RoiName=roi).
            RoiVolumeDistribution.TotalVolume
   d99, d1 = plan_dose.GetDoseAtRelativeVolumes
             (RoiName=roi, RelativeVolumes=[.99, .01])
   average = plan_dose.GetDoseStatistic(RoiName=roi, DoseType='Average')
   row.Cells[0].AddParagraph(roi)
   row.Cells[1].AddParagraph('{0:.2f}'.format(volume))
   row.Cells[2].AddParagraph('{0:.2f}'.format(d99))
   row.Cells[3].AddParagraph('{0:.2f}'.format(average))
   row.Cells[4].AddParagraph('{0:.2f}'.format(d1))

# Add table to document.
sec.Add(table)

# Render the document, save it to file, and display the file.
renderer = PdfDocumentRenderer(True, Pdf.PdfFontEmbedding.Always)
renderer.Document = doc
renderer.RenderDocument()
renderer.PdfDocument.Save(filename)
process = Process()
process.StartInfo.FileName = filename
process.Start()
process.WaitForExit()
```

9 WORK-AROUNDS FOR UNSCRIPTABLE ACTIONS

This chapter contains an incomplete list of work-arounds for unscriptable actions. Note that more and more actions become scriptable for every new version of RayStation, thus some actions that were described with work-arounds in previous versions are no longer included as they are now scriptable.

Unscriptable action	Workaround
Save patient	Use `patient.Save()`.
Edit beam	Set the properties of the beam object, for example `beam.GantryAngle = 10`. Note that the collimator angle cannot be edited in this way.
DICOM import	Use `patient_db.ImportPatientFromPath(...)` to import a patient from DICOM files and use `patient_db.ImportPatientFromRepository(...)` to import a patient from DICOM repository. To import DICOM data to an existing patient, use `patient.ImportDicomDataFromPath(...)` to import from files and `patient.ImportDicomDataFromRepository(...)` to import from a repository. How to use these methods is described in *section 5.3 DICOM import on page 47*.
Open patient	Use `patient_db.QueryPatientInfo(...)` and `patient_db.LoadPatient(...)`. These methods are described in *section 4.2 Patients on page 24*.
Edit patient data	Set the properties of the patient object, for example `patient.BodySite = 'Brain'`. `BodySite, Comments, DateOfBirth, Diagnosis, Gender, PatientName` and `Physician` can be set from scripting.
Set level/window for examination	The level/window property can be set and it is located here: `examination.Series[0].LevelWindow`. This property is an `ExpandoObject` in IronPython and an `ExpandoDictionary` in CPython, where x corresponds to the level value in HU and y corresponds to the window value in HU. For more information see *ExpandoObjects on page 35*.

Unscriptable action	Workaround
Edit optimization settings	Set the properties you want to change directly. The optimization settings properties can be found in the `OptimizationParameters` object of a `PlanOptimization` object. There are a number of different properties and the easiest way to find them is to expand the `OptimizationParameters` object in the state tree. An example of how to edit some of the optimization settings is found in *section 4.10 Optimization functions and parameters on page 31*. Note that it is not possible to set the jaw limits by scripting.
Renumber beams	Set the property `Number` of the beam objects. It should be noted that two beams cannot have the same number, that is you need to give one beam a dummy number first to be able to switch the number of two beams.
Edit jaw or leaf positions	Set the property `JawPositions/LeafPositions`. For a beam, the jaw/leaf positions of segment `i` is found here: `beam.Segments[i].JawPositions/beam.Segments[i].LeafPositions`. For instructions on how to set list properties, see *Setting list properties on page 36*.
Edit plan	Some plan and beam set properties can be edited by scripting. For a plan, it is possible to edit `Comments, Name` and `PlannedBy`. For a beam set, it is possible to edit `DicomPlanLabel` (name) and `FractionationPattern.IsSequentiallyDelivered`.
Delete evaluation dose distribution	Use `dose_distribution.DeleteEvaluationDose()`.
Delete the geometry of an ROI	Use `roi_geometry.DeleteGeometry()`.
Delete a deformable registration	Use `case.DeleteDeformableRegistration(StructureRegistration=registration)`, where `registration` is the deformable registration that is to be deleted.
Edit color tables	Set the properties of the color table. An example of how to edit the dose color table is described in *section 5.16 Edit color tables on page 79*.
Edit dose specification point	Set the properties `Name` and `Coordinates` of the dose specification point object. The dose specification points are found in the list `DoseSpecificationPoints` of the `BeamSet` object. The `Coordinates` property is an `ExpandoObject` in IronPython and an `ExpandoDictionary` in CPython. For more information see *ExpandoObjects on page 35*.

Unscriptable action	Workaround
Remove holes from contour	Use `structure_set.SimplifyContours(RoiNames=[...], RemoveHoles3D=True)`, where `structure_set` is a `StructureSet` object. For full documentation of the method, check the state tree.
Translate, rotate or scale an ROI in 3D	Use `roi.TransformROI3D(Examination=examination, TransformationMatrix={...})`, where `roi` is a `RegionOfInterest` object. For full documentation of the method, check the state tree.
Change beam set dependencies for plans with multiple beam sets	Use `plan.UpdateDependency(DependentBeamSetName='bs2', BackgroundBeamSetName='bs1', DependencyUpdate=...)`, where `plan` is the plan containing the beam sets and `bs2` is the name of the beam set that is or shall be the dependent beam set, `bs1` is the name of the beam set that is or shall be the background beam set. `DependencyUpdate` can be set to either '`CreateDependency`' or '`RemoveDependency`'. For full documentation of the method, check the state tree.
Create plan report	Use `beam_set.CreateReport(…)`. The method is described in *section 5.7 Create report for a beam set on page 73*.
Backup patient	Use `patient_db.BackupPatient(…)`. The method is described in *section 5.2 Create a backup of a patient on page 46*.
Create ROI from dose	Use `roi.CreateRoiGeometryFromDose(…)`. The method is described in *section 5.13 Create ROI from dose on page 78*.
Set a Treat or Protect ROI	Use `beam.SetTreatOrProtectRoi(RoiName=roi_name)`. See *section 5.15 Set treat or protect ROIs and beam margins on page 79* or the state tree for more information.
Remove a Treat or Protect ROI	Use `beam.RemoveTreatOrProtectRoi(RoiName=roi_name)`. See the state tree for more information.
Set beam margins for a photon beam	Use `beam.SetBeamMargins(…)`. See *section 5.15 Set treat or protect ROIs and beam margins on page 79* or the state tree for more information.

CONTACT INFORMATION

RaySearch Laboratories AB (publ) - Head office
P.O. Box 3297
SE-103 65 Stockholm, Sweden
Phone: +46 8 510 530 00
Fax: +46 8 510 530 30
Visiting address:
Sveavägen 44
SE-111 34 Stockholm, Sweden
info@raysearchlabs.com
www.raysearchlabs.com

RaySearch Americas
Phone: +1 877 778 3849
RaySearch France
Phone: +33 975 433 632
RaySearch Korea
Phone: +82 10 2230 2046

RaySearch Belgium
Phone: +32 2 213 83 65
RaySearch Germany
Phone: +49 30 89 36 06 90
RaySearch Singapore
Phone: +65 81 28 59 80

RaySearch China
Phone: +86 137 0111 5932
RaySearch Japan
Phone: +81 3 4405 6902
RaySearch UK
Phone: +44 7508 426 563

RSL-D-RS-8B-SG-EN-1.0-2018-12-20
RayStation 8B Scripting Guideline

RaySearch Laboratories